Quick answers to common problems

OpenLayers Cookbook

60 recipes to create GIS web applications with the open source JavaScript library

Antonio Santiago Perez

[PACKT] PUBLISHING open source community experience distilled

OpenLayers Cookbook

60 recipes to create GIS web applications with the open source JavaScript library

Antonio Santiago Perez

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

OpenLayers Cookbook

First published: August 2012

Production Reference: 1170812

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84951-784-3

www.packtpub.com

Cover Image by Avishek Roy (roy007avishek88@gmail.com)

Credits

Author

Antonio Santiago Perez

Reviewers

David Burgoon

Mohammad Motamedi

Jorge Sanz

Srinivas Shanmugam

Davor Zelic

Richard Zijlstra

Acquisition Editor

Usha Iyer

Lead Technical Editor

Kedar Bhat

Technical Editors

Madhuri Das

Joyslita Dsouza

Prasad Dalvi

Prashant Salvi

Project Coordinator

Joel Goveya

Proofreaders

Mario Cecere

Linda Morris

Copy Editors

Laxmi Subramanian

Alfida Paiva

Indexer

Rekha Nair

Production Coordinator

Shantanu Zagade

Cover Work

Shantanu Zagade

About the Author

Antonio Santiago Perez is a Computer Science Engineer with more than 10 years of experience in designing and implementing systems.

Since the beginning of his professional life, his experience has always been related to the world of meteorology, working for different companies as an employee and a freelancer. He is experienced in development of systems to collect, store, transform, analyze, and visualize data, and actively interested in any GIS-related technology, with preference for data visualization.

Having a restless mind and being experienced mainly in Java ecosystem, he also has been working actively with many related web technologies, always looking to improve the client side of web applications.

As a firm believer in Software Engineering practices, he is an enthusiast of Agile methodologies involving customers as a main key for the project's success.

First, I would like to dedicate this book to my wife, for understanding my passion for programming and the world of computers.

Second, I would like to dedicate this book to all the restless people who make great open source projects possible, such as OpenLayers, for the simple pleasure to create something one step better.

About the Reviewers

David Burgoon is a Software Developer with over 15 years of experience and lives and works in New York City. He specializes in designing and developing web mapping applications and geographic information systems (GIS).

David currently works at the Center for Urban Research at the City University of New York, where he develops web applications and data visualizations related to issues concerning New York City and other large metropolitan areas. Some of his recent projects using OpenLayers include visualizing demographic changes between the 2000 and 2010 census and the effects of voter redistricting on local populations.

Mohammad Motamedi is a Software Developer/Analyst, specializing in GIS (Geographic Information System) for more than 10 years. He currently works as a GIS Systems Analyst in the energy sector in Alberta, Canada.

Jorge Sanz is a GIS Consultant from Valencia, Spain. Formerly a Surveying and Cartography Engineer, he has been working on Geographical Information Systems for the last eight years. After working as a researcher at the Polytechnic University of Valencia, he joined Prodevelop, a software development company. There he started as a Geospatial Developer, working on web and desktop GIS projects. Over the years he has been working mainly to help his colleagues with his geospatial experience on analysis, development, consultancy, documenting, training, and participating in many different conferences and workshops.

Apart from working on Prodevelop projects, he has also been part of the gvSIG team (a free GIS desktop software project) since 2006. Nowadays he works at gvSIG project as a technical collaborations manager, and he is also a charter member of the Open Source Geospatial Foundation (OSGeo) as well as a member of the OSGeo Spanish Language Local Chapter and other local local groups where he devotes some time and energy.

In 2005, during his time as a researcher, he participated in the authoring of a UMN Mapserver book, one of the first publications in Spanish on this excellent software.

I want to thank my colleagues at Prodevelop. I've learned from them more than from any book or published resource; they give me inspiration and energy to work on innovative projects, always looking for excellence and perfection.

Srinivas Shanmugam has more than 10 years of software application development and architect experience using open source technologies in various domains. He specializes in analysis, design, implementation, development, and data migration. He has expertise in implementing User Interface components, Map components using OpenLayers, Web 2.0, OOPS/Design pattern methodologies, open source custom framework development, MVC (Model View Controller), and other open source software development methodologies.

I would like to thank my past company Logica, my PM Rajesh Roy and Finland counterparts for giving me the opportunity to work in OpenLayers, which made me review this book.

Davor Zelic is an IT professional who has been working in the IT industry for 12 years. During his career, Davor has gained expertise in working with various Geographic Information Systems. He originally worked with Intergraph Technologies where he earned the certificate of Intergraph Certified Developer for GeoMedia Desktop Solutions. Later, his focus moved to Open Source GIS technology where he gained significant experience working with server-side technology such as Geoserver and client-side technology such as OpenLayers.

From the beginning of his career, Davor has worked constantly with Oracle technology as an SQL, PLSQL, Spatial, Forms, and Reports expert and earned the certificate of Oracle Certified Professional issued by Oracle Corporation.

Davor holds a Master's degree in Electrical Engineering from the University of Zagreb. Currently he works at a small Croatian IT company, TEB Informatika, as a Chief Technology Officer.

Richard Zijlstra is a Civil Engineer. He has used his engineering degree in the Netherlands on Water Management, Infrastructure Planning, and Geographical Information Management on all of the environmental and social human aspects. He collaborates on system architecture, requirement management, and development of cloud-based Geographical Information Technology.

At the moment (2012) Richard Zijlstra is developing an OpenLayers application for all of the Governmental institutes in the Netherlands. This application will be an interactive alternative to Google Maps. Also, interactivity in social media will be possible. His future vision is based on Geographical Intelligence in all contexts in life and on earth.

Richard Zijlstra is the owner of the company Geoneer. Geoneer is a pioneer in geography and information technology. From this vision and point of view, Geoneer will help and collaborate in all aspects of geographical information technology worldwide. You can find Geoneer at `@geoneer` on Twitter.

Richard Zijlstra has written a lot of documents, system architectures, and on the usage of Geographical Information Technology, which you can find on the Web.

I want to thank my parents for my healthy brain and childhood environment in the Frisian country side. Also, I thank the people from the town of Groningen who inspired me a lot for doing my thing (they know what I mean). Also I'm very thankful to those people who know how I think, what I do, and what I wish to do in the future. My greatest thanks goes out to my son Alessio Mori Zijlstra, my greatest inspiration in life!

---

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

We live in the century of information and a lot of this information is susceptible to being represented geographically. This is probably the main feature that has made Geographic Information System (GIS) become one of the most important aspects of many companies. GIS-related technologies are a growing industry and, because of this, GIS has become a desired skill for many professionals.

The universality of the Web and the improvement on the browsers' performance has made it possible for the Web to become a main part of the current GIS and has subtracted the importance of desktop applications because of its capabilities: the browsers allow us to show data visualizations to the masses, create online data editors, and so on.

Nowadays, OpenLayers is the most complete and powerful open source JavaScript library to create any kind of web mapping application. In addition to offering a great set of components, such as maps, layers, or controls, OpenLayers offers access to a great number of data sources using many different data formats, implements many standards from *Open Geospatial Consortium* (OGC, `http://www.opengeospatial.org`), and is compatible with almost all browsers.

What this book covers

Chapter 1, *Web Mapping Basics*, introduces OpenLayers to the reader and describes the basics of how to create a map, how to manage the layers stack, how to work with controls, and how to add OpenLayers within your project.

Chapter 2, *Adding Raster Layers*, is centered on working and understanding the main raster layers. OpenLayers offers the opportunity to work with main service providers, such as Google or Bing, plus integrating with open source ones, such as OpenStreetMap, or working with WMS servers.

Chapter 3, *Working with Vector Layers*, explores the power of vector layers and explains how we can load data from different sources using different formats and how we can create features, markers, and popups.

Chapter 4, *Working with Events*, describes the importance of events and how we can react when they are triggered by the OpenLayers components, such as map or layers, each time a layer is added to the map, a layer is loaded, a feature is added, and so on.

Chapter 5, *Adding Controls*, explains how to manage controls and describes the most commonly used and important controls the OpenLayers offers to the users: adding or editing features, measuring distances, getting information about features, and so on.

Chapter 6, *Theming*, describes how OpenLayers is designed to control the appearance of its components. This chapter shows how we can change the position or the controls' look and introduces the basics of creating a complete new theme.

Chapter 7, *Styling Features*, is completely oriented to show how we can control the features' appearance: the different ways we have to style features, the concept of renderers, styling depending on feature attributes, and so on.

Chapter 8, *Beyond the Basics*, explores some advanced topics of OpenLayers: work with projections, request remote data, create new controls, and so on. It collects some recipes that show the possibilities OpenLayers offers to developers.

What you need for this book

The fact that OpenLayers is a JavaScript library, which must be integrated within HTML pages, implies that the user must be familiarized with these technologies.

To run the recipes you need to place the code bundle within a HTTP server that serves the pages. All library dependencies required by the recipes code, such as OpenLayers or Dojo toolkit (`http://dojotoolkit.org`), are included in the bundle itself.

Some recipes request data from PHP scripts included in the source bundle. The function of these scripts is to generate some random data to later integrate within the map. To run these recipes properly the reader needs a HTTP server with PHP support.

Solutions such as XAMPP (`http://www.apachefriends.org/en/xampp.html`) or Bitnami Stacks (`http://bitnami.org/stack/lampstack`) are an easy way to install the required stack.

For better user experience, we have created a main application that allows the desired recipe to run and show its source code. Supposing the reader has installed and configured a local web server, and the bundle code is copied within the HTTP server root content folder in the `openlayers-cookbook` folder, the user can run the main application by accessing the `http://localhost/openlayers-cookbook/index.html` URL in the browser.

We have made use of the Dojo toolkit in many of the recipes because it allows us to create rich components such as sliders or toggle buttons. Dojo's JavaScript library and CSS files are included in the `index.html` file, so the HTML recipe files do not have to include them. If the reader plans to run the recipes as standalone web pages, he/she will need to include Dojo's JavaScript library and CSS files in the recipe file, otherwise the recipe will not work properly.

Who this book is for

This book is ideal for GIS-related professionals who need to create web-mapping applications.

From basic to advanced topics, the recipes of this book cover in a direct way the most common issues a user can find in his/her day-to-day job.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "The `topLayer()` and `bottomLayer()` actions are similar too, they move the specified layer to the top or bottom of the stack."

A block of code is set as follows:

```
<style>
    html, body {
        width: 100%;
        height: 100%;
        margin: 0;
        padding: 0;
    }
</style>
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "The layer opacity is set to **50%** in the following screenshot."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

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

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Web Mapping Basics

In this chapter we cover:

- Creating a simple full screen map
- Different ways to include OpenLayers
- Understanding base and non-base layers
- Avoiding the need of a base layer
- Playing with the map's options
- Managing map's stack layers
- Managing map's controls
- Moving around the map view
- Restricting the map extent

Introduction

Every history has a beginning, in the same way every recipe starts with the initial condiments.

This chapter shows us the basics and more important things that we need to know when we start creating our first web mapping applications with OpenLayers.

As we will see in this chapter and the following chapters, OpenLayers is a big and complex framework but, at the same time it is also very powerful and flexible.

In contrast to other libraries, such as the nice but much more simple Leaflet project (`http://leaflet.cloudmade.com`) library, OpenLayers tries to implement all the required things a developer could need to create a web GIS application. That is, not only GIS related concepts such as map, layer, or standard formats but also manipulation of document elements or helper functions to make asynchronous requests.

Trivializing, we have described a big picture of the framework in the next paragraph.

The main concept in OpenLayers is the map. It represents the view where information is rendered. The map can contain any number of layers, which can be the raster or vector layer. On its way, each layer has a data source that serves data with its own format: a PNG image, a KML file, and so on. In addition, the map can contain controls, which help to interact with the map and its contents: pan, zoom, feature selection, and so on.

Let's get started with learning OpenLayers by examples.

Creating a simple full screen map

When you work in mapping applications, the first and important task is the creation of the map itself. The map is the core of your application and it is where you will add and visualize data.

This recipe will guide you through the process of creating our first and very simple web map application.

It is supposed that a web server is configured and ready. Remember our recipes are nothing more than HTML, CSS, and JavaScript code and because of this we need a web server that serves them to be interpreted on the browser's side.

Getting ready

Programming with OpenLayers is mainly related to writing HTML code and, of course, JavaScript code. So, we simply need a text editor to start coding our recipes.

There exist plenty of great text editors, many of them with web programming capabilities. Because we are going to start exploring an open source project such as OpenLayers we will refer to a couple of great open projects.

For Windows users, Notepad++ (`http://notepad-plus-plus.org`) is a great alternative to the default text editor. It is simple and quick, offers syntax highlighting, and addition of plugins to extend capabilities.

On the other hand, instead of text editors you can find complete development frameworks with support for web development, not only with syntax highlighting but with autocomplete, code navigation, and many more.

In this group, two projects are the stars within the open source projects universe: Eclipse (`http://www.eclipse.org`) and NetBeans (`http://netbeans.org`). Both are Java-based projects and run on any platform.

You can find the source code at `recipe/ch01/ch01_simple_map_book.html` file.

How to do it...

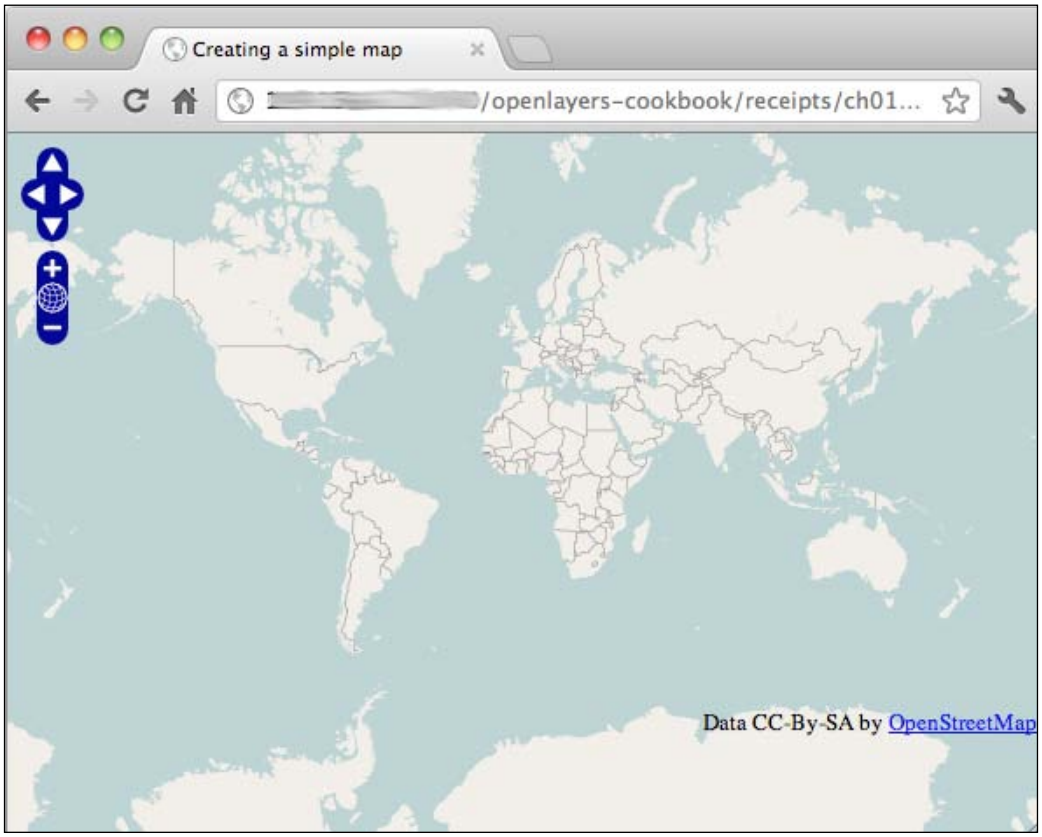
1. Let's start by creating a new empty `index.html` file and inserting the following block of code in it. We will explain it step-by-step in the next section:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Creating a simple map</title>
        <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8">

        <!-- Include OpenLayers library -->
        <script type="text/javascript"
            src="http://openlayers.org/api/2.11/
            OpenLayers.js"></script>
        <style>
            html, body {
                width: 100%;
                height: 100%;
                margin: 0;
                padding: 0;
            }
        </style>

        <!-- The magic comes here -->
        <script type="text/javascript">
            function init() {
                // Create the map using the specified
                // DOM element
                var map = new OpenLayers.Map("rcp1_map");

                // Create an OpenStreeMap raster layer
                // and add to the map
                var osm = new OpenLayers.Layer.OSM();
                map.addLayer(osm);

                // Set view to zoom maximum map extent
                map.zoomToMaxExtent();
            }
        </script>
    </head>
    <body onload="init()">
        <div id="rcp1_map" style="width: 100%;
            height: 100%;"></div>
    </body>
</html>
```

Downloading the example code

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

2. Open the file in your browser and see the result. You will see a whole screen map with some controls on the top-left corner, as shown in the following screenshot:



How it works...

Let us explain the mystery step-by-step. First, we created a HTML5 document, see the `doctype` declaration code `<!DOCTYPE html>`.

In the `head` section, we have included a reference to the OpenLayers library using a `script` element, as follows:

```
<script type="text/javascript"
src="http://openlayers.org/api/2.11/OpenLayers.js"></script>
```

We have added a `style` element to force the document to occupy the whole page, as follows:

```
<style>
    html, body {
        width: 100%;
        height: 100%;
        margin: 0;
        padding: 0;
    }
</style>
```

After the `style` element comes the `script` element with some JavaScript code, but we will explain it at the end.

After the `head` section starts the `body` section. Our `body` has an `onload` event associated to it. This means, once the whole content of the `body` section is completely loaded by the browser, the `init()` function will be called:

```
<body onload="init()">
```

Finally, within the body we have put a `div` element with the identifier `rcp1_map`, which will be used by OpenLayers to render the map.

Again, we force the element to fill the entire parent's space:

```
<div id="rcp1_map" style="width: 100%; height: 100%;"></div>
```

A word about styles...

