Google Maps API Version 2 Documentation

The Google Maps JavaScript API lets you embed Google Maps in your own web pages. To use the API, you need to sign up for an API key, and then follow the instructions below.

The API is new, so there may be bugs and slightly less than perfect documentation. Bear with us as we fill in the holes, and join the Maps API discussion group to give feedback and discuss the API.

This documentation refers to Version 2 of the Maps API, launched on April 3, 2006. If your API uses Version 1 of the Maps API (i.e., you developed your site before April 3, 2006), you should attempt to upgrade your web site. Please see:

- Version 2 Upgrade Guide
- Google Maps API Version 1 Documentation

Table of Contents

Audience

This documentation is designed for people familiar with JavaScript programming and object-oriented programming concepts. You should also be familiar with Google Maps from a user's point of view. There are many JavaScript tutorials available on the Web

Introduction

The "Hello, World" of Google Maps

The easiest way to start learning about this API is to see a simple example. The following web page displays a 500x300 map centered on Palo Alto, California:

```
<!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=abcdefg"
            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>
```

You can download this example to edit and play around with it, but you'll have to replace the key in that file with your own Maps API key. (If you register a key for a particular directory, it works for all subdirectories as well.)

The URL in the example above (`http://maps.google.com/maps?file=api&v=2`) is the location of a JavaScript file that includes all of the symbols you need for placing Google Maps on your pages. Your page must contain a `script` tag pointing to that URL, using the key you got when you signed up for the API. If your Maps API key were "`abcdefg`", then your `script` tag might look like this:

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

The class that represents a map is `GMap2`. This class represents a single map on the page. You can create as many instances of this class as you want (one for each map on the page). When you create a new map instance, you specify a named element in the page (usually a `div` element) to contain the map. Unless you specify a size explicitly for the map, the map uses the size of the container to determine its size.

Browser Compatibility

The Google Maps API supports the same browsers as the Google Maps site. Since different applications require different behaviors for users with incompatible browsers, the Maps API provides a global method (`GBrowserIsCompatible()`) to check compatibility, but it does not have any automatic behavior when it detects an incompatible browser. The script `http://maps.google.com/maps?file=api&v=2` can be parsed in almost every browser without errors, so you can safely include that script before checking for

compatibility.

None of the examples in this document check for compatibility (other than the first example, above), nor do they display an error message for older browsers. Clearly real applications should do something more friendly with incompatible browsers, but we have omitted such checks to make the examples more readable.

Non-trivial applications will inevitably encounter inconsistencies between browsers and platforms. There is no simple solution to these problems, but the Google Maps API discussion group and quirksmode.org are both good resources to find workarounds.

XHTML and VML

We recommend that you use standards-compliant XHTML on pages that contain maps. When browsers see the XHTML `DOCTYPE` at the top of the page, they render the page in "standards compliance mode," which makes layout and behaviors much more predictable across browsers.

If you want to show polylines on your map (like the lines used by Google Maps to show driving directions), you need to include the VML namespace and some CSS code in your XHTML document to make everything work properly in IE. The beginning of your XHTML document should look something like this:

```
<!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" xmlns:v="urn:schemas-microsoft-com:vml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8"/>
    <title>My Google Maps Hack</title>
    <style type="text/css">
    v\:* {
      behavior:url(#default#VML);
    }
    </style>
    <script src="http://maps.google.com/maps?file=api&v=2&key=abcdefg"
            type="text/javascript"></script>
  </head>
```

API Updates

The `v=2` part of the URL `http://maps.google.com/maps?file=api&v=2` refers to "Version 2 " of the API. When we do a significant update to the API in the future, we will change the version number and post a notice on Google Code and the Maps API discussion group.

After a new version is released, we will try to run the old and new versions concurrently for about a month. After a month, the old version will be turned off, and code that uses the old version will no longer work.

The Maps team transparently updates the API with the most recent bug fixes and performance enhancements. These bug fixes should only improve performance and fix bugs, but we may inadvertently break some API clients. Please use the Maps API discussion group to

report such issues.

Geocoding

Geocoding is the process of converting addresses (like "1600 Amphitheatre Parkway, Mountain View, CA") into geographic coodinates (like latitude 37.423021 and longitude -122.083739), which you can use to place markers or position the map based on street addresses in your database or addresses supplied by users. The Google Maps API includes a geocoder that can be accessed via HTTP request or directly from within the JavaScript.

Routing and Local Search

The Google Maps API does not include routing services at this time. However, there are a number of free routing APIs on the web. If you would like to add local search capabilities to your site, you can use the Google AJAX Search API to embed a local search module into your site.

Marker Manager

Adding a large number of markers to a Google map may both slow down rendering of the map and introduce too much visual clutter, especially at certain zoom levels. The marker manager utility provides a solution to both of these issues, allowing efficient display of hundreds of markers on the same map and the ability to specify at which zoom levels markers should appear.

The marker manager offloads management of markers registered with the utility, keeping track of which markers are visible at certain zoom levels within the current view, and passing only these markers to the map for drawing purposes. The manager monitors the map's current viewport and zoom level, dynamically adding or removing markers from the map as they become active. In addition, by allowing markers to specify the zoom levels at which they display themselves, developers can implement marker clustering. Such management can greatly speed up map rendering and reduce visual clutter.

Examples

Each of the following examples shows only the relevant JavaScript code, not the full HTML file. You can plug the JS code into the skeleton HTML file shown earlier, or you can download the full HTML file for each example by clicking the link after the example.

The Basics

The following example creates a map and centers it on Palo Alto, California.

```
var map = new GMap2(document.getElementById("map"));
map.setCenter(new GLatLng(37.4419, -122.1419), 13);
```

View example (simple.html)

Map Movement and Animation

The following example displays a map, waits two seconds, and then pans to a new center point.