Setting the `div` element `width/height` to `100%` means it will fill 100 percent of the parent's space. In this case, the parent is the `body` element, which is also set to fill 100 percent of the page space. More and better information about CSS can be found at `http://www.w3schools.com/css`.

Now, let's take a look at the `script` element in the `head` section.

As we have mentioned previously, using the `onload` event we ensure the `init` function is executed once the entire `body` elements are loaded by the browser, which means we can access `<div id="rcp1_map" ...>` without any problem.

First we created an `OpenLayers.Map` object that will be rendered in the previously mentioned `div` element. This is achieved by passing the `DOM` element identifier in the constructor:

```
// Create the map using the specified DOM element
var map = new OpenLayers.Map("rcp1_map");
```

Next, we created a new raster layer that will show imagery from the `OpenStreetMaps` project:

```
// Create an OpenStreetMap raster layer and add to the map
var osm = new OpenLayers.Layer.OSM();
```

Once created we add it to the map:

```
map.addLayer(osm);
```

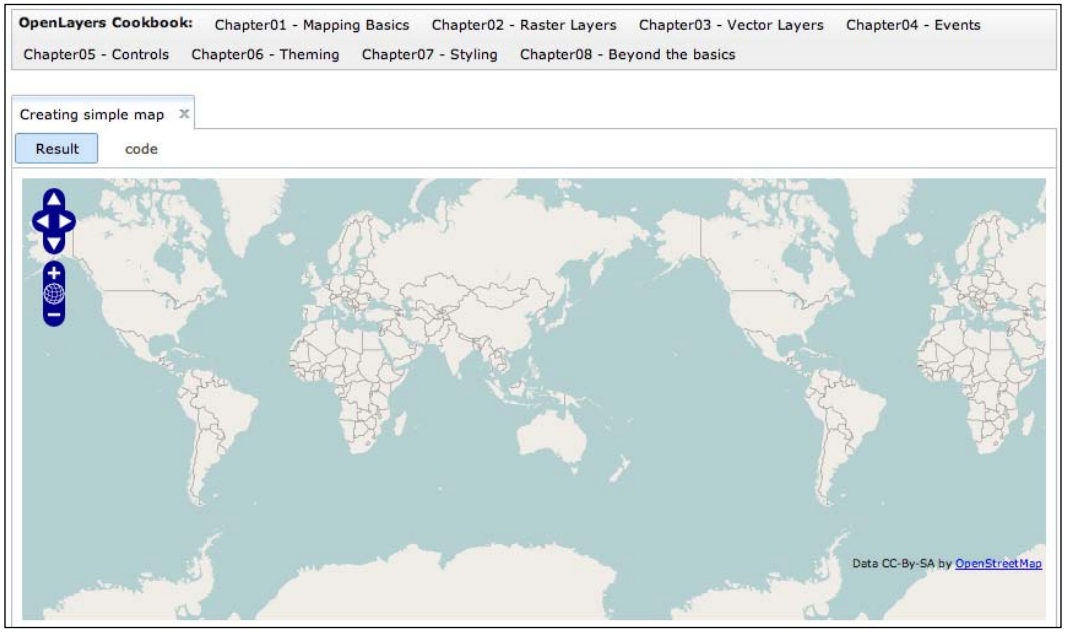
Finally, set the `map` view to the maximum valid extent:

```
// Set view to zoom maximum map extent
map.zoomToMaxExtent();
```

There's more...

Remember there is no one way to do things.

The recipes in this book have not been coded as standalone applications. Instead, to improve the user experience, we have created a rich application that allows you to choose and run the desired recipe, with the possibility to see the source code.



So the way to code the recipes in the book is slightly different, because they must be integrated with the application's design. For example, they do not require including the OpenLayers libraries because this is included in another place of the main application.

In addition, the way presented in the *How to do it...* section is more oriented toward standalone applications.

If you are looking at the source code of this recipe, located at `recipes/ch01/ch01_simple_map.html`, we will see a slightly different code:

```
<!-- Map DOM element -->
<div id="ch1_simple_map" style="width: 100%; height: 95%;"></div>

<!-- The magic comes here -->
<script type="text/javascript">

    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch1_simple_map");

    // Create an OpenStreeMap raster layer and add to the map
    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);

    // Set view to zoom maximum map extent
    map.zoomToMaxExtent();
</script>
```

As we can see, it contains the main parts described in the previous sections. We have a `div` element to hold the map instance and a `script` element with all the required JavaScript code.

To create the rich application, we have to use the Dojo Toolkit framework (`http://dojotoolkit.org`), which offers almost any kind of required feature: access and modification of the document object structure, event handling, internationalization, and so on. But the main reason we have chosen it is because, in addition it offers a great set of homogeneous widgets (tabs, buttons, lists, and so on) to create applications with a great look and feel.

It is beyond the scope of this book to teach Dojo but its use is so easy and specific that it will not disturb the objective of this recipe, which is to teach OpenLayers.

See also

- The *Different ways to include OpenLayers* recipe
- The *Understanding base and non-base layers* recipe

Different ways to include OpenLayers

There are different ways we can include OpenLayers in our projects depending on the environment we are working in, that is development or production.

The environment refers to the server tier required for a stage in our process. In this way, the development environment is related to the development process, where programmers are working day to day, testing and checking.

Production environment refers to the final stage of the projects. It must run on stable servers and without dependency problems.

As we will see shortly, we can summarize the solutions to include OpenLayers JavaScript code in two groups, those with code hosted on a remote server or those with code hosted on our own server.

Let's start and see the pros and cons of each solution.

Getting ready

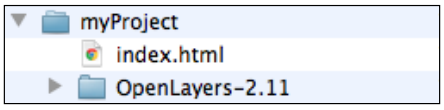
Create a folder called `myProject` that will contain all our project files and library dependencies. Then create an `index.html` file and continue with the code given in the *Creating a simple full screen map* recipe.

It is supposed that the project folder resides within a folder accessible by the web server folder, so it can serve its content.

Now download OpenLayers code from the project's web page at `http://www.openlayers.org`.

At the time of writing this book, OpenLayers version 2.11 is the stable release, which can be found at `http://openlayers.org/download/OpenLayers-2.11.tar.gz`.

Save the bundle in the `myProject` folder and uncompress it. We need to have a folder structure similar to the following screenshot:



How to do it...

We have three ways to include the OpenLayers library in our code:

- `<script type="text/javascript" src="http://openlayers.org/api/2.11/OpenLayers.js"></script>`
- `<script type="text/javascript" src="../js/OpenLayers-2.11/OpenLayers.js"></script>`
- `<script type="text/javascript" src="../js/OpenLayers-2.11/lib/OpenLayers.js"></script>`

How it works...

The first option includes an all-in-one compressed file hosted at the OpenLayers project server. It is simple to use but you cannot work locally in offline mode:

```
<script type="text/javascript"
src="http://openlayers.org/api/2.11/OpenLayers.js"></script>
```

The size of the compressed all-in-one file `OpenLayers.js` is nearly 1 MB, so in a production environment with lots of requests it is probably better to host this file in a Content Delivery Network or CDN (`http://en.wikipedia.org/wiki/Content_delivery_network`).

The second option is very similar to the first one, but the all-in-one compressed file is attached to the project. This option is suitable for cases in which you need OpenLayers to be in your own server with the code of your application.

```
<script type="text/javascript" src="../js/
OpenLayers-2.11/OpenLayers.js"></script>
```

Finally, the third option includes the uncompressed code of the OpenLayers library, which in fact includes many other files required by layers, controls, and so on.

```
<script type="text/javascript" src="../js/
OpenLayers-2.11/lib/OpenLayers.js"></script>
```

This option is mainly used by programmers who want to extend OpenLayers and need to debug the code.

I encourage you to work in this mode. Use some tool such as Firebug for Firefox web browser or the Chrome browser console and put breakpoints on OpenLayers classes to better understand what is happening.

It is worth saying when using this method lots of files are loaded from the server, one per class, which means many more server requests are made.

The most notable impact of this is that the page load time is much longer than with the previous options.

There's more...

If you choose to download OpenLayers and include it within your project, you don't need to put the whole uncompressed bundle. As you can see, it contains lots of files and folders: source code, building scripts, test code, and other tools, but only a few are really required.

In this case, the only things you need to attach are:

- The all-in-one `OpenLayers.js` file
- The `theme` and `img` folders

See also

- The *Understanding base and non-base layers* recipe
- The *Creating a simple full screen map* recipe

Understanding base and non-base layers

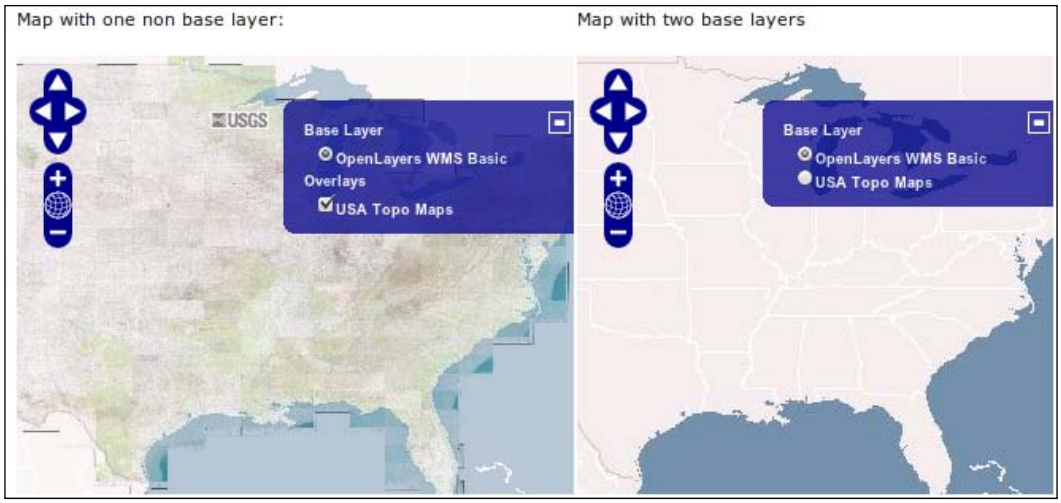
One of the first things you need to have clear when working with OpenLayers is the base layer concept.

A **base layer** is a special kind of layer, which is always visible and determines some map properties such as projection and zoom levels.

A map can have more than one base layer but only one of them can be active at a time.

In addition, if you add more than one flagged base layer to the map, the first base layer added will be used as the map's active base layer.

This recipe will show you how to add layers to the map flagging them to be base layers. We are going to create a page with two maps side-by-side and every map will have a layer switcher control that allows you to control the map layers.



Getting ready

We assume you have created an `index.html` file and included the OpenLayers library as we have seen in the *Different ways to include OpenLayers* recipe.

How to do it...

1. Start by creating the necessary HTML code to have both our maps side-by-side:

```
<table style="width: 100%; height: 100%;">
    <tr>
        <td>
            <p>Map with one non base layer:</p>
            <div id="ch01_base_nonbase_map_a"
                style="width: 100%; height: 500px;">
            </div>
        </td>
        <td>
            <p>Map with two base layers</p>
            <div id="ch01_base_nonbase_map_b"
                style="width: 100%; height: 500px;">
            </div>
        </td>
    </tr>
</table>
```

2. After this, add a `script` element (`<script type="text/javascript"></script>`) with the necessary code to initialize every map. The map on the left will contain two layers, one base layer and one non-base layer:

```
//
// Initialize left map
//

// Create the map using the specified DOM element
var map_a = new OpenLayers.Map("ch01_base_nonbase_map_a");
// Add a WMS layer
var wms = new OpenLayers.Layer.WMS("OpenLayers WMS Basic",
"http://vmap0.tiles.osgeo.org/wms/vmap0",
{
    layers: 'basic'
});
map_a.addLayer(wms);
// Add a WMS layer
var topo = new OpenLayers.Layer.WMS("USA Topo Maps",
    "http://terraservice.net/ogcmap.ashx",
{
    layers: "DRG"
},
{
    opacity: 0.5,
    isBaseLayer: false
});
map_a.addLayer(topo);
// Add LayerSwitcher control
map_a.addControl(new OpenLayers.Control.LayerSwitcher());

// Set view to zoom maximum map extent
// NOTE: This will fail if there is no base layer defined
map_a.setCenter(new OpenLayers.LonLat(-100, 40), 5);
```

3. The map on the right will contain two base layers:

```
//
// Initialize right map
//

// Create the map using the specified DOM element
var map_b = new OpenLayers.Map("ch01_base_nonbase_map_b");
// Add a WMS layer
```

```
var wms = new OpenLayers.Layer.WMS("OpenLayers WMS Basic",
"http://vmap0.tiles.osgeo.org/wms/vmap0",
{
    layers: 'basic'
});
map_b.addLayer(wms);
// Add a WMS layer
var topo = new OpenLayers.Layer.WMS("USA Topo Maps",
    "http://terraservice.net/ogcmap.ashx",
{
    layers: "DRG"
});
map_b.addLayer(topo);
// Add LayerSwitcher control
map_b.addControl(new OpenLayers.Control.LayerSwitcher());

// Set view to zoom maximum map extent
// NOTE: This will fail if there is no base layer defined
map_b.setCenter(new OpenLayers.LonLat(-100, 40), 5);
```

How it works...

Let's take a look at the explanation for the map on the left. The first thing we have done is the creation of an `OpenLayers.Map` instance that will be rendered in the `div` element prepared for it, on the left side:

```
var map_a = new OpenLayers.Map("ch01_base_nonbase_map_a");
```

Next, we have created two layers and added them to the map. The magic to make the second layer a non-base layer comes with the properties specified in the constructor:

```
var topo = new OpenLayers.Layer.WMS("USA Topo Maps", "http://
terraservice.net/ogcmap.ashx",
{
    layers: "DRG"
},
{
    opacity: 0.5,
    isBaseLayer: false
});
```

In OpenLayers, all layer classes are inherited from the base class `OpenLayers.Layer`. This class defines some properties common for all layers, such as `opacity` or `isBaseLayer`.

The Boolean `isBaseLayer` property is used by the map to know if a layer must act as a base or non-base layer.

Non-base layers are also called **overlays**.

As you can imagine, the `opacity` property is a float value ranging from 0.0 to 1.0 and specifies the opacity of the layer. We have set it to `50%` of the opacity to allow view through the overlay layer, that is, to be able to see the base layer.

For the right-hand map, we have added two layers without any specific property. This, by default, makes the WMS layer a base layer.

If you expand the layer switcher control, you will see that on the left map you can show/hide the overlay layer but you can't hide the base layer. In contrast, in the right map, both are base layers and they are mutually exclusive, which means only one can be active at a time.

There's more...

When you play with the layer switcher control, an internal call is made to the map's `setBaseLayer(newBaseLayer)` method. This method is responsible for changing the active base layer used by the map.

In addition to the specify properties at construction time, you can also use the setter methods `setOpacity(opacity)` and `setIsBaseLayer(isBaseLayer)` to change the values at runtime.

See also

- The *Avoiding the need of a base layer* recipe
- The *Managing map's stack layers* recipe

Avoiding the need of a base layer

There can be situations where you don't want a base layer and only want a bunch of layers to work on.

Imagine an online GIS editor where users can add and remove layers but they are not obligated to have an always visible one.

This recipe shows how we can easily avoid the requirement of setting a base layer within the map.

How to do it...

1. As always, create a DOM element to render the map:

```
<div id="ch1_avoid_baselayer" style="width: 100%;
height: 100%;"></div>
```

2. Now create a new `OpenLayers.Map` instance and set the `allOverlays` property to `true`:

```
// Create the map using the specified DOM element
var map = new OpenLayers.Map("ch1_avoid_baselayer", {
    allOverlays: true
});
```

3. Add two layers to see a result. Also add the layer switcher control:

```
// Add a WMS layer
var wms = new OpenLayers.Layer.WMS("OpenLayers WMS Basic",
          "http://vmap0.tiles.osgeo.org/wms/vmap0",
{
    layers: 'basic'
});
map.addLayer(wms);
// Add a WMS layer
var topo = new OpenLayers.Layer.WMS("USA Topo Maps",
           "http://terraservice.net/ogcmap.ashx",
{
    layers: "DRG"
},
{
```

```
    opacity: 0.5
});
map.addLayer(topo);
// Add LayerSwitcher control
map.addControl(new OpenLayers.Control.LayerSwitcher());
```

4. Center the map view to some nice place:

```
// Set view to zoom maximum map extent
// NOTE: This will fail if there is no base layer defined
map.setCenter(new OpenLayers.LonLat(-100, 40), 5);
```

How it works...

When the map's property `allOverlays` is set to `true`, the map ignores the `isBaseLayer` property of the layers.

If you expand the layer switcher control, you will see that it contains two overlay layers, no base layer, which you can show or hide and, if desired, leave a blank map without layers.

In addition, in this recipe we have used the `map.setCenter()` method, which needs a position, an `OpenLayers.LonLat` instance, and a zoom level to work.

There's more...

When working in the `allOverlays` mode, the lowest layer will act as base layer, although all the layers will be flagged as `isBaseLayer` is set to `false`.

See also

- The *Understanding base and non-base layers* recipe
- The *Moving around the map view* recipe
- The *Restricting the map extent* recipe

Playing with the map's options

When you create a map to visualize data, there are some important things you need to take into account: projection to use, available zoom levels, the default tile size to be used by the layer requests, and so on.

Most of these important things are enclosed in the so-called map properties and, if you work in the `allOverlays` mode, you need to take them specially into account.

This recipe shows you how to set some of the most common map properties.

Getting ready

Before you continue, it is important to note that instances of the `OpenLayers.Map` class can be created in three ways:

- Indicating the identifier of the DOM element where the map will be rendered:
  ```
  var map = new OpenLayers.Map("map_id");
  ```
- Indicating the identifier of the DOM element and also indicating a set of options:
  ```
  var map = new OpenLayers.Map("map_id", {some_options_here});
  ```
- Only indicating a set of options. This way we can later set the DOM element where the map will be rendered:
  ```
  var map = new OpenLayers.Map({some_options_here});
  ```

How to do it...

Perform the following steps:

1. Create a DOM element to render the map:
   ```
   <div id="ch1_map_options" style="width: 100%;
   height: 100%;"></div>
   ```
2. Define some map options:
   ```
   var options = {
       div: "ch1_map_options",
       projection: "EPSG:4326",
       units: "dd",
       displayProjection: new OpenLayers.Projection("EPSG:900913"),
       numZoomLevels: 7
   };
   ```
3. Create the map by passing `options`:
   ```
   var map = new OpenLayers.Map(options);
   ```
4. Add the `MousePosition` control to see the mouse position over the map:
   ```
   map.addControl(new OpenLayers.Control.MousePosition());
   ```
5. Add a WMS layer and set the map view on some desired place:
   ```
   var wms = new OpenLayers.Layer.WMS("OpenLayers WMS Basic",
   "http://vmap0.tiles.osgeo.org/wms/vmap0",
   {
       layers: 'basic'
   });
   map.addLayer(wms);
   map.setCenter(new OpenLayers.LonLat(-100, 40), 5);
   ```

How it works...

In this case we have used five map options to initialize our `OpenLayers.Map` instance.

We have used the `div` option to pass the identifier of the DOM element where the map will be rendered: `div: "ch1_map_options"`.

The `OpenLayers.Map` class uses some default values for most of its options: `projection="EPSG:4326"`, `units="degrees"`, and so on.

The `projection` option is used to set the projection used by the map to render data from layers: `projection: "EPSG:4326"`. Take into account it must be a string with the projection code. On other classes or options it can also be an `OpenLayers.Projection` instance.

There are some implications with the map's projection. Firstly, the tiles to fill WMS layers will be requested using the map's projection, if no other projection is explicitly used by the layer. So you need to be sure the WMS server accepts the projection. Secondly, data from vector layers will be translated from the specific projection of every vector layer to the map's projection, so you will need to set the vector layer's projection options while creating them.

For translations other than `EPSG:4326` and `EPSG:900913`, you need to include the `Proj4js` project (`http://proj4js.org`) in your web application.

Teaching map projections is beyond the scope of this book. A great description can be found on Wikipedia (`http://en.wikipedia.org/wiki/Map_projection`).

EPSG codes are a way to name and classify the set of available projections. The site Spatial Reference (`http://spatialreference.org`) is a great place to find more information about them.

The `units` option specifies that the units used by the map are decimal degrees: `units: "dd"`. This option is related to some resolution options.

The `displayProjection` option allows us to specify the projection that must be used to show the mouse position: `displayProjection: new OpenLayers.Projection("EPSG:900913")`. In this case, our map is in the `EPSG:4326` projection, also known as `WGS84`, with degree units but we show mouse position in `EPSG:900913`, also known as **Spherical Mercator**, which is in meter unit coordinates.

Finally, the `numZoomLevels` sets the number of available zoom levels the user can change. A value of `7` means the user can go from *zoom level 0* to *zoom level 6*.

There's more...

Imagery from sources such as Google Maps or OpenStreetMap are special cases where the pyramid of images is previously created with the Spherical Mercator projection – EPSG:900913. This means you can't set the projection when requesting tiles because it is implicit.

If you put a layer in a different projection other than the one used by the map, it will be automatically disabled.

See also

- The *Understanding base and non-base layers*, recipe
- The *Managing map's stack layers*, recipe
- The *Managing map's controls*, recipe
- The *Working with projections*, recipe in *Chapter 8*, *Beyond the Basics*.

Managing map's stack layers

Map is the core concept in OpenLayers. It allows us to visualize information from different kinds of layers and brings us methods to manage layers attached to it.

In this recipe, we will learn how to control layers. This is important because add, remove, or reorder layers are very common operations we need to do on almost every web mapping application.

The application will show a map on the left and a control panel on the right, with some buttons to control the layers.

Remember we have used the Dojo toolkit framework (`http://dojotoolkit.org`) to code a nicer and richer application to show the recipes of this book.

Because of this, you can see strange attributes in the HTML elements such as `dojoType="dijit.form.Button"` or `onClick="topLayer"`. Do not worry about it, there is no impact in the OpenLayers code we are covering in this book.

How to do it...

1. Start by creating an `index.html` file to put the code needed to create the application layout. We place it within a table. On the left we put the map:

```
<table class="tm">
    <tr>
        <td class="left">
            <div id="ch1_managing_layers"
                style="width: 100%; height: 500px;">
            </div>
        </td>
```

2. And, on the right we put the controls:

```
        <td class="right">
            <p>Maximize the layer switcher control to see
                the map layers and move it clicking the
                buttons:</p>

            <table class="tb">
                <tr>
                    <td>Select layer:</td>
                    <td>
                        <select id="layerSelection"
                            data-dojo-type=
                            "dijit.form.Select">
                            <option value="JPL">
                                JPL</option>
                            <option value="WorldMap">
                                WorldMap</option>
                            <option value="Canada">
                                Canada</option>
                        </select>
                    </td>
```

```
                    </tr>
                    <tr>
                        <td>Move to top:</td>
                        <td><button dojoType=
                            "dijit.form.Button" onClick=
                            "topLayer">Top</button></td>
                    </tr>
                    <tr>
                        <td>Move up:</td>
                        <td><button dojoType=
                             "dijit.form.Button" onClick=
                             "raiseLayer">Up</button></td>
                    </tr>
                    <tr>
                        <td>Move down:</td>
                        <td><button dojoType=
                            "dijit.form.Button" onClick=
                            "lowerLayer">Down</button></td>
                    </tr>
                    <tr>
                        <td>Move to bottom:</td>
                        <td><button dojoType=
                            "dijit.form.Button" onClick=
                            "bottomLayer">Bottom</button>
                        </td>
                    </tr>
                </table>

            </td>
        </tr>
    </table>
```

3. Create an `OpenLayers.Map` instance working in the `allOverlays` mode:

```
var map = new OpenLayers.Map("ch1_managing_layers", {
    allOverlays: true
});
```

4. Add some layers to the map:

```
var jpl = new OpenLayers.Layer.WMS("JPL",
    [
    "http://t1.hypercube.telascience.org/tiles?",
    "http://t2.hypercube.telascience.org/tiles?",
    "http://t3.hypercube.telascience.org/tiles?",
    "http://t4.hypercube.telascience.org/tiles?"
    ],
```

```
        {
            layers: 'landsat7'
        });
    var worldmap = new OpenLayers.Layer.WMS("WorldMap",
        "http://vmap0.tiles.osgeo.org/wms/vmap0",
    {
        layers: 'basic',
        format: 'image/png'
    },
    {
        opacity: 0.5
    });
    var canada = new OpenLayers.Layer.WMS("Canada",
        "http://www2.dmsolutions.ca/cgi-bin/mswms_gmap",
    {
        layers: "bathymetry,land_fn,park",
        transparent: "true",
        format: "image/png"
    },
    {
        opacity: 0.5
    });
    map.addLayers([jpl, worldmap, canada]);
```

5. Add a layers switcher control (to show the layers) and center the map view:

```
map.addControl(new OpenLayers.Control.LayerSwitcher({
    ascending: false
}));
map.setCenter(new OpenLayers.LonLat(-100, 40), 4);
```

6. Finally, add the JavaScript code that will react when the previous four buttons were clicked:

```
function raiseLayer() {
    var layerName =
        dijit.byId('layerSelection').get('value');
    var layer = map.getLayersByName(layerName)[0];
    map.raiseLayer(layer, 1);
}
function lowerLayer() {
    var layerName =
        dijit.byId('layerSelection').get('value');
    var layer = map.getLayersByName(layerName)[0];
    map.raiseLayer(layer, -1);
}
```

```
function topLayer() {
    var layerName =
        dijit.byId('layerSelection').get('value');
    var layer = map.getLayersByName(layerName)[0];
    var lastIndex = map.getNumLayers()-1;
    map.setLayerIndex(layer, lastIndex);
}
function bottomLayer() {
    var layerName =
        dijit.byId('layerSelection').get('value');
    var layer = map.getLayersByName(layerName)[0];
    map.setLayerIndex(layer, 0);
}
```

How it works...

There is not much to say about the HTML code for the layout. We have used a table to put the map on the left and the set of buttons on the right. In addition, we have associated actions to the buttons that will be executed when they are clicked.

With respect to the OpenLayers code, we have created the map instance working in the `allOverlays` mode. This will let us move any layer without being worried about a base layer:

```
var map = new OpenLayers.Map("ch1_managing_layers", {
    allOverlays: true
});
```

Later, we created three WMS layers and added them to the map. For some of them we have set the `opacity` property to `50%` to see through them:

```
map.addLayers([jpl, worldmap, canada]);
```

It is very important to note that we have used the same name for the option's value attribute in the HTML select element as we have used for the layer. Later, this will let us select a map's layer by its name.

Next, we have added an `OpenLayers.Control.LayerSwitcher` control by setting its `ascending` property to `false`:

```
map.addControl(new OpenLayers.Control.LayerSwitcher({
    ascending: false
}));
```

You can think of the map as storing layers in a stack and they are rendered from bottom to top, so the above layers can hide beneath the below layers depending on its opacity and extent.

By default the `ascending` property is `true`, and the layer switcher control shows the layers of the map in the reverse order, that is, the bottom layer is drawn first in the control and the top layer is drawn last. You can avoid this by setting `ascending` to `false`.

Finally, the only thing we need to take a look at is the code responsible for button actions, which is the most interesting code in this recipe.

Let's take a look to the `raiseLayer()` action (which is very similar to `lowerLayer()` action):

```
function raiseLayer() {
    var layerName = dijit.byId('layerSelection').get('value');
    var layer = map.getLayersByName(layerName)[0];
    map.raiseLayer(layer, 1);
}
```

First, we get the name of the currently selected layer in the select element (don't worry if you don't understand that line completely, it is more related to the Dojo framework than to OpenLayers).

Then, we use the `map.getLayersByName()` method, which returns an array with all the layers that have the specified name. Because of this, we get the first element of the array.

Now we have a reference to the layer instance. We can raise it in the map using the `map.raiseLayer()` method. You can raise it by one or more positions indicating a `delta` number or, like in the `lowerLayer()` function, you can lower it by one or more positions indicating a negative value.

Internally `OpenLayers.Map` stores layers in an array (the `layers` attribute) and they are rendered in the order they are stored in the array (so the first element is the bottom layer).

The `topLayer()` and `bottomLayer()` actions are similar too, they move the specified layer to the top or bottom of the stack. They both work using the `map.setLayerIndex()` method, which is responsible to move a layer to a specified position.

The method `map.setLayerIndex()` is used internally by `map.raiseLayer()` to move layers.

Because the bottom layer corresponds to the first layer in the array of layers, the `bottomLayer()` action is the easiest to implement because we simply need to move the layer to the first position:

```
function bottomLayer() {
    var layerName = dijit.byId('layerSelection').get('value');
    var layer = map.getLayersByName(layerName)[0];
    map.setLayerIndex(layer, 0);
}
```

For the `topLayer()` actions, we need to move the layer to the last position. To do this, we can get help from the `map.getNumLayers()` method, which returns the total number of layers in the map. In this way, if we have four layers in the map, the last corresponds to the index `3` (because the index value changes from `0` to `3`).

```
function topLayer() {
    var layerName = dijit.byId('layerSelection').get('value');
    var layer = map.getLayersByName(layerName)[0];
    var lastIndex = map.getNumLayers()-1;
    map.setLayerIndex(layer, lastIndex);
}
```

There's more...

The `OpenLayers.Map` class has plenty of methods to manipulate the contained layers. We have seen a few in these recipes, to add, get layers by name, move up or down in the stack, and so on. But you can find more methods to remove layers, get layers by their position, and so on.

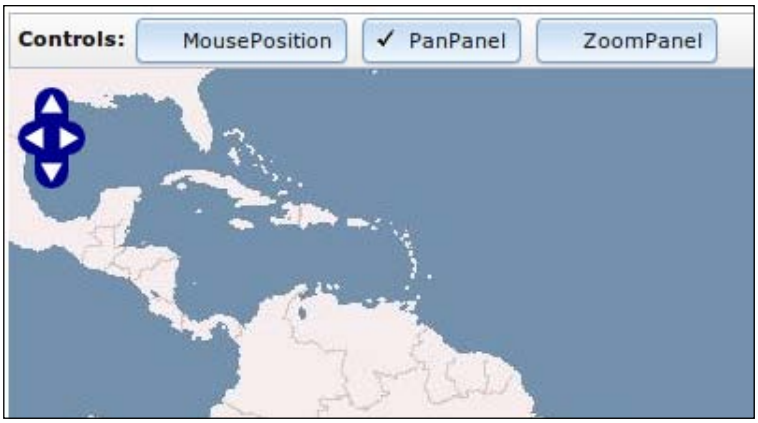
See also

- The *Managing map's controls* recipe
- The *Moving around the map view* recipe
- The *Restricting the map extent* recipe

Managing map's controls

OpenLayers comes with lots of controls to interact with the map: pan, zoom, show overview map, edit features, and so on.

In the same way as layers, the `OpenLayers.Map` class has methods to manage the controls attached to the map.



How to do it...

Follow the given steps:

1. Create a new HTML file and add the OpenLayers dependencies.
2. Now, add the required code to create the buttons and `div` element to hold the map instance:

```
<div class="sample_menu" dojoType="dijit.MenuBar">
    <span class="title">Controls: </span>
    <div dojoType="dijit.form.ToggleButton"
        iconClass="dijitCheckBoxIcon"
        onChange="updateMousePosition">MousePosition
    </div>
    <div dojoType="dijit.form.ToggleButton"
        iconClass="dijitCheckBoxIcon"
        onChange="updatePanPanel">PanPanel</div>
    <div dojoType="dijit.form.ToggleButton"
        iconClass="dijitCheckBoxIcon"
        onChange="updateZoomPanel">ZoomPanel</div>
</div>
<!-- Map DOM element -->
<div id="ch1_managing_controls" style="width: 100%;
height: 500px;"></div>
```

3. Within the `script` element section, create the map instance:

```
var map = new OpenLayers.Map("ch1_managing_controls", {
    controls: [
    new OpenLayers.Control.Navigation()
    ]
});
```

4. Add some layers to the map and center the view:

```
var wms = new OpenLayers.Layer.WMS("OpenLayers WMS Basic",
"http://vmap0.tiles.osgeo.org/wms/vmap0",
{
    layers: 'basic'
},
{
    wrapDateLine: false
});
map.addLayer(wms);
// Center the view
map.setCenter(OpenLayers.LonLat.fromString("0,0"),3);
```

5. Finally, add the actions code associated to the buttons:

```
function updateMousePosition(checked) {
    if(checked) {
        map.addControl(new
            OpenLayers.Control.MousePosition());
    } else {
        var controls =
            map.getControlsByClass
            ("OpenLayers.Control.MousePosition");
        console.log(controls);
        map.removeControl(controls[0]);
    }
}
function updatePanPanel(checked) {
    if(checked) {
        map.addControl(new
            OpenLayers.Control.PanPanel());
    } else {
        var controls = map.getControlsByClass
            ("OpenLayers.Control.PanPanel");
        map.removeControl(controls[0]);
    }
}
function updateZoomPanel(checked) {
```

```
        if(checked) {
            // Place Zoom control at specified pixel
            map.addControl(new OpenLayers.Control.ZoomPanel()
                , new OpenLayers.Pixel(50,10));
        } else {
            var controls = map.getControlsByClass
                ("OpenLayers.Control.ZoomPanel");
            map.removeControl(controls[0]);
        }
    }
```

How it works...

Every button action function checks if the toggle button is checked or unchecked and depending on the value we add or remove the control to the map:

```
if(checked) {
    // Place Zoom control at specified pixel
    map.addControl(new OpenLayers.Control.ZoomPanel(),
    new OpenLayers.Pixel(50,10));
} else {
    var controls = map.getControlsByClass
    ("OpenLayers.Control.ZoomPanel");
    map.removeControl(controls[0]);
}
```

Adding a control is fairly simple through the `map.addControl()` method, which, given a control instance—and, optionally a `OpenLayers.Pixel` instance—adds the control to the map at the specified position.

Usually, a control position is controlled by modifying the `top` and `left` values in the CSS class used by the control. If you use a `OpenLayers.Pixel` value to position the control, then that value will overwrite the CSS ones.

To remove a control we need to have a reference to the instance that has to be removed. The method `map.getControlsByClass()` returns an array of controls of the specified class and helps us to get a reference to the desired control. Next, we can remove it with `map.removeControl()`.

There's more...

Note, in this recipe we have centered the map's view passing a `OpenLayers.LonLat` instance created in a different way. Instead of using the `new` operator, we have used the method `OpenLayers.LonLat.fromString`, which created a new instance from a string:

```
map.setCenter(OpenLayers.LonLat.fromString("0,0"),3);
```

In addition, the map instance created in this recipe has initialized with only one control, `OpenLayers.Control.Navigation()`, which allows us to navigate the map using the mouse:

```
var map = new OpenLayers.Map("ch1_managing_controls", {
    controls: [
    new OpenLayers.Control.Navigation()
    ]
});
```

Passing an empty array to the `controls` property creates a map instance without any control associated with it. In addition, without specifying the `controls` property, OpenLayers creates a set of default controls for the map, which includes the `OpenLayers.Control.Navigation` and `OpenLayers.Control.PanZoom` controls.

See also

- The *Managing map's stack layers* recipe
- The *Moving around the map view* recipe

Moving around the map view

Unless you want to create a completely static map, without the controls required for the user to pan or zoom, you would like the user to be able to navigate and explore the map.

There can be situations when the controls are not enough. Imagine a web application where the user can make a search, such as Everest, and the application must find its location and fly to it. In this case, you need to navigate by code and not by using a control.

This recipe shows you some of the `OpenLayers.Map` class methods that will allow you to improve the user's experience.

The application layout contains three main sections. At the top there is a label to show the current map center position and zoom level. It is automatically updated when the map is moved or the zoom is changed.