The `panTo` method centers the map at a given point. If the specified point is in the visible part of the map, then the map pans smoothly to the point; if not, the map jumps to the point.

```
var map = new GMap2(document.getElementById("map"));
map.setCenter(new GLatLng(37.4419, -122.1419), 13);
window.setTimeout(function() {
  map.panTo(new GLatLng(37.4569, -122.1569));
}, 1000);
```

View example (animate.html)

Adding Controls to the Map

You can add controls to the map with the `addControl` method. In this case, we add the built-in `GSmallMapControl` and `GMapTypeControl` controls, which let us pan/zoom the map and switch between Map and Satellite modes, respectively.

```
var map = new GMap2(document.getElementById("map"));
map.addControl(new GSmallMapControl());
map.addControl(new GMapTypeControl());
map.setCenter(new GLatLng(37.4419, -122.1419), 13);
```

View example (controls.html)

Event Listeners

To register an event listener, call the `GEvent.addListener` method. Pass it a map, an event to listen for, and a function to call when the specified event occurs. In the following example code, we display the latitude and longitude of the center of the map after the user drags the map.

```
var map = new GMap2(document.getElementById("map"));
GEvent.addListener(map, "moveend", function() {
  var center = map.getCenter();
  document.getElementById("message").innerHTML = center.toString();
});
map.setCenter(new GLatLng(37.4419, -122.1419), 13);
```

View example (event.html)

For more information about events, see the Events Overview. For a complete list of `GMap2` events and the arguments they generate, see GMap2.Events.

Opening an Info Window

To create an info window, call the `openInfoWindow` method, passing it a location and a DOM element to display. The following example code displays an info window anchored to the center of the map with a simple "Hello, world" message.

```
var map = new GMap2(document.getElementById("map"));
map.setCenter(new GLatLng(37.4419, -122.1419), 13);
map.openInfoWindow(map.getCenter(),
                   document.createTextNode("Hello, world"));
```

View example (infowindow.html)

Map Overlays

This example displays 10 markers at random locations on the map and a polyline with 5 random points. `GMarker` use the default Google Maps icon if no other icon is given. See Creating Icons for an example using custom icons.

Remember that you must include the VML namespace and CSS in your HTML document to view polylines in IE. See XHTML and VML for more information.

```
var map = new GMap2(document.getElementById("map"));
map.addControl(new GSmallMapControl());
map.addControl(new GMapTypeControl());
map.setCenter(new GLatLng(37.4419, -122.1419), 13);

// Add 10 markers in random locations on the map
var bounds = map.getBounds();
var southWest = bounds.getSouthWest();
var northEast = bounds.getNorthEast();
var lngSpan = northEast.lng() - southWest.lng();
var latSpan = northEast.lat() - southWest.lat();
for (var i = 0; i < 10; i++) {
  var point = new GLatLng(southWest.lat() + latSpan * Math.random(),
                          southWest.lng() + lngSpan * Math.random());
  map.addOverlay(new GMarker(point));
```

```
}

// Add a polyline with five random points. Sort the points by
// longitude so that the line does not intersect itself.
var points = [];
for (var i = 0; i < 5; i++) {
  points.push(new GLatLng(southWest.lat() + latSpan * Math.random(),
                          southWest.lng() + lngSpan * Math.random()));
}
points.sort(function(p1, p2) {
  return p1.lng() - p2.lng();
});
map.addOverlay(new GPolyline(points));
```

View example (overlay.html)

Click Handling

To trigger an action when a user clicks the map, register a listener for the `"click"` event on your `GMap2` instance. When the event is triggered, the event handler will receive two arguments: the marker that was clicked (if any), and the `GLatLng` of the clicked point. If no marker was clicked, the first argument will be `null`.

Note: Markers are the only built-in overlay that supports the `"click"` event. Other types of overlays, like `GPolyline`, are not clickable.

In the following example, when the visitor clicks a point on the map without a marker, we create a new marker at that point. When the visitor clicks a marker, we remove it from the map.

```
var map = new GMap2(document.getElementById("map"));
map.addControl(new GSmallMapControl());
map.addControl(new GMapTypeControl());
map.setCenter(new GLatLng(37.4419, -122.1419), 13);

GEvent.addListener(map, "click", function(marker, point) {
  if (marker) {
    map.removeOverlay(marker);
  } else {
    map.addOverlay(new GMarker(point));
  }
});
```

View example (click.html)

For more information about events, see the Events Overview. For a complete list of `GMap2` events and the arguments they generate, see

GMap2.Events.

Display Info Windows Above Markers

In this example, we show a custom info window above each marker by listening to the click event for each marker. We take advantage of JavaScript function closures to customize the info window content for each marker.

```
var map = new GMap2(document.getElementById("map"));
map.addControl(new GSmallMapControl());
map.addControl(new GMapTypeControl());
map.setCenter(new GLatLng(37.4419, -122.1419), 13);

// Creates a marker at the given point with the given number label
function createMarker(point, number) {
  var marker = new GMarker(point);
  GEvent.addListener(marker, "click", function() {
    marker.openInfoWindowHtml("Marker #<b>" + number + "</b>");
  });
  return marker;
}

// Add 10 markers to the map at random locations
var bounds = map.getBounds();
var southWest = bounds.getSouthWest();
var northEast = bounds.getNorthEast();
var lngSpan = northEast.lng() - southWest.lng();
var latSpan = northEast.lat() - southWest.lat();
for (var i = 0; i < 10; i++) {
  var point = new GLatLng(southWest.lat() + latSpan * Math.random(),
                          southWest.lng() + lngSpan * Math.random());
  map.addOverlay(createMarker(point, i + 1));
}
```

View example (markerinfowindow.html)

For more information about events, see the Events Overview. For a complete list of `GMap2` events and the arguments they generate, see GMap2.Events.