The map is in the center and there are a bunch of controls on the right to set and test the main map methods to interact with the view.

As you will see, the map has no control attached to it, so the only way to interact with it is through the right controls.

We omit the HTML code necessary to create the application layout, so if you are interested in the HTML code you can take a look at the source code available on the Packt Publishing website.

How to do it...

1. Create an HTML file with OpenLayers dependencies.

The HTML code to create the buttons and layout of the previous screenshot is extensive and not related to the goal of the book, so here we avoid writing it.

2. Add a `div` element to hold the map instance:

```
<div id="ch1_moving_around" style="width: 100%;
    height: 500px;"></div>
```

3. Create a map instance. This time we specify a listener for some events that will update the center and zoom values of the label on top of the map:

```
var map = new OpenLayers.Map("ch1_moving_around", {
    controls: [],
    eventListeners: {
        "move": changeListener,
        "moveend": changeListener,
        "zoomend": changeListener
    }
});
function changeListener() {
    var center = map.getCenter();
    document.getElementById("center").innerHTML =
        "("+center.lon + " lon , " + center.lat +
        " lat)";
    var zoom = map.getZoom();
    document.getElementById("zoom").innerHTML = zoom +
    " level";
}
```

4. Add one layer and center the view:

```
var wms = new OpenLayers.Layer.WMS("OpenLayers WMS Basic",
"http://vmap0.tiles.osgeo.org/wms/vmap0",
{
    layers: 'basic'
});
map.addLayer(wms);
map.setCenter(new OpenLayers.LonLat(0, 0), 2);
```

5. Insert the code that will be executed by the button actions:

```
function moveByPx() {
    var x = dijit.byId('movebyxpix').get('value');
    var y = dijit.byId('movebyypix').get('value');

    map.moveByPx(x,y);
}
function moveTo() {
    var lon = dijit.byId('movetolon').get('value');
    var lat = dijit.byId('movetolat').get('value');
    var zoom = dijit.byId('movetozoom').get('value');
    var force = dijit.byId
        ('movetoforceZoomChange').get('checked');
    var drag = dijit.byId
        ('movetodragging').get('checked');

    map.moveTo(new OpenLayers.LonLat(lon, lat), zoom, {
        forceZoomChange: force,
        dragging: drag
    });
}
function setCenter() {
    var lon = dijit.byId('setCenterlon').get('value');
    var lat = dijit.byId('setCenterlat').get('value');
    var zoom = dijit.byId('setCenterzoom').get('value');
    var force = dijit.byId
        ('setCenterforceZoomChange').get('checked');
    var drag = dijit.byId
        ('setCenterdragging').get('checked');

    map.setCenter(new OpenLayers.LonLat(lon, lat),
    zoom, {
        forceZoomChange: force,
        dragging: drag
    });
}
function pan() {
    var x = dijit.byId('panxpix').get('value');
    var y = dijit.byId('panypix').get('value');
    var anim = dijit.byId('pananimate').get('checked');
    var drag = dijit.byId('pandragging').get('checked');

    map.pan(x,y, {
        animate: anim,
```

```
            dragging: drag
        });
    }
    function panTo() {
        var lon = dijit.byId('panTolon').get('value');
        var lat = dijit.byId('panTolat').get('value');

        map.panTo(new OpenLayers.LonLat(lon, lat));
    }
```

How it works...

To update the center and zoom level values at the top, we have instantiated the `Map` object with some event listeners attached to it. Actually, the same listener function is attached to all three events:

```
var map = new OpenLayers.Map("ch1_moving_around", {
    controls: [],
    eventListeners: {
        "move": changeListener,
        "moveend": changeListener,
        "zoomend": changeListener
    }
});
```

Within the `changeListener()` function we use `map.getCenter()`, which returns an `OpenLayers.LonLat` object, and `map.getZoom()` to get the current values and update the top-left label.

```
function changeListener() {
    var center = map.getCenter();
    document.getElementById("center").innerHTML =
        "("+center.lon + " lon , " + center.lat + " lat)";
    var zoom = map.getZoom();
    document.getElementById("zoom").innerHTML = zoom + " level";
}
```

For every button, an action is executed. They are responsible to get the required values and invoke a `map` method.

The `map.moveByPx()` method moves the map by a delta value specified in pixels. Note, it moves the map; it doesn't pan, so don't expect any effect.

```
function moveByPx() {
    var x = dijit.byId('movebyxpix').get('value');
    var y = dijit.byId('movebyypix').get('value');

    map.moveByPx(x,y);
}
```

The `map.moveTo()` method is similar to the previous one but moves the view to a specified position (instead of an increment) and is specified with an `OpenLayers.LonLat` instance.

The `map.setCenter()` method is similar to `map.moveTo()` but it centers the view on the specified location.

Finally, there are two pan-related actions, which make nice smooth movements. The `map.pan()` method moves the view with a pan effect specified by a pixel delta. The `map.panTo()` method does something similar, it moves the view to a specified location.

See also

- The *Managing map's stack layers* recipe
- The *Restricting the map extent* recipe

Restricting the map extent

Often, there are situations where you are interested to show data to the user but only for a specific area, which your available data corresponds to (a country, a region, a city, and so on).

In this case, there is no point in allowing the user to explore the whole world, so you need to limit the extent the user can navigate.

In this recipe, we present some ways to limit the area a user can explore.

How to do it...

1. Create a map instance. Take a look at the couple of properties used in the constructor:

```
var map = new OpenLayers.Map("ch1_restricting_view", {
    maxExtent: OpenLayers.Bounds.fromString
        ("-180,-90,180,90"),
    restrictedExtent: OpenLayers.Bounds.fromString
        ("-180,-90,180,90")
});
```

2. As always, add some layer to see content and center the view:

```
var wms = new OpenLayers.Layer.WMS("OpenLayers WMS Basic",
"http://vmap0.tiles.osgeo.org/wms/vmap0",
{
    layers: 'basic'
});
map.addLayer(wms);
map.setCenter(new OpenLayers.LonLat(0, 0), 2);
```

3. Add the functions that will be executed when buttons are clicked:

```
function updateMaxExtent() {
    var left = dijit.byId('left_me').get('value');
    var bottom = dijit.byId('bottom_me').get('value');
    var right = dijit.byId('rigth_me').get('value');
    var top = dijit.byId('top_me').get('value');
```

```
        map.setOptions({
            maxExtent: new OpenLayers.Bounds(left, bottom,
            right, top)
        });
    }
    function updateRestrictedExtent() {
        var left = dijit.byId('left_re').get('value');
        var bottom = dijit.byId('bottom_re').get('value');
        var right = dijit.byId('rigth_re').get('value');
        var top = dijit.byId('top_re').get('value');
        map.setOptions({
            restrictedExtent: new OpenLayers.Bounds(left,
            bottom, right, top)
        });
    }
```

How it works...

As you have seen, the map has been instantiated using the two properties `maxExtent` and `restrictedExtent`, which are responsible for limiting the area of the map we can explore.

Although similar, these two properties have different meanings. Setting the `maxExtent` property limits the viewport so its center cannot go outside the specified bounds. By setting the `restrictedExtent` property the map itself cannot be panned beyond the given bounds.

The functions that react when buttons are clicked get the values from the input fields and apply the new values through the `map.setOptions()` method:

```
map.setOptions({
    maxExtent: new OpenLayers.Bounds(left, bottom, right, top)
});
```

We can pass the same properties we use when creating a new `OpenLayers.Map` instance to the `map.setOptions()` method and it will take care to update them.

There's more...

Limiting the map extent is not the only way to limit the information we show to the user. The layers have also similar properties to filter or limit the information they must render.

See also

- The *Moving around the map view* recipe

2
Adding Raster Layers

In this chapter we will cover the following:

- Using Google Maps imagery
- Using Bing imagery
- Adding WMS layer
- Wrapping the date line options
- Changing the zoom effect
- Changing the layer opacity
- Using WMS with single tile mode
- Buffering the layer data to improve the map navigation
- Creating an image layer
- Setting the tile size in WMS layers

Introduction

This chapter is all about working with raster layers. We have tried to summarize, with a set of recipes, the most common and important use cases you can find day-to-day when working with OpenLayers.

Imagery is one of the most important kinds of data to work with in a GIS system.

OpenLayers offers several classes to integrate with different imagery providers, from proprietary providers such as Google Maps and Bing Maps, to Open Source ones such as OpenStreetMap or even any WMS service provider.

The base class for any layer type is the `OpenLayers.Layer` class, which offers a set of common properties and defines the common behavior for any other classes.

In addition, many layers inherit from the `OpenLayers.Layer.Grid` class, which divides the layer into zoom levels. This way each zoom level covers the same area but uses a greater set of tiles. For example, at level zero a grid with one tile covers the whole world, at level one a grid with four tiles covers the whole world, and so on. As we can see, on each level, the number of tiles and their resolution increases.

This chapter introduces you to the use of raster layers, with special attention to the WMS layers, and how to manage the most common properties.

Using Google Maps imagery

Google Maps is probably the most known web map application around the world. Their imageries, in the way of tiled layers, are well known by people; they are accustomed to their layer style and because of this you may be interested in using them in your own web mapping project.

OpenLayers counts with the `OpenLayers.Layer.Google` class, which is in fact a wrapper code around the Google Maps API, that allows us to use the Google Maps tiles in a homogeneous way within the OpenLayers API.

Do not confuse Google Maps API with the Google Maps imagery. Google Maps API is a bunch of JavaScript code, which is free to use, while the access to the Google Maps imagery has some usage restrictions and, depending on the number of hits, will be subject to some payments.

How to do it...

To use Google Maps imagery, perform the following steps:

1. Create an HTML file and add the OpenLayers dependencies.
2. Include the Google Maps API as follows:

```
<script type="text/javascript" src="http://maps.google.com/maps/
api/js?v=3.5&sensor=false"></script>
```

3. Add a `div` element to hold the map, as follows:

```
<!-- Map DOM element -->
<div id="ch2_google" style="width: 100%; height: 100%;"></div>
```

4. Within a `script` element, add the code to create the map instance and add a layer switcher control, as follows:

```
<!-- The magic comes here -->
<script type="text/javascript">

    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch2_google");
    map.addControl(new OpenLayers.Control.LayerSwitcher());
```

5. Create some Google based maps and add to the map, as follows:

```
    var streets = new OpenLayers.Layer.Google("Google Streets", {
        numZoomLevels: 20
    });
    var physical = new OpenLayers.Layer.Google("Google Physical", {
        type: google.maps.MapTypeId.TERRAIN
    });
    var hybrid = new OpenLayers.Layer.Google("Google Hybrid", {
        type: google.maps.MapTypeId.HYBRID, numZoomLevels: 20
    });
    var satellite = new OpenLayers.Layer.Google
     ("Google Satellite", {
        type: google.maps.MapTypeId.SATELLITE, numZoomLevels: 22
    });
    map.addLayers([physical, streets, hybrid, satellite]);
```

6. Finally, center the map on a desired location, as follows:

```
    map.setCenter(new OpenLayers.LonLat(0, 0), 2);
</script>
```

How it works...

As you can see, the code has three main sections. First we have placed a `div` element, which will be used for the map, as follows:

```
<div id="ch2_google" style="width: 100%; height: 100%;"></div>
```

Next, we have included the Google Maps API code, as follows:

```
<script type="text/javascript"
src="http://maps.google.com/maps/api/js?v=3.5&sensor=false"></script>
```

Remember that OpenLayers simply acts as a wrapper, so we need the real Google Maps API code in our application.

Within a `<script type="text/javascript"> </script>` element, we have added the code necessary to initialize the map and add a layer switcher control, as follows:

```
var map = new OpenLayers.Map("ch2_google");
map.addControl(new OpenLayers.Control.LayerSwitcher());
```

Finally, we have added some well known Google Maps layers and centered the map's viewport, as follows:

```
var streets = new OpenLayers.Layer.Google("Google Streets", {
    numZoomLevels: 20
});
var physical = new OpenLayers.Layer.Google("Google Physical", {
    type: google.maps.MapTypeId.TERRAIN
});
var hybrid = new OpenLayers.Layer.Google("Google Hybrid", {
    type: google.maps.MapTypeId.HYBRID, numZoomLevels: 20
});
var satellite = new OpenLayers.Layer.Google("Google Satellite", {
    type: google.maps.MapTypeId.SATELLITE, numZoomLevels: 22
});
map.addLayers([physical, streets, hybrid, satellite]);
map.setCenter(new OpenLayers.LonLat(0, 0), 2);
```

The type of the layers are defined by the Google Maps API class `google.maps.MapTypeId`, which you can find at `http://code.google.com/apis/maps/documentation/javascript/reference.html#MapTypeId`.

Note that we are working with two APIs, OpenLayers and Google Maps API, so it would be good to take a look at the Google Maps API to better understand its capabilities.

Documentation can be found at `https://developers.google.com/maps/documentation/javascript/tutorial`.

There's more...

In this recipe, we have shown you how to use the Google Maps API Version 3 to add the Google imagery to your OpenLayers projects.

For the previous version 2, Google requires you to register as a user and obtain an API key that you need to use to initialize the `OpenLayers.Layer.Google` instance. The key is later used on every tile request to identify you, so Google can know about your usage.

As you have seen, version 3 is much more simple to use within OpenLayers.

See also

- The *Adding WMS layer* recipe
- The *Using Bing imagery* recipe
- The *Understanding base and non-base layers* recipe in *Chapter 1*, *Web Mapping Basics*

Using Bing imagery

Bing Maps, previously known as Virtual Earth, is the mapping service provided by Microsoft.

In the same way as Google Maps, OpenLayers offers an `OpenLayers.Layer.Bing` class, which brings us the possibility to add Bing imagery in our projects.

Getting ready

Bing Maps requires you to register as a consumer user. Once registered, you will get an API key needed to initialize the `OpenLayers.Layer.Bing` layer and that will be used with every request to authenticate you against the Bing Maps service.

Opposite to Google Maps, Bing does not require any JavaScript code and the `OpenLayers.Layer.Bing` class does not act as a wrapper. Bing Maps offer a REST service to directly access tiles using your API key.

You can find out how to register as a user at `http://msdn.microsoft.com/en-us/library/ff428642.aspx`.

In addition you can learn about Bing Maps REST Services at `http://msdn.microsoft.com/en-us/library/ff701713.aspx`.

At this point, it is assumed that you have an API key to be used in the next code.

How to do it...

In this section we will see how to use Bing imagery. To use Bing imagery, perform the following steps:

1. Create an HTML file and add the OpenLayers dependencies.
2. Add a `DOM` element to place the map, as follows:

```
<div id="ch2_bing" style="width: 100%; height: 100%;"></div>
```

3. Within a `script` element create the map instance and add a layer switcher control, as follows:

```
<script type="text/javascript">
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch2_bing");
    map.addControl(new OpenLayers.Control.LayerSwitcher());
```

4. Create some Bing layers, add to the map and center the map's viewport, as follows:

```
    var bingApiKey = "your_bing_API_must_be_put_here";
    var road = new OpenLayers.Layer.Bing({
        name: "Road",
        type: "Road",
        key: bingApiKey
    });
    var hybrid = new OpenLayers.Layer.Bing({
        name: "Hybrid",
```

```
            type: "AerialWithLabels",
            key: bingApiKey
        });
        var aerial = new OpenLayers.Layer.Bing({
            name: "Aerial",
            type: "Aerial",
            key: bingApiKey
        });
        map.addLayers([road, hybrid, aerial]);

        map.setCenter(new OpenLayers.LonLat(0, 0), 2);
    </script>
```

How it works...

The main point to take into account in this recipe is that we are using Microsoft services. We request an URL using our API key and get a tile. Because of this, every Bing layer must include a `key` parameter while instantiating, as follows:

```
var bingApiKey = "your_bing_API_must_be_put_here";
var road = new OpenLayers.Layer.Bing({
    name: "Road",
    type: "Road",
    key: bingApiKey
});
```

We know about the `name` parameter as it is common in all layers. The `name` parameter is used to put a descriptive name for the layer and it will be used by switcher control.

As previously mentioned, the `key` parameter is used on every tile request and identifies us as registered Bing consumer users.

The `type` parameter is necessary to specify the kind of tile we want to get from Bing Maps. Bing offers Road, Aerial, or AerialWithLabels among other types.

You can find more information about Bing Maps layer types at `http://msdn.microsoft.com/en-us/library/ff701716.aspx`.

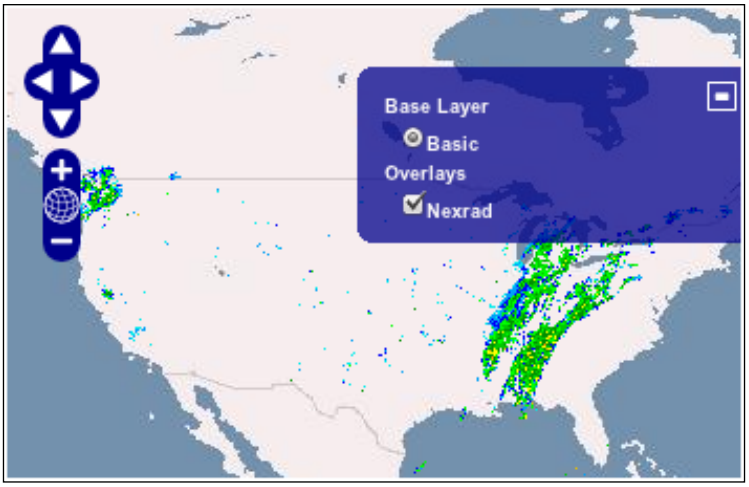
See also

- The *Using Google Maps imagery* recipe
- The *Adding WMS layer* recipe
- The *Understanding base and non-base layers* recipe in *Chapter 1*, *Web Mapping Basics*

Adding WMS layer

Web Map Service (**WMS**),is a standard developed by the **Open Geospatial Consortium** (**OGC**) implemented by many geospatial servers, among which we can find the free and open source projects GeoServer (`http://geoserver.org`) and MapServer (`http://mapserver.org`). More information on WMS can be found at `http://en.wikipedia.org/wiki/Web_Map_Service`.