Tabbed Info Windows

Version 2 of the API introduces `openInfoWindowTabs()` and the `GInfoWindowTab` class to support info windows with multiple, named tabs. The example below displays a simple tabbed info window above a single marker.

```
var map = new GMap2(document.getElementById("map"));
map.addControl(new GSmallMapControl());
map.addControl(new GMapTypeControl());
map.setCenter(new GLatLng(37.4419, -122.1419), 13);

// Our info window content
var infoTabs = [
  new GInfoWindowTab("Tab #1", "This is tab #1 content"),
  new GInfoWindowTab("Tab #2", "This is tab #2 content")
];

// Place a marker in the center of the map and open the info window
// automatically
var marker = new GMarker(map.getCenter());
GEvent.addListener(marker, "click", function() {
  marker.openInfoWindowTabsHtml(infoTabs);
});
map.addOverlay(marker);
marker.openInfoWindowTabsHtml(infoTabs);
```

View example (tabbedinfowindow.html)

Creating Icons

This example creates a new type of icon, using the Google Ride Finder "mini" markers as an example. We have to specify the foreground image, the shadow image, and the points at which we anchor the icon to the map and anchor the info window to the icon.

```
var map = new GMap2(document.getElementById("map"));
map.addControl(new GSmallMapControl());
map.addControl(new GMapTypeControl());
map.setCenter(new GLatLng(37.4419, -122.1419), 13);

// Create our "tiny" marker icon
var icon = new GIcon();
icon.image = "http://labs.google.com/ridefinder/images/mm_20_red.png";
icon.shadow = "http://labs.google.com/ridefinder/images/mm_20_shadow.png";
icon.iconSize = new GSize(12, 20);
icon.shadowSize = new GSize(22, 20);
icon.iconAnchor = new GPoint(6, 20);
icon.infoWindowAnchor = new GPoint(5, 1);

// Add 10 markers to the map at random locations
var bounds = map.getBounds();
var southWest = bounds.getSouthWest();
var northEast = bounds.getNorthEast();
```

```
var lngSpan = northEast.lng() - southWest.lng();
var latSpan = northEast.lat() - southWest.lat();
for (var i = 0; i < 10; i++) {
  var point = new GLatLng(southWest.lat() + latSpan * Math.random(),
                          southWest.lng() + lngSpan * Math.random());
  map.addOverlay(new GMarker(point, icon));
}
```

View example (icon.html)

Using Icon Classes

In many cases, your icons may have different foregrounds, but the same shape and shadow. The easiest way to achieve this behavior is to use the copy constructor for the =GIcon= class, which copies all the properties over to a new icon which you can then customize.

```
var map = new GMap2(document.getElementById("map"));
map.addControl(new GSmallMapControl());
map.addControl(new GMapTypeControl());
map.setCenter(new GLatLng(37.4419, -122.1419), 13);

// Create a base icon for all of our markers that specifies the
// shadow, icon dimensions, etc.
var baseIcon = new GIcon();
baseIcon.shadow = "http://www.google.com/mapfiles/shadow50.png";
baseIcon.iconSize = new GSize(20, 34);
baseIcon.shadowSize = new GSize(37, 34);
baseIcon.iconAnchor = new GPoint(9, 34);
baseIcon.infoWindowAnchor = new GPoint(9, 2);
baseIcon.infoShadowAnchor = new GPoint(18, 25);

// Creates a marker whose info window displays the letter corresponding
// to the given index.
function createMarker(point, index) {
  // Create a lettered icon for this point using our icon class
  var letter = String.fromCharCode("A".charCodeAt(0) + index);
  var icon = new GIcon(baseIcon);
  icon.image = "http://www.google.com/mapfiles/marker" + letter + ".png";
  var marker = new GMarker(point, icon);

  GEvent.addListener(marker, "click", function() {
    marker.openInfoWindowHtml("Marker <b>" + letter + "</b>");
  });
  return marker;
}

// Add 10 markers to the map at random locations
```

```
var bounds = map.getBounds();
var southWest = bounds.getSouthWest();
var northEast = bounds.getNorthEast();
var lngSpan = northEast.lng() - southWest.lng();
var latSpan = northEast.lat() - southWest.lat();
for (var i = 0; i < 10; i++) {
  var point = new GLatLng(southWest.lat() + latSpan * Math.random(),
                          southWest.lng() + lngSpan * Math.random());
  map.addOverlay(createMarker(point, i));
}
```

View example (iconclass.html)

Draggable Markers

Markers are interactive objects that can be clicked on and dragged to a new location. In this example, we place a draggable marker on the map, and listen to a few of its simpler events. Draggable markers support four kinds of events -- `click`, `dragstart`, `drag` and `dragend`. By default, markers are clickable but not draggable, so they need to be initialized with the additional marker option `draggable` set to true. Draggable markers are also bouncy by default. If you don't like this behavior, just set the `bouncy` option to false and it will drop down gracefully.

```
var map = new GMap2(document.getElementById("map"));
var center = new GLatLng(37.4419, -122.1419);
map.setCenter(center, 13);

var marker = new GMarker(center, {draggable: true});

GEvent.addListener(marker, "dragstart", function() {
  map.closeInfoWindow();
  });

GEvent.addListener(marker, "dragend", function() {
  marker.openInfoWindowHtml("Just bouncing along...");
  });

map.addOverlay(marker);
```

View example (dragmarker.html)

Encoded Polylines

The `GPolyline` object within a Google map denotes a line as a series of points, making it easy to use but not necessarily compact. Long

and complicated lines require a fair amount of memory, and often may take longer to draw. Also, the individual segments within an unencoded polyline are drawn on a Google Map regardless of their resolution at larger zoom levels.

The Google Maps API also allows you to represent paths using encoded polylines, which specify a series of points within a `GPolyline` using a compressed format of ASCII characters. The encoded polyline also allows you to specify groups of zoom levels that should be ignored when drawing line segments; doing so allows you to specify how detailed a polyline should be at a given zoom level. Although more difficult to set up, encoded polylines can make your overlays draw much more efficiently.

For example, a `GPolyline` of 3 points (2 line segments) is normally represented as:

```
var polyline = new GPolyline([
    new GLatLng(37.4419, -122.1419),
    new GLatLng(37.4519, -122.1519),
    new GLatLng( 37.4619, -122.1819)
], "#FF0000", 10);
map.addOverlay(polyline);
```

An encoded `GPolyline` of these same points appears below (for now, don't worry about particulars of the encoding algorithm).

```
var encodedPolyline = new GPolyline.fromEncoded({
    color: "#FF0000",
    weight: 10,
    points: "yzocFzynhVq}@n}@o}@nzD",
    levels: "BBB",
    zoomFactor: 32,
    numLevels: 4
});
map.addOverlay(encodedPolyline);
```

There are two things to notice about this code.

1. First, the series of points is represented as a series of ASCII characters in the encoded polyline, while familiar latitude and longitudes are used in the basic `GPolyline`. The algorithm for creating these points as a series of encoded ASCII values is documented here. This algorithm is needed if you wish to calculate encoded polylines on the fly via a server process, for example. However, if you just wish to convert existing points given latitudes and longitudes, you can use our interactive utility.
2. Second, the encoded polyline also allows you to specify the maximum zoom level for each line segment to draw itself on a Google map. If a point is not shown on a higher zoom level, the path is simply drawn from the previous displayable point to the next displayable point. Note that this feature is not available within non-encoded `GPolyline`s and is especially useful for allowing fast drawing at high zoom levels, where the details of some line segments may not be relevant. For example, an encoded polyline representing a drive from New York City to Chicago should not care about the line segments representing particular streets in Manhattan when the map is zoomed out to the state level.

View example (polylineencoding.html)

Using XML and Asynchronous HTTP with Maps

In this example, we download a static file (`"data.xml"`) that contains a list of lat/lng coordinates in XML using the `GDownloadUrl` method. When the download completes, we parse the XML with `GXml` and create a marker at each of those points in the XML document.

```
var map = new GMap2(document.getElementById("map"));
map.addControl(new GSmallMapControl());
map.addControl(new GMapTypeControl());
map.setCenter(new GLatLng(37.4419, -122.1419), 13);

// Download the data in data.xml and load it on the map. The format we
// expect is:
// <markers>
//   <marker lat="37.441" lng="-122.141"/>
//   <marker lat="37.322" lng="-121.213"/>
// </markers>
GDownloadUrl("data.xml", function(data, responseCode) {
  var xml = GXml.parse(data);
  var markers = xml.documentElement.getElementsByTagName("marker");
  for (var i = 0; i < markers.length; i++) {
    var point = new GLatLng(parseFloat(markers[i].getAttribute("lat")),
                            parseFloat(markers[i].getAttribute("lng")));
    map.addOverlay(new GMarker(point));
  }
});
```

View example (async.html). This example uses the external XML data file data.xml.

Custom Map Controls

Version 2 of the Maps API introduces the ability to create custom map controls, like the built-in pan and zoom controls, by subclassing the built-in `GControl` class. In this example, we create a simple zoom control that has textual links rather than the graphical icons used in the standard Google Maps zoom control.

The `GTextualZoomControl` class defines the two required methods of the `GControl` interface: `initialize()`, in which we create the DOM elements representing our control; and `getDefaultPosition()`, in which we return the `GControlPosition` used if another position is not given when this control is added to a map. See the class reference for `GControl` for more information about the methods you can override when you create your custom controls.

All map controls should be added to the map *container* which can be accessed with the `getContainer()` method on `GMap2`.

```
// A TextualZoomControl is a GControl that displays textual "Zoom In"
// and "Zoom Out" buttons (as opposed to the iconic buttons used in
// Google Maps).
function TextualZoomControl() {
}
TextualZoomControl.prototype = new GControl();

// Creates a one DIV for each of the buttons and places them in a container
// DIV which is returned as our control element. We add the control to
// to the map container and return the element for the map class to
// position properly.
TextualZoomControl.prototype.initialize = function(map) {
  var container = document.createElement("div");

  var zoomInDiv = document.createElement("div");
  this.setButtonStyle_(zoomInDiv);
  container.appendChild(zoomInDiv);
  zoomInDiv.appendChild(document.createTextNode("Zoom In"));
  GEvent.addDomListener(zoomInDiv, "click", function() {
    map.zoomIn();
  });

  var zoomOutDiv = document.createElement("div");
  this.setButtonStyle_(zoomOutDiv);
  container.appendChild(zoomOutDiv);
  zoomOutDiv.appendChild(document.createTextNode("Zoom Out"));
  GEvent.addDomListener(zoomOutDiv, "click", function() {
    map.zoomOut();
  });

  map.getContainer().appendChild(container);
  return container;
}

// By default, the control will appear in the top left corner of the
// map with 7 pixels of padding.
TextualZoomControl.prototype.getDefaultPosition = function() {
  return new GControlPosition(G_ANCHOR_TOP_LEFT, new GSize(7, 7));
}

// Sets the proper CSS for the given button element.
TextualZoomControl.prototype.setButtonStyle_ = function(button) {
  button.style.textDecoration = "underline";
  button.style.color = "#0000cc";
  button.style.backgroundColor = "white";
  button.style.font = "small Arial";
  button.style.border = "1px solid black";
  button.style.padding = "2px";
  button.style.marginBottom = "3px";
```

```
  button.style.textAlign = "center";
  button.style.width = "6em";
  button.style.cursor = "pointer";
}

var map = new GMap2(document.getElementById("map"));
map.addControl(new TextualZoomControl());
map.setCenter(new GLatLng(37.441944, -122.141944), 13);
```

View example (customcontrol.html)

Custom Overlays

Version 2 of the Maps API introduces the ability to create custom map overlays, like the built-in `GMarker` and `GPolyline` overlays, by subclassing the built-in `GOverlay` class. In this example, we create a `Rectangle` overlay that outlines a geographic region on the map.

The `Rectangle` class defines the four required methods of the `GOverlay` interface: `initialize()`, in which we create the DOM elements representing our overlay; `remove()`, in which we remove the overlay elements from the map; `copy()`, which copies the overlay for use in another map instance; and `redraw()`, which positions and sizes the overlay on the map based on the current projection and zoom level. See the class reference for `GOverlay` for more information about the methods you can override when you create your custom overlays.