As a very basic summary, you can understand a WMS server as a normal HTTP web server that accepts request with some GIS-related parameters (such as projection, bounding box, and so on) and returns a map similar to the following screenshot:



We are going to work with remote WMS servers, so it is not necessary you have one installed. As an advise, note that we are not responsible for these servers and that they may have problems, or are not available when you read this section.

Any other WMS server can be used, but the URL and layer name must be known.

How to do it...

To add a WMS layer, perform the following steps:

1. Create an HTML file and add the OpenLayers dependencies.
2. Add a `div` element to hold the map, as follows:

```
<div id="ch2_wms_layer" style="width: 100%; height: 100%;"></div>
```

3. Create the map instance as follows:

```
// Create the map using the specified DOM element
var map = new OpenLayers.Map("ch2_wms_layer");
```

4. Now, add two WMS layers. The first will be the base layer and the second will be an overlay, as follows:

```
// Add a WMS layer
var wms = new OpenLayers.Layer.WMS("Basic",
  "http://vmap0.tiles.osgeo.org/wms/vmap0",
{
    layers: 'basic'
});

// Add Nexrad WMS layer
var nexrad = new OpenLayers.Layer.WMS("Nexrad",
  "http://mesonet.agron.iastate.edu/cgi-bin/wms/nexrad/n0r.cgi",
{
    layers: "nexrad-n0r",
    transparent: "true",
    format: 'image/png'
},
{
    isBaseLayer: false
});

map.addLayers([wms, nexrad]);
```

5. Finally, we add a layer switcher control and center the view, as follows:

```
// Add layer switcher control
map.addControl(new OpenLayers.Control.LayerSwitcher());
// Set the center of the view
map.setCenter(new OpenLayers.LonLat(-90,40), 4);
```

How it works...

The `OpenLayers.Layer.WMS` class constructor requires four arguments to be instantiated (actually the fourth is optional), which are:

```
new OpenLayers.Layer.WMS(name, url, params, options)
```

The parameters are as follows:

- The `name` is common to all layers and is used as a user-friendly description
- The `url` is a string that must point to the WMS server

- The `params` parameter is an object and can contain any parameters used in a WMS request: layers, format, styles, and so on

Check the WMS standard to know which parameters you can use within the `params`.

The use of layers is mandatory, so you always need to specify this value. In addition, if using the SRS WMS request parameter, take into account that it is always ignored, because it is taken from the projection of the base layer or the map's projection.

- The `options` parameter is an optional object that contains specific properties for the layer object: `opacity`, `isBaseLayer`, among others

In this recipe, we have added one base layer as follows:

```
var wms = new OpenLayers.Layer.WMS("Basic",
  "http://vmap0.tiles.osgeo.org/wms/vmap0",
{
    layers: 'basic'
});
```

It makes use of the `name`, `url`, and `params` parameters, indicating the `basic` is the only layer to be requested.

Later, we have added a second overlay layer with weather radar information from NEXRAD (`http://en.wikipedia.org/wiki/NEXRAD`), at the Iowa State University servers (you can find more information at `http://mesonet.agron.iastate.edu/ogc`) as follows:

```
var nexrad = new OpenLayers.Layer.WMS("Nexrad",
  "http://mesonet.agron.iastate.edu/cgi-bin/wms/nexrad/n0r.cgi",
{
    layers: "nexrad-n0r",
    transparent: "true",
    format: 'image/png'
},
{
    isBaseLayer: false
});
```

In this case, in addition to the `layers` parameter, we have used the `transparent` and `format` parameters.

The `format` parameter is used in a WMS request to specify what image format we want to receive the images in.

The `transparent` property is set to `true`. If it were not set, we would get white tiles with some colored radar data that will hide the base layer. Make the test using `transparent: "false"`.

For this layer, we have also set the layer parameter `isBaseLayer` to `false`, to indicate we want it to act as an overlay.

There's more...

WMS servers returns images no matter whether there is information in the bounding box we are requesting or not.

The previously mentioned Nexrad WMS layer, showing the tile images with a white background was not desirable, so we used the `transparent` parameter to fix the issue.

When you set the `transparent` parameter to `true`, no matter which format you specify, internally the WMS class ensures the requests are made using the `format image/png` or `image/gif` to guarantee the transparency of those pixels that have no data.

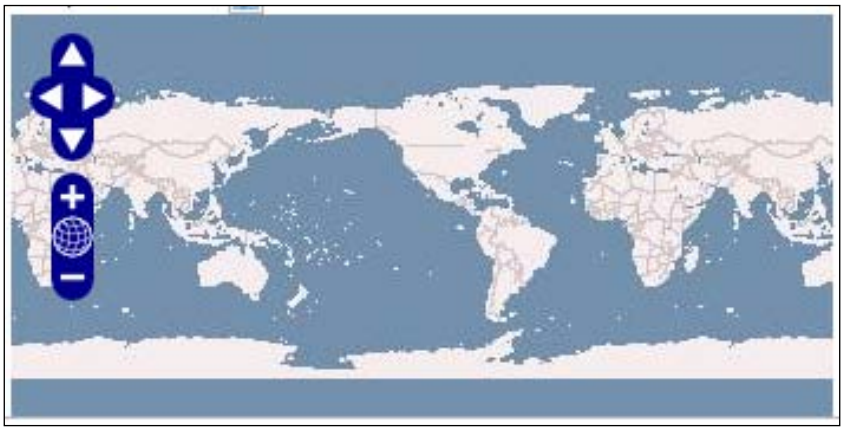
Finally, remember we can pass any parameter defined by WMS standard in the requests, by just specifying them in the `params` parameter.

See also

- The *Using Google Maps imagery* recipe
- The *Using WMS with single tile mode* recipe
- The *Changing the layer opacity* recipe
- The *Buffering the layer data to improve the map navigation* recipe

Wrapping the date line options

There might be situations where you do not want your map ends at -180 or +180 longitude degrees as you are working in that area and need a continuous map. For example, imagine a map where on the left you can see the end of Russia and at the right Alaska, as shown in the following screenshot:



This property is a common attribute from base class `OpenLayers.Layer` and is called the `wrapDateLine`.

How to do it...

To wrap the date line options, perform the following steps:

1. Create an HTML file and add the OpenLayers dependency.
2. In the beginning, we have put a checkbox to activate/deactivate the wrap data line feature, as follows:

```
Wrap date line: <input dojoType="dijit.form.CheckBox" checked
onChange="wrapDateLine" /> <br/>
```

Do not worry about the `dojoType="dijit.form.CheckBox"` attribute, it is because the Dojo Toolkit (`http://dojotoolkit.org`) is used in the sample.

Think of it as a normal HTML input element.

3. Next, we have added the `DOM` element used to render the map, as follows:

```
<div id="ch2_wrapdataline" style="width: 100%; height: 100%;"></div>
```

4. Within a `script` element, create the map instance, as follows:

```
<script type="text/javascript">
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch2_wrapdataline");
```

5. Now, create a WMS layer specifying the `wrapDateLine` property, as follows:

```
    // Add a WMS layer
    var wms = new OpenLayers.Layer.WMS("OpenLayers WMS Basic",
    "http://labs.metacarta.com/wms/vmap0",
    {
        layers: 'basic'
    },
    {
        wrapDateLine: true
    });
    map.addLayer(wms);

    // Center map view
    map.setCenter(new OpenLayers.LonLat(-110,0), 2);
```

6. Finally, implement the function that will change the `wrapDateLine` property value, as follows:

```
    function wrapDateLine(checked) {
        wms.wrapDateLine = checked;
        wms.redraw();
    }
</script>
```

How it works...

All the magic of this recipe is in the `wrapDateLine` property in the `OpenLayers.Layer` class. You need to set it to `true` to wrap and create a continuous layer on their longitudinal axes.

In addition, we have created a function that reacts to changes in the checkbox to activate/deactivate the `wrapDateLine` property, as in the following code:

```
function wrapDateLine(checked) {
    wms.wrapDateLine = checked;
    wms.redraw();
}
```

Note that after changing the property value we need to redraw the layer so that it takes effect. This is done using the `redraw()` method inherited from the `OpenLayers.Layer` base class.

The `wrapDateLine` property is not a property of the map but a property of every layer; so if you want the whole map to have the same behavior, you need to set it to `true` in all layers.

See also

- The *Adding WMS layer* recipe
- The *Using WMS with single tile mode* recipe

Changing the zoom effect

The panning and zoom effects are very important actions related to the user navigation experience.

In *Chapter 1*, *Web Mapping Basics*, the recipe *Moving around the map view* shows how you can control and create the way the map can be panned.

In the same way you can control the transition effect between two zoom levels on the layers.

The `OpenLayers.Layer` class has a `transitionEffect` property, which determines the effect applied to the layer when the zoom level is changed. For the moment only two values are allowed: `null` and `resize`.

The `null` value means no transition effect will be applied, because when you change the zoom level you probably see how the layer disappears until the tiles at the new zoom level are loaded.

With the `resize` value when we zoom into a level, the current tiles are resized, adapting to the new zoom, until the tiles at the new level are loaded in background. This way images are always visible and we avoid the ugly effect of seeing a blank map for a few seconds.

How to do it...

To change the zoom level, perform the following steps:

1. Create an HTML file and include the required OpenLayers dependencies.
2. For this recipe, we are going to add a checkbox button that allows us to change between the transition effects on a single layer, as follows:

```
Transition effect: <input dojoType="dijit.form.CheckBox" checked
onChange="transitionEffect" /> Resize
```

3. Next, add the `div` element, which holds the map as follows:

```
<!-- Map DOM element -->
<div id="ch2_transition_effect" style="width: 100%; height:
100%;"></div>
```

4. Add the JavaScript that initializes the map and creates one WMS layer, as follows:

```
<!-- The magic comes here -->
<script type="text/javascript">

    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch2_transition_effect");

    // Add a WMS layer
    var wms = new OpenLayers.Layer.WMS("OpenLayers WMS Basic",
    "http://vmap0.tiles.osgeo.org/wms/vmap0",
    {
        layers: 'basic'
    },
    {
        wrapDateLine: true,
        transitionEffect: 'resize'
    });
    map.addLayer(wms);

    // Center map view
    map.setCenter(new OpenLayers.LonLat(0,0), 3);
```

5. Finally, put the function that will toggle the `transitionEffect` property value, as follows:

```
    function transitionEffect(checked) {
        if(checked) {
            wms.transitionEffect = 'resize';
        } else {
            wms.transitionEffect = null;
        }
    }
</script>
```

How it works...

As explained at the beginning of the recipe, all the magic is in the `transitionEffect` property.

As the property is specific to a layer and not an `OpenLayers.Map` property, if you want to apply the same effect to the whole map, you need to set it on all its contained layers.

There's more...

One or more `OpenLayers.Tile.Image` forms a raster layer, so when it is rendered the real work to draw the tiles is made easy by the tiles themselves.

Although the `transitionEffect` is defined in the `OpenLayers.Layer` class (or subclasses), each individual tile is responsible for drawing the transition effect.

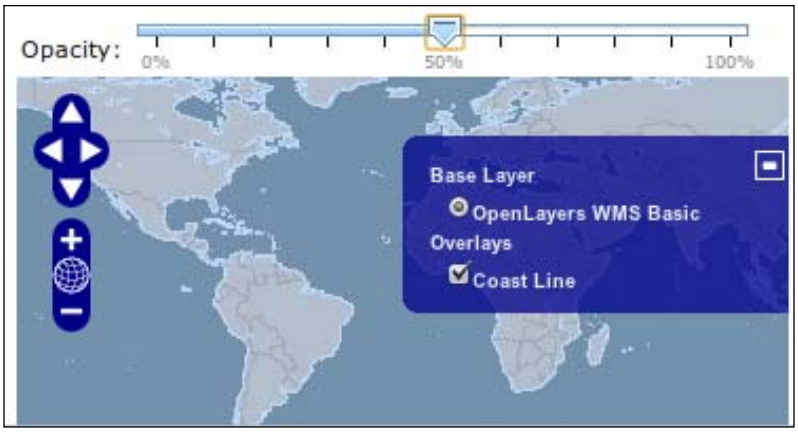
If you plan to create a new zoom transition effect, you will need to take a look at the `OpenLayers.Tile.Image` code as well.

See also

- The *Adding WMS layer* recipe
- The *Changing the layer opacity* recipe
- The *Using WMS with single tile mode* recipe

Changing the layer opacity

When you are working with many layers—both raster and vector layers—you will probably find situations where a layer that is on top of another layer hides the one below it. This is more common when working with raster WMS layers without the `transparent` property set to `true` or tiled layers such as, OpenStreetMaps, Google, and Bing. The layer opacity is set to **50%** in the following screenshot:



The `OpenLayers.Layer` base class has an `opacity` property, implemented by concrete subclasses, that allows us to modify the opacity of the layers. It is a float value that can range from `0.0` (completely transparent) to `1.0` (completely opaque).

How to do it...

The opacity of the layers can be changed. To change the opacity of the layer, perform the following steps:

1. Create an HTML file adding the required OpenLayers dependencies.

We have intentionally omitted the HTML code required for the slider control. Here, we have focused on the code for OpenLayers. If interested in knowing more about the code for the slider, it can be found in the recipe's source code.

2. Add a `div` element to hold the map, as follows:

```
<div id="ch2_opacity" style="width: 100%; height: 100%;"></div>
```

3. Next, create a map instance and add two layers, as follows:

```
// Create the map using the specified DOM element
var map = new OpenLayers.Map("ch2_opacity");

// Add a WMS layer
var wms = new OpenLayers.Layer.WMS("OpenLayers WMS Basic",
"http://vmap0.tiles.osgeo.org/wms/vmap0",
{
    layers: 'basic'
```

```
});
map.addLayer(wms);

// Add coast line layer
var wms2 = new OpenLayers.Layer.WMS("Coast Line",
"http:// vmap0.tiles.osgeo.org/wms/vmap0",
{
    layers: 'coastline_01,coastline_02'
},
{
    isBaseLayer: false
});
map.addLayer(wms2);
```

4. Add a layer switcher control and center the map's viewport, as follows:

```
map.addControl(new OpenLayers.Control.LayerSwitcher());

// Center map view
map.setCenter(new OpenLayers.LonLat(0,0), 3);
```

5. Finally, implement the function that receives the changes on the slider control and changes the layer opacity, as follows:

```
function opacity(value) {
    wms2.setOpacity(value/100);
}
```

How it works...

As we have commented that the `opacity` property is the key in this recipe, the way to modify it is not by changing the attribute value directly but by using the `setOpacity()` method.

The `setOpacity()` method is responsible for modifying the `opacity` property in addition to modifying any `DOM` element (for example, the images of the tiles) and emitting a `changelayer` event, which notifies any listener interested in knowing about any layer changes.

See also

- The *Wrapping the date line options* recipe
- The *Understanding base and non-base layers* recipe in *Chapter 1*, *Web Mapping Basics*
- The *Adding WMS layer* recipe
- The *Buffering the layer data to improve the map navigation* recipe

Using WMS with single tile mode

Web Map Service (**WMS**) is a protocol to serve georeferenced map images.

The basic idea is that, given a bounding box and some other parameters, such as a layer name, the client makes an HTTP request to the WMS server, which computes and returns an image with all the data for the specified layers and within the specified bounding box.

In OpenLayers, when you add a WMS layer to your map, the `OpenLayers.Layer.WMS` instance is provided with some parameters, such as resolutions and tile size. The WMS server computes the right number of tiles for each zoom level and divides the layer in that number of tiles.

This way, when you add a WMS layer to the map, there is not only one request to the server, but one by each tile that forms the current zoom level.

Dividing the WMS layer in tiles can be better from the server-side point of view when it is configured with a cache system. This way tiles are generated once and served many times.

If you have more than one web mapping application using WMS layers that point to the same WMS server, all the tiles can be served from the cache and the load on the server would drastically reduce.

Dividing the layer in tiles isn't the only way you can work with WMS layers; if you need to, you can work in the so-called single tile mode.

In this mode, only one image is used to cover the whole view—the map's bounding box, instead of using a bunch of tiles required for the tiled mode.

Every time the layer must be refreshed (as you move the map or change the zoom level) one request is made to the WMS server requesting data for the new map's bounding box.

As you can see, in single tile mode, the number of requests to the server are much less than those in tiled mode. In contrast, working in tiled mode, each tile request is easy to cache as the tile's bounding boxes are fixed for each zoom level; while in single tile mode each request is usually slightly different than the other (as little changes in the bounding box) and will result in a request to the WMS server, with the consequent computation time.

How to do it...

Follow through the steps to use WMS in single tile mode:

1. Create an HTML file and add the OpenLayers dependencies.
2. Now, we are going to create two maps side by side, each one with a WMS layer, as follows:

```
<table style="width: 100%; height: 95%;">
    <tr>
        <td style="width: 50%;">
            <p>WMS layer:</p>
            <div id="ch02_wms_non_singleTile" style="width: 100%;
            height: 100%;"></div>
        </td>
        <td style="width: 50%;">
            <p>WMS using <em>singleTile</em> property:</p>
            <div id="ch02_wms_singleTile" style="width: 100%;
            height: 100%;"></div>
        </td>
    </tr>
</table>
```

3. Next, write the required JavaScript code to initialize both maps. The first one will contain a normal WMS layer, as follows :

```
<script type="text/javascript">

    // Create the map using the specified DOM element
    var map_a = new OpenLayers.Map("ch02_wms_non_singleTile");

    // Add a WMS layer
    var wms = new OpenLayers.Layer.WMS("Basic",
    "http://vmap0.tiles.osgeo.org/wms/vmap0",
    {
        layers: 'basic'
    });
    map_a.addLayer(wms);

    // Set the center of the view
    map_a.setCenter(new OpenLayers.LonLat(-90,0), 2);
```

4. The second map will contain a WMS layer pointing to the same server but working in single tile mode, as follows:

```
    // Create the map using the specified DOM element
    var map_b = new OpenLayers.Map("ch02_wms_singleTile");

    // Add a WMS layer
    var wms = new OpenLayers.Layer.WMS("Basic",
```

```
        "http:// vmap0.tiles.osgeo.org/wms/vmap0",
        {
            layers: 'basic'
        },
        {
            singleTile: true
        });
        map_b.addLayer(wms);

        // Set the center of the view
        map_b.setCenter(new OpenLayers.LonLat(-90,0), 2);
    </script>
```

How it works...