Every DOM element that makes up an overlay exists on a *map pane* that defines the z-order at which it will be drawn. For example, polylines are flat against the map, so they are drawn in the lowest `G_MAP_MAP_PANE`. Markers place their shadow elements in the `G_MAP_MARKER_SHADOW_PANE` and their foreground elements in the `G_MAP_MARKER_PANE`. Placing your overlay elements in the correct panes ensures that polylines are drawn below marker shadows and the info window is drawn above other overlays on the map. In this example, our overlay is flat against the map, so we add it to the lowest z-order pane `G_MAP_MAP_PANE`, just like `GPolyline`. See the class reference for a complete list of map panes.

```
// A Rectangle is a simple overlay that outlines a lat/lng bounds on the
// map. It has a border of the given weight and color and can optionally
// have a semi-transparent background color.
function Rectangle(bounds, opt_weight, opt_color) {
  this.bounds_ = bounds;
  this.weight_ = opt_weight || 2;
  this.color_ = opt_color || "#888888";
}
Rectangle.prototype = new GOverlay();

// Creates the DIV representing this rectangle.
Rectangle.prototype.initialize = function(map) {
  // Create the DIV representing our rectangle
  var div = document.createElement("div");
  div.style.border = this.weight_ + "px solid " + this.color_;
  div.style.position = "absolute";
```

```
  // Our rectangle is flat against the map, so we add our selves to the
  // MAP_PANE pane, which is at the same z-index as the map itself (i.e.,
  // below the marker shadows)
  map.getPane(G_MAP_MAP_PANE).appendChild(div);

  this.map_ = map;
  this.div_ = div;
}

// Remove the main DIV from the map pane
Rectangle.prototype.remove = function() {
  this.div_.parentNode.removeChild(this.div_);
}

// Copy our data to a new Rectangle
Rectangle.prototype.copy = function() {
  return new Rectangle(this.bounds_, this.weight_, this.color_,
                       this.backgroundColor_, this.opacity_);
}

// Redraw the rectangle based on the current projection and zoom level
Rectangle.prototype.redraw = function(force) {
  // We only need to redraw if the coordinate system has changed
  if (!force) return;

  // Calculate the DIV coordinates of two opposite corners of our bounds to
  // get the size and position of our rectangle
  var c1 = this.map_.fromLatLngToDivPixel(this.bounds_.getSouthWest());
  var c2 = this.map_.fromLatLngToDivPixel(this.bounds_.getNorthEast());

  // Now position our DIV based on the DIV coordinates of our bounds
  this.div_.style.width = Math.abs(c2.x - c1.x) + "px";
  this.div_.style.height = Math.abs(c2.y - c1.y) + "px";
  this.div_.style.left = (Math.min(c2.x, c1.x) - this.weight_) + "px";
  this.div_.style.top = (Math.min(c2.y, c1.y) - this.weight_) + "px";
}

var map = new GMap2(document.getElementById("map"));
map.addControl(new GSmallMapControl());
map.addControl(new GMapTypeControl());
map.setCenter(new GLatLng(37.4419, -122.1419), 13);

// Display a rectangle in the center of the map at about a quarter of
// the size of the main map
var bounds = map.getBounds();
var southWest = bounds.getSouthWest();
var northEast = bounds.getNorthEast();
var lngDelta = (northEast.lng() - southWest.lng()) / 4;
var latDelta = (northEast.lat() - southWest.lat()) / 4;
```

```
var rectBounds = new GLatLngBounds(
    new GLatLng(southWest.lat() + latDelta, southWest.lng() + lngDelta),
    new GLatLng(northEast.lat() - latDelta, northEast.lng() - lngDelta));
map.addOverlay(new Rectangle(rectBounds));
```

View example (customoverlay.html)

Geocoder Examples

You can access the Maps API geocoder directly by sending HTTP requests to Google, or you can use the `GClientGeocoder` object to send those requests from within the Javascript.

Geocoding using JavaScript

The geocoder can be accessed from within the JavaScript using the `GClientGeocoder` object. The `getLatLng` method can be used to convert an address into a `GLatLng`. Because geocoding involves sending a request to Google's servers, it can take some time. To avoid making your script wait, you need to pass in a function to call after the response comes back. In this example, we geocode an address, add a marker at that point, and open an info window displaying the address.

```
var map = new GMap2(document.getElementById("map"));
var geocoder = new GClientGeocoder();

function showAddress(address) {
  geocoder.getLatLng(
    address,
    function(point) {
      if (!point) {
        alert(address + " not found");
      } else {
        map.setCenter(point, 13);
        var marker = new GMarker(point);
        map.addOverlay(marker);
        marker.openInfoWindowHtml(address);
      }
    }
  );
}
```

View example (geocoder.html)

Extracting Structured Address Information

If you would like to access structured information about an address, `GClientGeocoder` also provides a `getLocations` method that returns a JSON object consisting of the following information.

- `Status`
  - `request` -- The request type. In this case, it is always `geocode`.
  - `code` -- A response code (similar to HTTP status codes) indicating whether the geocode request was successful or not. See the full list of status codes.
- `Placemark` -- Multiple placemarks may be returned if the geocoder finds multiple matches.
  - `address` -- A nicely formatted and properly capitalized version of the address.
  - `AddressDetails` -- The address formatted as xAL, or eXtensible Address Language, an international standard for address formatting.
    - `Accuracy` -- An attribute indicating how accurately we were able to geocode the given address. See a list of possible values.
  - `Point` -- A point in 3D space.
    - `coordinates` -- The longitude, latitude, and altitude of the address. In this case, the altitude is always set to 0.

Here we show the JSON object returned by the geocoder for the address of Google's headquarters.

```
{
  "name": "1600 Amphitheatre Parkway, Mountain View, CA, USA",
  "Status": {
    "code": 200,
    "request": "geocode"
  },
  "Placemark": [
    {
      "address": "1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA",
      "AddressDetails": {
        "Country": {
          "CountryNameCode": "US",
          "AdministrativeArea": {
            "AdministrativeAreaName": "CA",
            "SubAdministrativeArea": {
              "SubAdministrativeAreaName": "Santa Clara",
              "Locality": {
                "LocalityName": "Mountain View",
                "Thoroughfare": {
                  "ThoroughfareName": "1600 Amphitheatre Pkwy"
                },
                "PostalCode": {
                  "PostalCodeNumber": "94043"
                }
              }
            }
          }
```

```
        },
        "Accuracy": 8
      },
      Point: {
        coordinates: [-122.083739, 37.423021, 0]
      }
    }
  ]
}
```

In this example, we use the `getLocations` method to geocode addresses, and extract the nicely formatted version of the address and the two-letter country from the JSON and display it in the info window.

```
var map;
var geocoder;

function addAddressToMap(response) {
  map.clearOverlays();
  if (!response || response.Status.code != 200) {
    alert("\"" + address + "\" not found");
  } else {
    place = response.Placemark[0];
    point = new GLatLng(place.Point.coordinates[1],
                        place.Point.coordinates[0]);
    marker = new GMarker(point);
    map.addOverlay(marker);
    marker.openInfoWindowHtml(place.address + '<br>' +
    '<b>Country code:</b> ' + place.AddressDetails.Country.CountryNameCode);
  }
}
```

View example (geocoder2.html)

Caching Geocodes

`GClientGeocoder` is equipped with a client-side cache by default. The cache stores geocoder responses, so that if the same address is geocoded again, the response will be returned from the cache rather than from Google's geocoder. To turn off caching, pass `null` to the `setCache` method of `GClientGeocoder`. However, we recommend that you leave caching on because it improves performance. To change the cache being used by `GClientGeocoder`, use the `setCache` method and pass in the new cache. To empty the contents of the current cache, use the `reset` method on the geocoder or on the cache directly.