The recipe code is pretty easy. The long explanation at the beginning, on how WMS layers can work in single tile mode, is simply achieved in practice using the `singleTile` property of the `OpenLayers.Layer` class. Pan or zoom the maps to see how differently it works.

See also

- The *Buffering the layer data to improve the map navigation* recipe
- The *Adding WMS layer* recipe
- The *Changing the layer opacity* recipe

Buffering the layer data to improve the map navigation

Map navigation is an important factor to take into account to make the user experience better.

When we pan the map, many times we get to see blank areas (meaning that the content is loading) and after a few seconds the image appears.

On gridded layers and WMS layers working in single tile mode, we can improve this at the cost of increasing the requests number or increasing the computation time at the server side.

Most of the raster layers inherit from the base class `OpenLayers.Layer.Grid`, which is responsible for dividing each zoom level into tiles.

For WMS layers working in single tile mode, the grid is formed only by one tile, which fills the whole map view.

The idea behind improving map navigation is simple; load the tiles outside the map view so that they are loaded before the user pans the map view in that direction.

This recipe shows you how to preload content outside the map view, both for gridded layers and also for WMS layers working in single tile mode, so that you can improve the navigation experience of the users.

How to do it...

1. Create an HTML file and include the OpenLayers dependencies.
2. We are going to create two maps side by side and on top of each one we are going to add a spinner control, from the Dojo Toolkit framework (`http://dojotoolkit.org`), to control the properties `buffer` and `singleTile` values:

```
<table style="width: 100%; height: 95%;">
    <tr>
        <td style="width: 50%;">
            <p>
                WMS layer with <em>buffer</em>:
                <input id="buffer_a"
                dojoType="dijit.form.NumberSpinner"
                onChange="changeBufferA"

intermediateChanges="true"
style="width:100px"
value="0" smallDelta="1"
constraints="{min:0,max:5}" />
            </p>
            <div id="ch02_wms_buffer" style="width: 100%;
            height: 100%;"></div>
        </td>
        <td style="width: 50%;">
            <p>
                WMS using <em>singleTile</em>
                property and <em>ratio</em>:
                <input id="buffer_b"
                dojoType="dijit.form.NumberSpinner"
                onChange="changeBufferB"

intermediateChanges="true"
style="width:100px"
value="1.0" smallDelta="0.1"
constraints="{min:0.0,max:2.0}" />
            </p>
```

```
            <div id="ch02_wms_ratio" style="width: 100%;
            height: 100%;"></div>
        </td>
    </tr>
</table>
```

3. The left-hand side panel will show how to control the number of tiles that can be loaded outside the map view:

```
<script type="text/javascript">
    // Create the map using the specified DOM element
    var map_a = new OpenLayers.Map("ch02_wms_buffer");

    // Add a WMS layer
    var wms_a = new OpenLayers.Layer.WMS("Basic",
    "http://vmap0.tiles.osgeo.org/wms/vmap0",
    {
        layers: 'basic'
    },
    {
        buffer: 0
    });
    map_a.addLayer(wms_a);

    // Set the center of the view
    map_a.setCenter(new OpenLayers.LonLat(-90,0), 3);
```

4. The right-hand side panel shows how to control the amount of data you can preload in a WMS layer working in single tile mode.

```
    // Create the map using the specified DOM element
    var map_b = new OpenLayers.Map("ch02_wms_ratio");

    // Add a WMS layer
    var wms_b = new OpenLayers.Layer.WMS("Basic",
    "http://vmap0.tiles.osgeo.org/wms/vmap0",
    {
        layers: 'basic'
    },
    {
        singleTile: true,
        ratio: 1
    });
    map_b.addLayer(wms_b);

    // Set the center of the view
    map_b.setCenter(new OpenLayers.LonLat(-90,0), 3);
```

5. Finally, there is the code responsible for changes on the spinner controls, shown as follows:

```
    // Handle events
    function changeBufferA(value) {
        wms_a.addOptions({buffer: value});
    }
    function changeBufferB(value) {
        map_b.removeLayer(wms_b);
        wms_b.destroy();
        wms_b = new OpenLayers.Layer.WMS("Basic",
        "http:// vmap0.tiles.osgeo.org/wms/vmap0",
        {
            layers: 'basic'
        },
        {
            singleTile: true,
            ratio: value
        });
        map_b.addLayer(wms_b);
    }
</script>
```

How it works...

The left-hand side map contains a WMS layer working in the default tiled mode. In this mode, the `buffer` property from the base class `OpenLayers.Layer.Grid` specifies how many tiles must be loaded outside the map view.

When a user changes the spinner value for the `buffer` property, we simply update it with the following line of code:

```
function changeBufferA(value) {
    wms_a.addOptions({buffer: value});
}
```

The right-hand side map, on the other hand, has a WMS layer working in single tile mode (see the `singleTile` property set to `true`). In this mode, only one request is made to get an image, which fills the whole map view.

We can control the size of the image with the `ratio` property, which belongs to the `OpenLayers.Layer.WMS` class. A ratio of value 1.0 means an image with exact dimensions of the map view. By default the ratio value is 1.5, which means we are requesting an image with the map view dimensions plus a half.

In this case, the ratio value is set once while creating the layer and to update it we need to delete the previous layer and create a new one with the new value. This is done as follows:

```
function changeBufferB(value) {
    map_b.removeLayer(wms_b);
    wms_b.destroy();
    wms_b = new OpenLayers.Layer.WMS("Basic",
    "http://vmap0.tiles.osgeo.org/wms/vmap0",
    {
        layers: 'basic'
    },
    {
        singleTile: true,
        ratio: value
    });
    map_b.addLayer(wms_b);
}
```

We first remove the layer from the map and later invoke the `destroy()` method to free internal resources used by the layer and avoid memory leaks.

There's more...

Remember, the more tiles we load the more requests to the server. The same goes for a WMS layer in single tile mode; the greater the bounding box you request, the greater the computation time on the server results.

Because of this, increasing the `buffer` or `ratio` values too much is not always the best solution.

Think about your data and how the user will explore it. If your data is probably better to explore in its extension—a great area in the same zoom level—then a buffer of one or two can be a good idea. If your data is mainly zoomed but the user is not interested in exploring large areas, then the default values are fine.

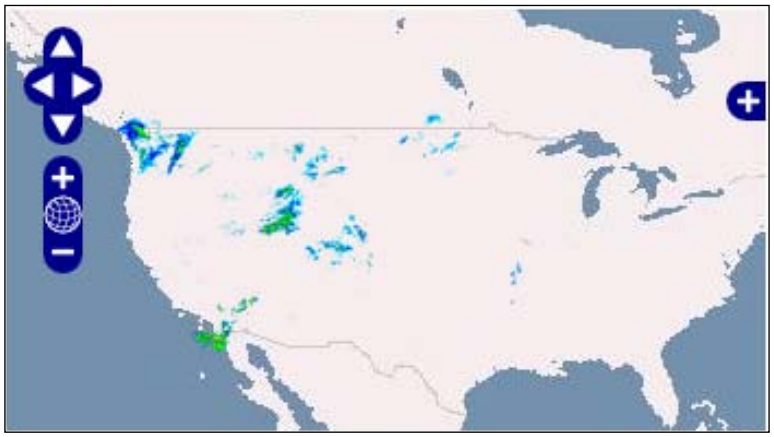
See also

- The *Using WMS with single tile mode* recipe
- The *Setting the tile size in WMS layers* recipe
- The *Adding WMS layer* recipe

Creating an image layer

Sometimes a tiled layer, such as Google Maps, OpenStreetMap, or WMS, is not what you need. It is quite possible that you have a georeferenced image, knowing its projection and bounding box, and want to render it on the map.

In these cases, OpenLayers offers the `OpenLayers.Layer.Image` class that allows us to create a layer based on a simple image. A georeferenced image is shown in the following screenshot:



How to do it...

To create an image layer, perform the following steps:

1. Let's go and create an HTML file with the OpenLayers dependencies.
2. First, add the `div` element that will hold the map, as follows:

```
<!-- Map DOM element -->
<div id="ch2_image" style="width: 100%; height: 100%;"></div>
```

3. Next, initialize the map and add a WMS base layer, as follows:

```
<!-- The magic comes here -->
<script type="text/javascript">
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch2_image", {
        allOverlays: true
    });
    map.addControl(new OpenLayers.Control.LayerSwitcher());

    // Add WMs layer
```

```
    var wms = new OpenLayers.Layer.WMS("OpenLayers WMS Basic",
    "http://vmap0.tiles.osgeo.org/wms/vmap0",
    {
        layers: 'basic'
    });
    map.addLayer(wms);
```

4. Now, define the image URL, its extent and size, and create an image layer as follows:

```
    // Add an Image layer
    var img_url =
    "http://localhost:8080/openlayers-cookbook/data/nexrad.png";
    var img_extent = new OpenLayers.Bounds(-131.0888671875,
    30.5419921875, -78.3544921875, 53.7451171875);
    var img_size = new OpenLayers.Size(780, 480);

    var image = new OpenLayers.Layer.Image("Image Layer", img_url,
    img_extent, img_size, {
        isBaseLayer: false,
        alwaysInRange: true // Necessary to always draw the image
    });
    map.addLayer(image);

    // Center the view
    map.setCenter(new OpenLayers.LonLat(-85, 40), 3);
</script>
```

How it works...

The `OpenLayers.Layer.Image` class constructor needs five parameters, as follows:

- **name**: This is the desired descriptive name for the layer
- **url**: This is the URL for the image
- **extent**: This is an instance of the `OpenLayers.Bounds` class with the bounding box of the image
- **size**: This is an instance of the `OpenLayers.Size` with the image dimensions in pixels
- **options**: This indicates a JavaScript object with different options for the layer

The image used in this recipe was previously obtained from NEXRAD (see `http://mesonet.agron.iastate.edu/current/mcview.phtml`) so we know the exact coordinates of their bounding box. They are:

```
var img_extent = new OpenLayers.Bounds(-131.0888671875, 30.5419921875,
-78.3544921875, 53.7451171875);
```

It is important to note that the bounds must be expressed in the same projection as the map, in this case EPSG:4326.

We also know the image size in pixels:

```
var img_size = new OpenLayers.Size(780, 480);
```

Given the image extent and size, OpenLayers computes the appropriate resolutions (think of it as zoom levels) where the image must be shown.

In this case, we always want to show the image on the map, no matter at which zoom level we are, and because of this we have used the `alwaysInRange` property set to `true`.

See also

- The *Adding WMS layer* recipe
- The *Using WMS with single tile mode* recipe
- The *Buffering the layer data to improve the map navigation* recipe

Setting the tile size in WMS layers

The `OpenLayers.Layer.Grid` class is a special kind of layer, which divides the layer in different zoom levels composed of a grid of tiles.

The `OpenLayers.Layer.WMS` class is a subclass of the preceding one and, in addition to working in single tile mode, it can work in tiled mode as well.

Of course, controlling the size of the tiles of the WMS request can affect the performance. By default, the tile size is 256 x 256 pixels, but we can set this to any desired value. Bigger tile sizes means less request to the server but more computation time to generate a bigger image. On the contrary, smaller tile sizes means more server requests and less time to compute smaller images.

How to do it...

To set the tile size, perform the following steps:

1. Create an HTML file with OpenLayers library dependency.
2. Add a `div` element that will hold the map, as follows:

```
<!-- Map DOM element -->
<div id="ch2_tilesize" style="width: 100%; height: 100%;"></div>
```

3. Next, initialize the map and add two layers, as follows:

```
<!-- The magic comes here -->
<script type="text/javascript">
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch2_tilesize", {
        allOverlays: true,
        tileSize: new OpenLayers.Size(256, 256)
    });
    map.addControl(new OpenLayers.Control.LayerSwitcher());

    // Add WMs layer
    var wms1 = new OpenLayers.Layer.WMS("OpenLayers WMS Basic",
    "http://vmap0.tiles.osgeo.org/wms/vmap0",
    {
        layers: 'basic'
    });
    map.addLayer(wms1);
```

4. For the second layer, specify the size of the tiles as follows:

```
    var wms2 = new OpenLayers.Layer.WMS("Coast Line",
    "http://vmap0.tiles.osgeo.org/wms/vmap0",
    {
        layers: 'coastline_01,coastline_02'
    },
    {
        tileSize: new OpenLayers.Size(512, 512),
        opacity: 0.65
    });
    map.addLayer(wms2);

    // Center the view
    map.setCenter(new OpenLayers.LonLat(-85, 40), 3);
</script>
```

How it works...

There is not much mystery in this recipe. The `tileSize` property is available both for `OpenLayers.Map` and `OpenLayers.Layer.Grid` subclasses.

The `tileSize` must be an instance of `OpenLayers.Size` class, indicating the width and height in pixels.

When the tile size is set in the map instance all layers use this value unless you specify another value for each individual layer.

By default, the OpenLayers.Map instance is configured to use 256 x 256 size tiles. Because of this, the first layer makes requests to the WMS server using a tile size of 256 x 256 pixels.

On the other hand, we have specified a 512 x 512 tile size value for the second layer, so the requests against the WMS are made waiting for tiles with 512 x 512 size.

There's more...

For tiled services, such as Google Maps or OpenStreetMap, the `tileSize` property is simply ignored because these services have precomputed the images in a fixed 256 x 256 size.

The reason for the tile size value being 256 x 256 pixels is because the size (in bytes) of each image file is optimum for bandwidth use.

See also

- The *Using WMS with single tile mode* recipe
- The *Buffering the layer data to improve the map navigation* recipe

3
Working with Vector Layers

In this chapter we cover:

- Adding a GML layer
- Adding KML layer
- Creating features programmatically
- Reading and creating features from a WKT
- Adding markers to the map
- Using point features as markers
- Working with popups
- Adding features from a WFS server
- Using the cluster strategy
- Filtering features in WFS requests
- Reading features directly using Protocols

Introduction

This chapter talks about vector layers. In addition to raster, vector information is the other important type of information we can work with in a GIS system.

The chapter tries to summarize the most common and important recipes you may need to work with in OpenLayers.

In GIS, a real-world phenomenon is represented by the concept of a feature. It can be a place—like a city or a village—it can be a road or a railway, it can be a region, a lake, the border of a country, or something similar.

Every feature has a set of attributes: population, length, and so on. It is represented visually by a geometrical symbol: point, line, polygon, and so on, using some visual style: color, radius, width, and so on.

As you can see, there are many concepts to take into account when working with vector information. Fortunately, OpenLayers provides us classes to work with them. We will learn more about these in this chapter.

The base class for vector layers is `OpenLayers.Layer.Vector` class, which defines the common properties and behavior for all the subclasses.

The `OpenLayers.Layer.Vector` class contains a set of features. These features are instances of the `OpenLayers.Feature.Vector` subclasses (which, in fact, are inherited from a more generic `OpenLayers.Feature` class).

Each feature has an `attributes` property and an `OpenLayers.Geometry` class instance associated with it.

The vector layer itself or each feature can have a visual style associated with it, which will be used to render the feature on the map.

In addition to the representation on the screen, we need to take into account the data source. OpenLayers offers classes to read/write features from/to many sources, or protocols, and using different formats: GML, KML, GeoJSON, GeoRSS, and so on.

The vector layer has optionally associated an instance of the `OpenLayers.Protocol` class and a list of instances of the `OpenLayers.Strategy` class. The first is responsible to read/write data using some protocol, such as HTTP or WFS, while the second (the strategy) is responsible to control tasks such as when to load or refresh the data in the layer: only once, every time the layer is moved, every few seconds, and so on.

Let's get started and see these classes in action.

Adding a GML layer

The **Geography Markup Language (GML)** is an XML grammar used to express geographic features. It is an OGC standard and is very well accepted by the GIS community.

In this recipe, we will show you how to create a vector layer from a GML file.

You can find the necessary files in the GML format attached to the source code of this book on the Packt Publishing website.

How to do it...

1. Create an HTML file with the required OpenLayers dependencies and insert the following code. First add the `div` element to hold the map:

```
<!-- Map DOM element -->
<div id="ch3_gml" style="width: 100%;
    height: 100%;"></div>
```

2. Next, add the JavaScript code to initialize the map, add a base layer, and a layer switcher control:

```
<!-- The magic comes here -->
<script type="text/javascript">
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch3_gml");

    var layer = new
        OpenLayers.Layer.OSM("OpenStreetMap");
    map.addLayer(layer);

    map.addControl(new
        OpenLayers.Control.LayerSwitcher());
    map.setCenter(new OpenLayers.LonLat(0,0), 2);
```

3. Finally, add a vector layer with the GML data:

```
        map.addLayer(new OpenLayers.Layer.Vector("Europe (GML)", {
            protocol: new OpenLayers.Protocol.HTTP({
                url: "http://localhost:8080/
                    openlayers-cookbook/recipes/
                    data/europe.gml",
                format: new OpenLayers.Format.GML()
            }),
            strategies: [new OpenLayers.Strategy.Fixed()]
        }));
    </script>
```

How it works...

Before using the `OpenLayers.Layer.Vector` class, we need to take some aspects into consideration.

If we need to load data from some source then we need to set a protocol and a strategy. In this case, we have used a fixed strategy, through the `OpenLayers.Strategy.Fixed` class instance, which means the data content is loaded only once. It is never refreshed or loaded again.

```
new OpenLayers.Layer.Vector("Europe (GML)", {
        protocol: new OpenLayers.Protocol.HTTP({
            url: "http://localhost:8080/openlayers-cookbook/recipes/
                data/europe.gml",
            format: new OpenLayers.Format.GML()
        }),
        strategies: [new OpenLayers.Strategy.Fixed()]
    })
```

The data to be loaded is accessible via the HTTP protocol and a URL to the file. The protocol, as an instance of the `OpenLayers.Protocol.HTTP` class, is responsible to read the data from the specified resource and requires a URL and a format to know how to read the data.