Developers are encouraged to build their own client-side caches. In this example, we construct a cache that contains pre-computed geocoder responses to six capital cities in countries covered by geocoding API. First, we build an array of geocode responses. Next, we create a custom cache that extends a standard GeocodeCache. Once the cache is defined, we call the `setCache` method. There is no

strict checking of objects stored in the cache, so you may store other information (such as population size) in the cache as well.

```
// Builds an array of geocode responses for the 6 capitals
var city = [
  {
    name: "Washington, DC",
    Status: {
      code: 200,
      request: "geocode"
    },
    Placemark: [
      {
        address: "Washington, DC, USA",
        population: "0.563M",
        AddressDetails: {
          Country: {
            CountryNameCode: "US",
            AdministrativeArea: {
              AdministrativeAreaName: "DC",
              Locality: {
                LocalityName: "Washington"
              }
            }
          },
          Accuracy: 4
        },
        Point: {
          coordinates: [-77.036667, 38.895000, 0]
        }
      }
    ]
  },
  ... // etc., and so on for other cities
];

  var map;
  var geocoder;

  // CapitalCitiesCache is a custom cache that extends the standard GeocodeCache.
  // We call apply(this) to invoke the parent's class constructor.
  function CapitalCitiesCache() {
    GGeocodeCache.apply(this);
  }

  // Assigns an instance of the parent class as a prototype of the
  // child class, to make sure that all methods defined on the parent
  // class can be directly invoked on the child class.
```

```
  CapitalCitiesCache.prototype = new GGeocodeCache();

  // Override the reset method to populate the empty cache with
  // information from our array of geocode responses for capitals.
  CapitalCitiesCache.prototype.reset = function() {
    GGeocodeCache.prototype.reset.call(this);
    for (var i in city) {
      this.put(city[i].name, city[i]);
    }
  }

  map = new GMap2(document.getElementById("map"));
  map.setCenter(new GLatLng(37.441944, -122.141944), 6);

  // Here we set the cache to use the UsCitiesCache custom cache.
  geocoder = new GClientGeocoder();
  geocoder.setCache(new CapitalCitiesCache());
```

View example (geocodercache.html)

HTTP Request

To access the Maps API geocoder directly using server-side scripting, send a request to http://maps.google.com/maps/geo? with the following parameters in the URI:

- `q` -- The address that you want to geocode.
- `key` -- Your API key.
- `output` -- The format in which the output should be generated. The options are `xml`, `kml`, `csv`, or `json`.

In this example, we request the geographic coordinates of Google's headquarters.

```
http://maps.google.com/maps/geo?q=1600+Amphitheatre+Parkway,+Mountain+View,
+CA&output=xml&key=abcdefg
```

If you specify `json` as the output, the response is formatted as a JSON object, as shown in the example above. If you specify `xml` or `kml`, the response is returned in XML or KML. The XML and KML outputs are identical except for the MIME types.

For example, this is the response that the geocoder returns for "1600 amphitheatre mtn view ca".

```
<kml>
  <Response>
```

```
    <name>1600 amphitheatre mtn view ca</name>
    <Status>
      <code>200</code>
      <request>geocode</request>
    </Status>
    <Placemark>
      <address>
        1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA
      </address>
      <AddressDetails Accuracy="8">
        <Country>
          <CountryNameCode>US</CountryNameCode>
          <AdministrativeArea>
            <AdministrativeAreaName>CA</AdministrativeAreaName>
           <SubAdministrativeArea>
             <SubAdministrativeAreaName>Santa Clara</SubAdministrativeAreaName>
             <Locality>
               <LocalityName>Mountain View</LocalityName>
               <Thoroughfare>
                 <ThoroughfareName>1600 Amphitheatre Pkwy</ThoroughfareName>
               </Thoroughfare>
               <PostalCode>
                 <PostalCodeNumber>94043</PostalCodeNumber>
               </PostalCode>
             </Locality>
           </SubAdministrativeArea>
         </AdministrativeArea>
       </Country>
     </AddressDetails>
     <Point>
       <coordinates>-122.083739,37.423021,0</coordinates>
     </Point>
   </Placemark>
  </Response>
</kml>
```

If you prefer a shorter response that is easier to parse and don't need special features like multiple results or pretty formatting, we also provide a `csv` output. A reply returned in the `csv` format consists of four numbers, separated by commas. The first number is the status code, the second is the accuracy, the third is the latitude, while the fourth one is the longitude. The following shows replies for three addresses, in increasing order of accuracy: "State St, Troy, NY", "2nd st & State St, Troy, NY" and "7 State St, Troy, NY"

```
200,6,42.730070,-73.690570
200,7,42.730210,-73.691800
200,8,42.730287,-73.692511
```

Marker Manager Examples

To use a marker manager, create a `GMarkerManager` object. In the simplest case, just pass a map to it.

```
var map = new GMap2(document.getElementById("map"));
var mgr = new GMarkerManager(map);
```

You may also specify a number of options to fine-tune the marker manager's performance. These options are passed via a `GMarkerManagerOptions` object, which contains the following fields:

- `maxZoom`: specifies the maximum zoom level monitored by this marker manager. The default value is the highest zoom level supported by Google maps.
- `borderPadding`: specifies the extra padding, in pixels, monitored by the manager outside the current viewport. This allows for markers just out of sight to be displayed on the map, improving panning over small ranges. The default value is 100.
- `trackMarkers`: specifies whether movement of movements of markers should be tracked by the marker manager. If you wish to have managed markers that change their positions through the `setPoint()` method, set this value to `true`. By default, this flag is set to `false`. Note that if you move markers with this value set to `false`, they will appear in both the original location and the new location(s).