OpenLayers offers many format classes to read/write data, but in this recipe we have made use of an `OpenLayer.Format.GML` instance because our data source is a GML file.

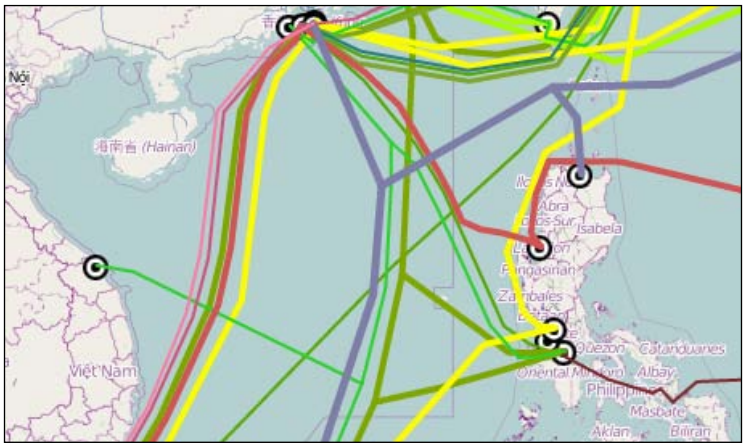
See also

- The *Adding a KML layer* recipe
- The *Creating features programmatically* recipe

Adding a KML layer

The arrival of Google Maps leads to an explosion in the world of GIS and web mapping. Google introduced not only an API but also some file formats.

The **Keyhole Markup Language (KML)** had become one of the most extensively used formats, and finally it became an OGC standard.



This recipe will show you how easy it is to add features from a KML file. You can find the necessary files in the KML format attached to the source code of this book available on the Packt Publishing website.

How to do it...

1. Create an HTML file including the OpenLayers library and insert the following code in it. First, add the `DOM` element that will hold the map:

```
<!-- Map DOM element -->
<div id="ch3_kml" style="width: 100%; height: 100%;"></div>
```

2. Next, initialize a map instance, add a base layer, add a layer switcher control, and center the view:

```
<!-- The magic comes here -->
<script type="text/javascript">

    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch3_kml");

    var layer = new OpenLayers.Layer.OSM("OpenStreetMap");
    layer.wrapDateLine = false;
```

```
        map.addLayer(layer);

        map.addControl(new OpenLayers.Control.LayerSwitcher());
        map.setCenter(new OpenLayers.LonLat(0,0), 2);
```

3. Finally, add a vector layer that will load data from a KML file:

```
        // Global Undersea Fiber Cables
        map.addLayer(new OpenLayers.Layer.Vector("Global
            Undersea Fiber Cables", {
            protocol: new OpenLayers.Protocol.HTTP({
                url: "http://localhost:8080/
                openlayers-cookbook/recipes/
                data/global_undersea.kml",
                format: new OpenLayers.Format.KML({
                    extractStyles: true,
                    extractAttributes: true
                })
            }),
            strategies: [new OpenLayers.Strategy.Fixed()]
        }));
    </script>
```

How it works...

After initializing the map, we centered the view and added some controls. Then we added a vector layer.

Because we want to load data from a KML file, that is accessible via HTTP protocol, we have set an `OpenLayers.Protocol.HTTP` instance as the protocol of the vector layer. It uses the URL of the file and uses an `OpenLayers.Format.KML` instance as the `format` property.

In addition, we have set an `OpenLayers.Strategy.Fixed` instance as the strategy of the vector layer, which makes the file load only once.

In addition, we have used a couple of `OpenLayers.Format.KML` classes, `extractStyles` and `extractAttributes`, to maintain the color styles and attributes specified in the source KML file. Otherwise, OpenLayers will apply a default style.

There's more...

The KML format, like GML, offers tons of options and possibilities at the cost of complexity.

In the KML format, placemarks can have a description attached to them and, if you load a KML file in Google Maps, the placemark's description is shown as a balloon (or popup) when you click on them.

In OpenLayers, this approach differs a bit. As we will see in the *Working with popups* recipe, the process to load the KML data and the behavior to show them are completely different. So don't expect the vector layer that loads the data to also attach the required code to control the click event, show the popup, and so on. That is our work.

See also

- The *Adding a GML layer* recipe
- The *Creating features programmatically* recipe
- The *Working with popups* recipe

Creating features programmatically

Loading data from an external source is not the only way to work with vector layers.

Imagine a web mapping application where the user can create new features on the fly: cities, rivers, areas of interest, and so on, and add them to a vector layer with some style. This scenario requires the ability to create and add the features programmatically.

In this recipe we will see some ways to create and manage features programmatically.

How to do it...

1. Start by creating a new HTML file with the required OpenLayers dependencies. Add the `div` element to hold the map:

```
<!-- Map DOM element -->
<div id="ch3_features_programmatically"
    style="width: 100%; height: 100%;"></div>
```

2. Next, initialize the map instance and add a base layer:

```
<!-- The magic comes here -->
<script type="text/javascript">

    // Create the map using the specified DOM element
    var map = new
        OpenLayers.Map("ch3_features_programmatically");

    // Add a WMS layer
    var wms = new OpenLayers.Layer.WMS("Basic",
        "http://vmap0.tiles.osgeo.org/wms/vmap0",
    {
        layers: 'basic'
    });
```

```
map.addLayer(wms);

map.addControl(new
    OpenLayers.Control.LayerSwitcher());
map.setCenter(new OpenLayers.LonLat(0,0), 2);
```

3. Now, create three vector layers to put three different types of features:

```
// Create some empty vector layers
var pointLayer = new
    OpenLayers.Layer.Vector("Points");
var lineLayer = new OpenLayers.Layer.Vector("Lines");
var polygonLayer = new
    OpenLayers.Layer.Vector("Polygon");

// Add layers to the map
map.addLayers([polygonLayer, lineLayer, pointLayer]);
```

4. Call the function that will create the point, line, and polygon features and add them to each of the previous layers:

```
// Fill layers
initializePointLayer();
initializeLineLayer();
initializePolygonLayer();

// Create some random points.
function initializePointLayer() {
    var pointFeatures = [];
    for(var i=0; i< 50; i++) {
        var px = Math.random()*360-180;
        var py = Math.random()*180-90;

        var pointGeometry = new
            OpenLayers.Geometry.Point(px, py);
        var pointFeature = new
            OpenLayers.Feature.Vector(pointGeometry);
        pointFeatures.push(pointFeature);
    }
    pointLayer.addFeatures(pointFeatures);
}

// Create some random lines
function initializeLineLayer() {
    for(var j=0; j< 2; j++) {
        var pointGeometries = [];
        for(var i=0; i< 10; i++) {
```

```
                var px = Math.random()*240-120;
                var py = Math.random()*100-50;

                var pointGeometry = new
                    OpenLayers.Geometry.Point(px, py);
                pointGeometries.push(pointGeometry);
            }
            var lineGeometry = new OpenLayers.Geometry.
                LineString(pointGeometries);
            var lineFeature = new
                OpenLayers.Feature.Vector(lineGeometry);
            lineLayer.addFeatures(lineFeature);
        }
    }

    // Create some random polygons
    function initializePolygonLayer() {
        for(var j=0; j< 2; j++) {
            var pointGeometries = [];
            for(var i=0; i< 5; i++) {
                var px = Math.random()*240-180;
                var py = Math.random()*100-90;

                var pointGeometry = new
                    OpenLayers.Geometry.Point(px, py);
                pointGeometries.push(pointGeometry);
            }
            var linearGeometry = new OpenLayers.Geometry.
                LinearRing(pointGeometries);
            var polygonGeometry = new OpenLayers.
                Geometry.Polygon([linearGeometry]);
            var polygonFeature = new OpenLayers.
                Feature.Vector(polygonGeometry);
            polygonLayer.addFeatures(polygonFeature);
        }
    }
</script>
```

How it works...

As described in the chapter's introduction, a vector layer contains a set of features. Each feature represents some phenomenon of the real world and has a geometry and a style associated with it, which will determine the visual representation.

Let's start looking at the code responsible for creating random points:

```
var pointFeatures = [];
for(var i=0; i< 50; i++) {
    var px = Math.random()*360-180;
    var py = Math.random()*180-90;

    var pointGeometry = new
        OpenLayers.Geometry.Point(px, py);
    var pointFeature = new
        OpenLayers.Feature.Vector(pointGeometry);
    pointFeatures.push(pointFeature);
}
pointLayer.addFeatures(pointFeatures);
```

In this case, each feature is represented by a point geometry, because we first need to create an `OpenLayers.Geometry.Point` instance with the coordinates of the point.

Remember to express the coordinates in the appropriate projection, the one used by the map, or set the right projection in the vector layer so that OpenLayers can translate the coordinates.

Once we have the geometry instance, we can create a new `OpenLayers.Feature.Vector` instance by passing the desired geometry instance to be used by the feature.

Note that we will cover working with feature styles in another chapter. It will be rendered with a default OpenLayers style.

All the features are stored in an array and passed at once to the vector layer using the `addFeatures()` method.

Next in the difficulty order is the creation of lines, named in the geometry objects terminology as LineStrings. When you want to represent a feature as a LineString you need to use an instance of the geometry class `OpenLayers.Geometry.LineString`. As we can see in the following block of code, the line string constructor needs an array of the `OpenLayers.Geometry.Point` instance that conforms the set of points for the lines.

```
var pointGeometries = [];
for(var i=0; i< 10; i++) {
    var px = Math.random()*240-120;
    var py = Math.random()*100-50;

    var pointGeometry = new
        OpenLayers.Geometry.Point(px, py);
    pointGeometries.push(pointGeometry);
}
```

```
var lineGeometry = new
    OpenLayers.Geometry.LineString(pointGeometries);
var lineFeature = new
    OpenLayers.Feature.Vector(lineGeometry);
lineLayer.addFeatures(lineFeature);
```

The OGC's Simple Feature Access specification (`http://www.opengeospatial.org/standards/sfa`) contains an in-depth description of the standard. It also contains an UML class diagram where you can see all the geometry classes and hierarchy.

Finally, we found the code that creates some polygons.

Polygons are great geometries to represent states or countries. We can think of polygons as a simple set of lines where the start and end point is the same, a so called *LineRing*, and filled with some color. But be aware, polygons can be very complex structures that complicate the way we must express them.

For example, think of a region with a hole in it. In this case we have two line rings to describe the external and internal perimeters. We must also specify which part must be colored.

Take a look at the following code:

```
var pointGeometries = [];
for(var i=0; i< 5; i++) {
    var px = Math.random()*240-180;
    var py = Math.random()*100-90;

    var pointGeometry = new
        OpenLayers.Geometry.Point(px, py);
    pointGeometries.push(pointGeometry);
}
var linearGeometry = new
    OpenLayers.Geometry.LinearRing(pointGeometries);
var polygonGeometry = new
    OpenLayers.Geometry.Polygon([linearGeometry]);
var polygonFeature = new
    OpenLayers.Feature.Vector(polygonGeometry);
polygonLayer.addFeatures(polygonFeature);
```

Here we create an `OpenLayers.Geometry.LineRing` instance by passing an array of `OpenLayers.Geometry.Point` with the set of points that conforms the line ring.

Once we have one or more line rings, we can create a new instance of the `OpenLayers.Geometry.Polygon` class, which will be used to render our new vector layer feature.

See also

- The *Adding marker to the map* recipe
- The *Reading and creating features from WKT* recipe
- The *Working with popups* recipe
- The *Styling features using symbolizers* recipe in *Chapter 7*, *Styling Features*

Reading and creating features from a WKT

OpenLayers comes with a great set of format classes, which are used to read/write from/to different file data formats. GeoJSON, GML, or GPX are some of the many formats we can find.

If you have read the *Adding a GML layer* recipe in this chapter, you will know that a vector class can read the features stored in a file, specify the format of the data source, and place the contained features in the map.

This recipe wants to show us exactly that. We will see the magic step responsible to read data from a file using a format class, and transform it to the corresponding feature ready to be placed in the layer.

For simplicity, we will only see how to read features from the WKT text. You can learn more about WKT (Well-Known Text) format from `http://en.wikipedia.org/wiki/Well-known_text`.

As can be seen in the previous screenshot, we are going to create a map on the left side, and on the right we will place a couple of text area components to add and get features in the WKT format.

How to do it...

1. Create a new HTML file with OpenLayers dependencies. Then, add the following HTML code for the map, text area, and buttons:

```
<!-- Map DOM element -->
<table style="width: 100%; height: 95%;">
    <tr>
        <td>
            <div id="ch3_reading_wkt" style="width: 100%;
                height: 100%;"></div>
        </td>
        <td style="width: 30%; vertical-align: top;">
            <p>Write the WKT describing features:</p>
            <textarea id="wktText"
                dojoType="dijit.form.SimpleTextarea"
                rows="10" style="width: 100%;">
                MULTIPOLYGON
                (((40 40, 20 45, 45 30, 40 40)),
                ((20 35, 45 20, 30 5, 10 10, 10 30, 20 35),
                (30 20, 20 25, 20 15, 30 20)))</textarea>
            <button dojoType="dijit.form.Button"
                onClick="addFeature">Add Feature</button>
            <button dojoType="dijit.form.Button"
                onClick="clearLayer">Clear Layer</button>
        </td>
    </tr>
</table>
```

Remember, we are using Dojo toolkit framework (`http://dojotoolkit.org`) to improve our components, so some elements will have attributes like `dojoType="dijit.form.Button"`.

2. Now, we will initialize the map component and place a base layer:

```
<!-- The magic comes here -->
<script type="text/javascript">
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch3_reading_wkt");

    // Add a WMS layer
    var wms = new OpenLayers.Layer.WMS("Basic",
        "http://vmap0.tiles.osgeo.org/wms/vmap0",
    {
        layers: 'basic'
    });
```

```
map.addLayer(wms);

map.addControl(new
    OpenLayers.Control.LayerSwitcher());
map.setCenter(new OpenLayers.LonLat(0,0), 2);
```

3. Let's go on to create a vector layer to hold the features we will read from the WKT:

```
// Create some empty vector layers
var wktLayer = new
    OpenLayers.Layer.Vector("wktLayer");
// Add layers to the map
map.addLayer(wktLayer);
```

4. We need a couple of functions to handle the button events. The first function is responsible to clean the vector layer:

```
function clearLayer() {
    wktLayer.removeAllFeatures();
}
```

5. The second function reads the data from the WKT string and places the features on the vector layer:

```
function addFeature() {
    // Read features and add to the vector layer
    var text = dijit.byId('wktText').get('value');
    var wkt = new OpenLayers.Format.WKT();
    var features = wkt.read(text);
    wktLayer.addFeatures(features);

    // Dump the vector layer features to WKt format
    var currentWkt = wkt.write(wktLayer.features);
    dijit.byId('wktFeatures').set('value', currentWkt);
}
</script>
```

How it works...

All the format classes are inherited from the `OpenLayers.Format` base class, which defines the basic behavior of the format classes, that is, have a `read` and a `write` method.

The `read()` method is supposed to read data in some format (a JSON string, a WKT string, and so on) and return an array of features as instances of the `OpenLayers.Feature.Vector` class.

The `write()` method, on the other hand, receives an array of features and returns a string that represents the desired format.

Depending on the format subclass, the `read` and `write` methods can accept additional parameters. Always be careful and read the API documentation.

To read the features from a WKT string, we only need to instantiate the desired format class and call its `read` method by passing a valid string as the argument:

```
var wkt = new OpenLayers.Format.WKT();
var features = wkt.read(text);
wktLayer.addFeatures(features);
```

Then, we get the current features of the vector layer and convert them to a WKT representation by passing them to the `write` method:

```
// Dump the vector layer features to WKt format
var currentWkt = wkt.write(wktLayer.features);
dijit.byId('wktFeatures').set('value', currentWkt);
```

See also

- The *Adding a GML layer* recipe
- The *Creating features programmatically* recipe
- The *Reading features directly using Protocols* recipe

Adding markers to the map

Markers are widely used in web mapping applications. They allow us to quickly identify points of interest (POI) by showing an icon at the desired place.

This recipe shows how to add markers to our maps by using the `OpenLayers.Marker` and `OpenLayers.Layer.Markers` classes.

How to do it...

1. Start by creating an HTML page with dependencies on the OpenLayers library. Add the `div` element that will hold the map:

```
<!-- Map DOM element -->
<div id="ch3_markers" style="width: 100%; height: 100%;"></div>

Create the map instance, add a base layer and a layer switcher
control:
<!-- The magic comes here -->
<script type="text/javascript">

    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch3_markers");

    layer = new OpenLayers.Layer.OSM("OpenStreetMap");
    map.addLayer(layer);

    map.addControl(new
        OpenLayers.Control.LayerSwitcher());
    map.setCenter(new OpenLayers.LonLat(0,0), 3);
```

2. Now, add a new kind of layer, `OpenLayers.Layer.Markers`, specially designed to contain the `OpenLayers.Marker` instances:

```
    var markers = new
        OpenLayers.Layer.Markers("Markers");
    map.addLayer(markers);
```

3. We will now create markers at random places by using a random icon from an array:

```
    // Create some random markers with random icons
    var icons = [
         // Here goes an array of image file names
    ];

    for(var i=0; i< 150; i++) {
        // Compute a random icon and lon/lat position.
        var icon = Math.floor(Math.random() *
            icons.length);
        var px = Math.random() * 360 - 180;
        var py = Math.random() * 170 - 85;

        // Create size, pixel and icon instances
        var size = new OpenLayers.Size(32, 37);
```

```
            var offset = new OpenLayers.Pixel(-(size.w/2),
                -size.h);
            var icon = new OpenLayers.Icon('./recipes/data/
                icons/'+icons[icon], size, offset);
            icon.setOpacity(0.7);

            // Create a lonlat instance and transform it to
            // the map projection.
            var lonlat = new OpenLayers.LonLat(px, py);
            lonlat.transform(new
                OpenLayers.Projection("EPSG:4326"), new
                OpenLayers.Projection("EPSG:900913"));

            // Add the marker
            var marker = new OpenLayers.Marker(lonlat, icon);

            // Event to handler when the mouse is over
            // Inflate the icon and change its opacity
            marker.events.register("mouseover", marker,
                function() {
                console.log("Over the marker "+this.id+"
                    at place "+this.lonlat);
                this.inflate(1.2);
                this.setOpacity(1);
            });
            // Event to handler when the mouse is out
            // Inflate the icon and change its opacity
            marker.events.register("mouseout", marker,
                function() {
                console.log("Out the marker "+this.id+" at
                    place "+this.lonlat);
                this.inflate(1/1.2);
                this.setOpacity(0.7);
            });

            markers.addMarker(marker);
        }
    </script>
```

How it works...