The `GMarkerManagerOptions` object is an object literal, so you simply declare the object without a constructor:

```
var map = new GMap2(document.getElementById("map"));
var mgrOptions = { borderPadding: 50, maxZoom: 15, trackMarkers: true };
var mgr = new GMarkerManager(map, mgrOptions);
```

Once you create a manager, you will want to add markers to it. `GMarkerManager` supports adding single markers one at a time using the `addMarker()` method or a collection passed as an array using the `addMarkers()` method. Single markers added using `addMarker()` will appear immediately on the map provided that they fall within the current view and specified zoom level constraints.

Adding markers collectively using `addMarkers()` is recommended as it is more efficient. Markers added using the `addMarkers()` method will not appear on the map until you explicitly call the `GMarkerManager`'s `refresh()` method, which adds all markers within the current viewport and border padding region to the map. After this initial display, `GMarkerManager` takes care of all visual updates by monitoring the map's "moveend" events.

Zoom Level Example: Weather Map

The following example creates a mock weather map for Europe. At zoom level 3, 20 randomly distributed weather icons are displayed. At level 6, when all 200 cities with population over 300,000 are easily distinguished, an additional 200 markers are shown. Finally, at level 8, 1000 markers are shown. (Note: to simplify the example, markers will be added at random locations.)

```
function setupMap() {
  if (GBrowserIsCompatible()) {
    map = new GMap2(document.getElementById("map"));
    map.addControl(new GLargeMapControl());
    map.setCenter(new GLatLng(41, -98), 4);
    window.setTimeout(setupWeatherMarkers, 0);
  }
}

function getWeatherMarkers(n) {
  var batch = [];
  for (var i = 0; i < n; ++i) {
    batch.push(new GMarker(getRandomPoint(), { icon: getWeatherIcon() }));
  }
  return batch;
}

function setupWeatherMarkers() {
  mgr = new GMarkerManager(map);
  mgr.addMarkers(getWeatherMarkers(20), 3);
  mgr.addMarkers(getWeatherMarkers(200), 6);
  mgr.addMarkers(getWeatherMarkers(1000), 8);
  mgr.refresh();
}
```

View example (weather_map.html)

Clustering Example: Google Offices

The marker manager can also perform simple clustering of markers. While it does not do so automatically, you can achieve the desired effect by setting both the minimum and maximum zoom at which a given marker is shown. In this example we create a map of Google offices in North America. At the highest level we show flags of countries in which offices are located. For zoom levels 3 to 7 we show icons over population centers where one or more offices are located. Finally, at levels 8 or higher, we show individual markers for each office.

```
var officeLayer = [
  {
    "zoom": [0, 3],
    "places": [
      { "name": "US Offices", "icon": ["us", "flag-shadow"], "posn": [40, -97] },
      { "name": "Canadian Offices", "icon": ["ca", "flag-shadow"], "posn": [58, -101] }
    ]
  },
  ...
};
```

```
function setupOfficeMarkers() {
  var mgr = new GMarkerManager(map);
  for (var i in officeLayer) {
    var layer = officeLayer[i];
    var markers = [];
    for (var j in layer["places"]) {
      var place = layer["places"][j];
      var icon = getIcon(place["icon"]);
      var posn = new GLatLng(place["posn"][0], place["posn"][1]);
      markers.push(new GMarker(posn, { title: place["name"], icon: icon }));
    }
    mgr.addMarkers(markers, layer["zoom"][0], layer["zoom"][1]);
  }
  mgr.refresh();
}
```

View example (google_northamerica_offices.html)

Troubleshooting

If your code doesn't seem to be working, here are some approaches that might help you solve your problems:

- Make sure your API key is valid.
- Look for typos. Remember that JavaScript is a case-sensitive language.
- Use a JavaScript debugger. In Firefox, you can use the JavaScript console or the Venkman Debugger. In IE, you can use the Microsoft Script Debugger.
- Search the Maps API discussion group. If you can't find a post that answers your question, post your question to the group along with a link to a web page that demonstrates the problem.
- See Other Resources for third party developer resources.

Other resources

Here are some additional resources. Note that these sites are not owned or supported by Google.

- Mapki is a popular wiki with information about the Maps API, including an FAQ page.

API Overview

The API Overview walks through the central concepts of the Maps API. For a complete reference of methods and classes exported by the API, see the Maps API Class Reference.

The GMap2 class

An instance of `GMap2` represents a single map on the page. You can create as many instances of this class as you want (one for each map on the page). When you create a new map instance, you specify a named element in the page (usually a `div` element) to contain the map. Unless you specify a size explicitly, the map uses the size of the container to determine its size.

The `GMap2` class has methods to manipulate the map's center and zoom level and to add and remove overlays (such as `GMarker` and `GPolyline` instances). It also has a method to open an "info window," which is the popup window you see on Google Maps. See The Info Window for more information.

For more information about GMap2, see the GMap2 class reference.

Events

You can add dynamic elements to your application by using event listeners. An object exports a number of named events, and your application can "listen" to those events using the static methods `GEvent.addListener` and `GEvent.bind`. For example, this code snippet shows an alert every time the user clicks on the map:

```
var map = new GMap2(document.getElementById("map"));
map.setCenter(new GLatLng(37.4419, -122.1419), 13);
GEvent.addListener(map, "click", function() {
  alert("You clicked the map.");
});
```

`GEvent.addListener` takes a function as the third argument, which promotes the use of function closures for event handlers. If you want to bind an event to a method on a class instance, you can use `GEvent.bind`. In the following example, an application class instance binds map events to its own methods, modifying class state when events are triggered:

```
function MyApplication() {
  this.counter = 0;
  this.map = new GMap2(document.getElementById("map"));
  this.map.setCenter(new GLatLng(37.4419, -122.1419), 13);
  GEvent.bind(this.map, "click", this, this.onMapClick);
}

MyApplication.prototype.onMapClick = function() {
  this.counter++;
  alert("You have clicked the map " + this.counter + " " +
        (this.counter == 1 ?"time" : "times"));
}

var application = new MyApplication();
```

View example (bind.html)

The Info Window

Each map has a single "info window," which displays HTML content in a floating window above the map. The info window looks a little like a comic-book word balloon; it has a content area and a tapered stem, where the tip of the stem is at a specified point on the map. You can see the info window in action by clicking a marker in Google Maps.

You can't show more than one info window at a time for a given map, but you can move the info window and change its contents as needed.

The basic info window method is `openInfoWindow`, which takes a point and an HTML DOM element as arguments. The HTML DOM element is appended to the info window container, and the info window window tip is anchored to the given point.

The `openInfoWindowHtml` method is similar, but it takes an HTML string as its second argument rather than a DOM element.

To display an info window above an overlay like a marker, you can pass an optional third argument that specifies the pixel offset between the specified point and the tip of the info window. So, if your marker is 10 pixels tall, you might pass the pixel offset `GSize(0, -10)`.

The `GMarker` class exports `openInfoWindow` methods that handle the pixel offset issues for you based on the size and shape of the icon, so you generally don't have to worry about calculating icon offsets in your application.

See the `GMap2` and `GMarker` class references for more information.

Overlays

Overlays are objects on the map that are tied to latitude/longitude coordinates, so they move when you drag or zoom the map and when you switch map projections (such as when you switch from Map to Satellite mode).

The Maps API exports two types of overlays: markers, which are icons on the map, and polylines, which are lines made up of a series of points.

Markers and Icons

The `GMarker` constructor takes an icon and a point as arguments and exports a small set of events like "click". See the overlay.html example above for a simple example of creating markers.

The most difficult part of creating a marker is specifying the icon, which is complex because of the number of different images that make up a single icon in the Maps API.

Every icon has a foreground image and a shadow image. The shadow should be created at a 45 degree angle (upward and to the right) from the foreground image, and the bottom left corner of the shadow image should align with the bottom-left corner of the icon foreground image. The shadow should be a 24-bit PNG image with alpha transparency so that the edges of the shadow look correct on top of the map.

The `GIcon` class requires you specify the size of these images when you initialize the icon so the Maps API can create image elements of

the appropriate size. This is the minimum amount of code required to specify an icon (in this case, the icon used on Google Maps):

```
var icon = new GIcon();
icon.image = "http://www.google.com/mapfiles/marker.png";
icon.shadow = "http://www.google.com/mapfiles/shadow50.png";
icon.iconSize = new GSize(20, 34);
icon.shadowSize = new GSize(37, 34);
```

The `GIcon` class also has several other properties that you should set to get maximum browser compatibility and functionality from your icons. For example, the `imageMap` property specifies the shape of the non-transparent parts of the icon image. If you do not set this property in your icon, the entire icon image (including the transparent parts) will be clickable in Firefox/Mozilla. See the GIcon class reference for more information.

Polylines

The `GPolyline` constructor takes an array of points as an argument, and creates a series of line segments that connect those points in the given sequence. You can also specify the color, weight, and opacity of the line. The color should be in the hexadecimal numeric HTML style, e.g., use `#ff0000` instead of `red`. `GPolyline` does not understand named colors.

The following code snippet creates a 10-pixel-wide red polyline between two points:

```
var polyline = new GPolyline([
  new GLatLng(37.4419, -122.1419),
  new GLatLng(37.4519, -122.1519)
], "#ff0000", 10);
map.addOverlay(polyline);
```

In Internet Explorer, Google Maps uses VML to draw polylines (see XHTML and VML for more information). In all other browsers, we request an image of the line from Google servers and overlay that image on the map, refreshing the image as necessary as the map is zoomed and dragged around.

Controls

To add controls like the zoom bar to your map, use the `addControl` method. The Maps API comes with a handful of built-in controls you can use in your maps:

- `GLargeMapControl` - a large pan/zoom control used on Google Maps. Appears in the top left corner of the map.
- `GSmallMapControl` - a smaller pan/zoom control used on Google Maps. Appears in the top left corner of the map.
- `GSmallZoomControl` - a small zoom control (no panning controls) used in the small map blowup windows used to display driving directions steps on Google Maps.

- `GScaleControl` - a map scale
- `GMapTypeControl` - buttons that let the user toggle between map types (such as Map and Satellite)
- `GOverviewMapControl` New! - a collapsible overview map in the corner of the screen

For example, to add the panning/zooming control you see on Google Maps to your map, you would include the following line in your map initialization:

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

`addControl` takes an optional second `GControlPosition` parameter that lets you specify the position of the control on your map. If this argument is excluded, the Maps API uses the default position specified by the control. This example adds the map type control to the bottom right corner of the map with 10 pixels of padding:

```
map.addControl(new GMapTypeControl(),
               new GControlPosition(G_ANCHOR_BOTTOM_RIGHT, new GSize(10, 10)));
```

See the `GControlPosition` class reference for more information.

XML and RPC

The Google Maps API exports a factory method for creating `XmlHttpRequest` objects that work in recent versions of IE, Firefox, and Safari. The following example downloads a file called `myfile.txt` and displays its contents in a JavaScript `alert`:

```
var request = GXmlHttp.create();
request.open("GET", "myfile.txt", true);
request.onreadystatechange = function() {
  if (request.readyState == 4) {
    alert(request.responseText);
  }
}
request.send(null);
```

The API also exports a simpler method for typical HTTP GET requests called `GDownloadUrl` that eliminates the need for `XmlHttpRequest` `readyState` checking. The example above could be rewritten using `GDownloadUrl` like this:

```
GDownloadUrl("myfile.txt", function(data, responseCode) {
```

```
  alert(data);
});
```

You can parse an XML document with the static method `GXml.parse`, which takes a string of XML as its only argument. This method is compatible with most modern browsers, but it throws an exception if the browser does not support XML parsing natively.

See the `GXmlHttp` and `GXml` class references for more information.

Reducing Browser Memory Leaks

The Google Maps API encourages the use of function closures, and the API event handling system `GEvent` attaches events to DOM nodes in such a way that almost inevitably causes some browsers to leak memory, particularly Internet Explorer. Version 2 of the Maps API introduces a new method, `GUnload()`, that will remove most of the circular references that cause these leaks. You should call `GUnload()` in the `unload` event of your page to reduce the potential that your application leaks memory:

```
<body onunload="GUnload()">
```

Using this function has virtually eliminated Internet Explorer memory leaks in Google Maps, though you should test for memory leaks on your own site using tools like Drip if you are noticing memory consumption problems.