The class `OpenLayers.Layer.Markers` is a direct subclass of the `OpenLayers.Layer` base class, and is specially designed to contain markers.

On the other hand, a marker is represented by instances of the class `OpenLayers.Layer.Markers`. Every marker has an associated point, expressed with an instance of the `OpenLayers.LonLat` class, and an icon using an instance of `OpenLayers.Icon`.

An icon requires a *URL* of the image to be loaded, a *size* expressed as an instance of `OpenLayers.Size`, and an *offset* expressed as an instance of `OpenLayers.Pixel`.

In addition, for each marker we have registered two listeners, one to know when the mouse is over and one to know when it leaves the marker. In this way, we can modify the size and opacity of the marker to highlight when the mouse has selected or deselected it.

Inside the handler functions, we have made use of the methods `inflate()`, to change the size of the icon augmenting its proportions, and `setOpacity()`, to change the icon opacity:

```
marker.events.register("mouseover", marker, function() {
    console.log("Over the marker "+this.id+"
        at place "+this.lonlat);
    this.inflate(1.2);
    this.setOpacity(1);
});
```

For beginners in JavaScript, remember the object that calls the anonymous function that handles the marker event is the marker itself. Because the `this` keyword is referencing the marker with which we can call the `inflate()` or `setOpacity()` methods.

There's more...

The use of markers through the `OpenLayers.Marker` and `OpenLayers.Layer.Markers` classes is not the only way we can show POIs in our maps.

As you can see in the *Using point features as markers* recipe, we can also use features to show POIs as an alternative that can be improved by the use of strategies, formats, and so on.

In addition, OpenLayers offers some classes, such as `OpenLayers.Layer.GeoRSS` or `OpenLayers.Layer.Text`, that create markers automatically from the GeoRSS and CSV files respectively. They are relatively simple and are implemented for a specific usage and, most probably, you will soon need more flexibility than offered by those classes.

See also

- The *Using point features as markers* recipe
- The *Creating features programmatically* recipe
- The *Reading features directly using Protocol* recipe

Using point features as markers

Displaying markers is not only limited to using the `OpenLayers.Marker` and `OpenLayers.Layer.Markers` classes.

A marker can be understood as a point of interest (POI) where we place an icon to identify it and has some information associated with it: a monument, a parking area, a bridge, and so on.

In this recipe, we will learn how to use these features with a point geometry type associated to create markers.

How to do it...

1. Once you have created the right HTML file with OpenLayers dependencies, add a `div` element to hold the map:

```
<div id="ch3_feature_markers" style="width: 100%;
height: 100%;"></div>
```

2. Start initializing the map instance and add a base layer and control:

```
// Create the map using the specified DOM element
var map = new OpenLayers.Map("ch3_feature_markers");

var layer = new
    OpenLayers.Layer.OSM("OpenStreetMap");
map.addLayer(layer);

map.addControl(new
    OpenLayers.Control.LayerSwitcher());
map.setCenter(new OpenLayers.LonLat(0,0), 2);
```

3. Next, add a vector layer that will contain a set of random markers:

```
var pointLayer = new
    OpenLayers.Layer.Vector("Features", {
    projection: "EPSG:933913"
});
map.addLayer(pointLayer);
```

4. Create some random points. To improve the performance we add all the points to an array and then to the vector layer all at once with the `addFeatures` method:

```
// Create some random feature points
var pointFeatures = [];
for(var i=0; i< 150; i++) {
    var px = Math.random() * 360 - 180;
    var py = Math.random() * 170 - 85;

    // Create a lonlat instance and transform it
    // to the map projection.
    var lonlat = new OpenLayers.LonLat(px, py);
    lonlat.transform(new
        OpenLayers.Projection("EPSG:4326"),
        new OpenLayers.Projection("EPSG:900913"));

    var pointGeometry = new
        OpenLayers.Geometry.Point
        (lonlat.lon, lonlat.lat);
    var pointFeature = new
        OpenLayers.Feature.Vector(pointGeometry);

    pointFeatures.push(pointFeature);
}
// Add features to the layer
pointLayer.addFeatures(pointFeatures);
```

5. Now, attach two event listeners to the vector layer for the `featureselected` and `featureunselected` events. The listener will be responsible for changing the feature style:

```
// Event handler for feature selected
pointLayer.events.register("featureselected", null,
    function(event){
    var layer = event.feature.layer;
    event.feature.style = {
        fillColor: '#ff9900',
        fillOpacity: 0.7,
```

```
            strokeColor: '#aaa',
            pointRadius: 12
        };
        layer.drawFeature(event.feature);
    });
    // Event handler for feature unselected
    pointLayer.events.register("featureunselected", null,
        function(event){
        var layer = event.feature.layer;
        event.feature.style = null;
        event.feature.renderIntent = null;
        layer.drawFeature(event.feature);
    });
```

6. Finally, we need to attach a `SelectFeature` control to the map, and reference the vector layer:

```
    // Add select feature control required to trigger events on
the vector layer.
    var selectControl = new OpenLayers.Control.
SelectFeature(pointLayer);
    map.addControl(selectControl);
    selectControl.activate();
```

How it works...

The idea is simple, add point features to the layer and listen for their selection event to change the style.

In a different way than working with the `OpenLayers.Marker` instances, we need to attach listeners to the vector layer and not to the feature itself, using the following code:

```
pointLayer.events.register("featureselected", null,
    function(event){
    // Code here
});
```

Within the listener function, we can access the selected feature or the vector layer it belongs to with the `event` variable:

```
    var layer = event.feature.layer;
    event.feature.style = {
        fillColor: '#ff9900',
        fillOpacity: 0.7,
        strokeColor: '#aaa',
        pointRadius: 12
    };
```

In *Chapter 7*, *Styling Features*, we will learn more about styling features and improving its look using images, in a similar way to the `OpenLayers.Marker` class.

Once the feature style is changed, we can call `drawFeature()` on the vector layer to refresh the feature on the map:

```
layer.drawFeature(event.feature);
```

To allow the vector layer to trigger events, we need to attach a `SelectFeature` control to the map, reference the vector layer, and activate it. Without it the listeners will never be invoked.

```
var selectControl = new
    OpenLayers.Control.SelectFeature(pointLayer);
map.addControl(selectControl);
selectControl.activate();
```

See also

- The *Creating features programmatically* recipe
- The *Adding markers to the map* recipe
- The *Working with popups* recipe
- The *Using the cluster strategy* recipe

Working with popups

A common charecteristic of web mapping applications is the ability to show information related to the features the map contains. By feature we mean any real phenomenon or aspect we can visually represent with points, lines, polygons, and so on.

Of course we can select a feature, retrieve its associated information and show it anywhere in our application layout, but the most common way to show it is by using popups.

How to do it...

1. Create an HTML file with OpenLayers dependencies. Then add the `div` element to hold the map:

```
<div id="ch3_popups" style="width: 100%; height: 100%;">
</div>
```

2. Within the JavaScript section, initialize the map and add a base layer:

```
var map = new OpenLayers.Map("ch3_popups");
var layer = new
    OpenLayers.Layer.OSM("OpenStreetMap");
map.addLayer(layer);
map.addControl(new
    OpenLayers.Control.LayerSwitcher());
map.setCenter(new OpenLayers.LonLat(0,0), 2);
```

3. Create a vector layer and add some features to it:

```
var pointLayer = new
    OpenLayers.Layer.Vector("Features", {
    projection: "EPSG:900913"
});
map.addLayer(pointLayer);
```

4. Next, add some random features to the vector layer:

```
var pointFeatures = [];
for(var i=0; i< 150; i++) {
    var icon = Math.floor(Math.random() *
        icons.length);
    var px = Math.random() * 360 - 180;
    var py = Math.random() * 170 - 85;
```

```
        // Create a lonlat instance and transform it to
        // the map projection.
        var lonlat = new OpenLayers.LonLat(px, py);
        lonlat.transform(new
            OpenLayers.Projection("EPSG:4326"), new
            OpenLayers.Projection("EPSG:900913"));

        var pointGeometry = new
            OpenLayers.Geometry.Point(lonlat.lon,
            lonlat.lat);
        var pointFeature = new
            OpenLayers.Feature.Vector(pointGeometry,
            null, {
            pointRadius: 16,
            fillOpacity: 0.7,
            externalGraphic:
                'http://localhost:8080/
                openlayers-cookbook/recipes/data/
                icons/'+icons[icon]
        });

        pointFeatures.push(pointFeature);
    }
    // Add features to the layer
    pointLayer.addFeatures(pointFeatures);
```

You need to change the previous URL to the right address of your custom server.

5. Finally, add the code responsible to manage the feature selection to show the popup:

```
// Add select feature control required to trigger events on the
vector layer.
    var selectControl = new
        OpenLayers.Control.SelectFeature(pointLayer, {
        hover: true,
        onSelect: function(feature) {
            var layer = feature.layer;
            feature.style.fillOpacity = 1;
            feature.style.pointRadius = 20;
            layer.drawFeature(feature);

            var content = "<div><strong>Feature:</strong>
                <br/>" + feature.id +
                "<br/><br/><strong>Location:</strong>
```

```
                <br/>" + feature.geometry +"</div>";

            var popup = new OpenLayers.Popup.FramedCloud(
            feature.id+"_popup",
            feature.geometry.getBounds().
            getCenterLonLat(),
            new OpenLayers.Size(250, 100),
            content,
            null,
            false,
            null);
            feature.popup = popup;
            map.addPopup(popup);
        },
        onUnselect: function(feature) {
            var layer = feature.layer;
            feature.style.fillOpacity = 0.7;
            feature.style.pointRadius = 16;
            feature.renderIntent = null;
            layer.drawFeature(feature);

            map.removePopup(feature.popup);
        }
    });
    map.addControl(selectControl);
    selectControl.activate();
```

How it works...

The first thing we did, after creating the vector layer, was the creation of some random point features.

Because we are computing random latitude and longitude values in decimal degrees (`"EPSG:4326"` projection), we need to translate it to the projection used by the map. In this case, because `OpenStreetMap` is the base layer, it applies an `"EPSG:900913"` projection as the map's projection.

```
var lonlat = new OpenLayers.LonLat(px, py);
lonlat.transform(new OpenLayers.Projection("EPSG:4326"),
    new OpenLayers.Projection("EPSG:900913"));

var pointGeometry = new
    OpenLayers.Geometry.Point(lonlat.lon, lonlat.lat);
var pointFeature = new
    OpenLayers.Feature.Vector(pointGeometry, null, {
```

```
            pointRadius: 16,
            fillOpacity: 0.7,
            externalGraphic: 'http://localhost:8080/
                openlayers-cookbook/recipes/
                data/icons/'+icons[icon]
        });
```

Here we are creating features with a custom style. The constructor of the `OpenLayers.Feature.Vector` class accepts three parameters: a *geometry* parameter, which is mandatory, and two optional parameters, the feature *attributes* and the feature *style*.

Our features have no special attributes so we have passed a `null` value but, on the other hand, we have used a custom style to show an icon image instead of a simple point to represent them.

Once we have the features we want, it is time to show a popup with some nice description when a feature is selected.

To achieve this, we have used the `SelectFeature` control. Given a layer, this control allows the user to select features. We can customize the behavior of the control with the `options` argument:

```
var selectControl = new
    OpenLayers.Control.SelectFeature(pointLayer, {
    hover: true,
    onSelect: function(feature) { … },
    onUnselect: function(feature) { … }
});
```

In this recipe we have used the following three options:

- `hover`: It indicates that the features must be selected or unselected without the need of clicking on it, and by simply moving the mouse over the button.
- `onSelect`: This function is executed when a feature is selected. It receives the selected feature as an argument.
- `onUnselect`: This function is executed when a feature is unselected. It receives the unselected feature as an argument.

Now let's take a look at how to create the popups.

The important point to be noted here is that popups are added to the map. They are not added to a feature and nor to a layer. So to show or hide a popup, we simply need to add or remove it from the map with the methods `addPopup()` or `removePopup()`.

OpenLayers offers some classes to be used as popups, but all of them are inherited from the base class `OpenLayers.Popup`.

We have chosen the `OpenLayers.Popup.FramedCloud` subclass, which is a visually decent styled popup. The constructor requires the following parameters:

- `id`: A string that identifies the popup among all the popups that can exist, which are attached to the map
- `lonlat`: The location where the popup must appear
- `contentSize`: The dimensions of the popup, as an instance of the `OpenLayers.Size` class
- `contentHTML`: The HTML string to be put as content
- `anchor`: An object where the popup will be anchored
- `closeBox`: Boolean indicating if the close buttons must be shown
- `closeBoxCallback`: A function that will be executed when the user clicks on the close button

With all these parameters, our code to create a `FramedCloud` popup looks as follows:

```
var popup = new OpenLayers.Popup.FramedCloud(
feature.id+"_popup",
feature.geometry.getBounds().getCenterLonLat(),
new OpenLayers.Size(250, 100),
content,
null,
false,
null);
```

Once created, we add it to the map, which makes it visible automatically:

```
feature.popup = popup;
map.addPopup(popup);
```

We have also stored a reference of the popup within the feature. In this way, we can easily find a reference to the popup in the function that is executed when the feature is unselected and remove it from the map:

```
map.removePopup(feature.popup);
```

As a note, the map's `addPopup()` method has a second and optional parameter `exclusive`, which if set, automatically removes all existing popups in the map when a new one is added.

See also

- The *Adding markers to the map* recipe
- The *Using point features as markers* recipe

Adding features from a WFS server

The **Web Feature Service (WFS)** is an OGC standard, which provides independent platform calls to request geographical features to a server. In practice, it means a client makes a HTTP request to a server that implements the WFS standard and gets a set of features in the GML (Geographic Markup Language, `http://en.wikipedia.org/wiki/Geography_Markup_Language`) format.

A nice introduction to WFS can be found in the tutorial about WFS available at `https://www.e-education.psu.edu/geog585/book/export/html/1724`. If you want to learn more about this, there is a complete specification on the OGC site `http://www.opengeospatial.org/standards/wfs`.

From the OpenLayers point of view, the WFS is nothing more than another data source we can read to fill a vector layer.

Before continuing, there is an important point to take into account. Most of the requests made by OpenLayers when data is loaded, say GML, KML, or GeoRSS files, are made asynchronously through the helper class `OpenLayers.Request`.

Any JavaScript call is limited by the security model imposed by the browser, which avoids cross domain requests. This means you can only make requests to the same server that the web page originally came from.

There are different ways to avoid this fact, but a simple one is the use of a proxy on the server side.

You can read a clearer explanation at `http://developer.yahoo.com/javascript/howto-proxy.html`.

The idea of a proxy is simple, instead of making a request directly to a cross domain we make a request to a script on the same domain, which is responsible for making the cross domain request and returning the results.

A script, say PHP, Python, or Java servlet, is not limited by the cross domain requests. It is only security imposed by the browser in the JavaScript calls.

OpenLayers offers a proxy implementation as a Python script that we can use in our application. It can be found in the `examples/proxy.cgi` file in the source code bundle.

It is not the only possibility. For this recipe we will be using a PHP proxy file (see the `utils/proxy.php` file in the book's source code) from the `MapBuilder` project.

How to do it...

1. Create a HTML file, set the OpenLayers dependencies, and add a `div` element to hold the map:

```
<!-- Map DOM element -->
<div id="ch3_wfs" style="width: 100%;
    height: 100%;"></div>
```

2. Set the `OpenLayers.ProxyHost` variable to our proxy URL:

```
<!-- The magic comes here -->
<script type="text/javascript">
    OpenLayers.ProxyHost = "./utils/proxy.php?url=";
```

3. Initialize the map and add a base layer:

```
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch3_wfs");

    var baseLayer = new
        OpenLayers.Layer.OSM("OpenStreetMap");
    map.addLayer(baseLayer);

    map.addControl(new
        OpenLayers.Control.LayerSwitcher());
    map.setCenter(new OpenLayers.LonLat(0,0), 2);
```

4. Finally, create a vector layer that uses the WFS protocol to access the data source:

```
    var statesLayer = new
        OpenLayers.Layer.Vector("States", {
        protocol: new OpenLayers.Protocol.WFS({
            url: "http://demo.opengeo.org/geoserver/wfs",
            featureType: "states",
            featureNS: "http://www.openplans.org/topp"
        }),
        strategies: [new OpenLayers.Strategy.BBOX()]
    });
    map.addLayer(statesLayer);
</script>
```

How it works...

The first important step is to set the `OpenLayers.ProxyHost` variable:

```
OpenLayers.ProxyHost = "./utils/proxy.php?url=";
```

Most of the JavaScript requests in OpenLayers are made through the helper class `OpenLayers.Request`, which checks if the previous variable is set. If so, all requests are made using the proxy.

After that, the main action in this recipe is the creation of a vector layer filling its data from a WFS server:

```
var statesLayer = new OpenLayers.Layer.Vector("States", {
    protocol: new OpenLayers.Protocol.WFS({
        url: "http://demo.opengeo.org/geoserver/wfs",
        featureType: "states",
        featureNS: "http://www.openplans.org/topp"
    }),
    strategies: [new OpenLayers.Strategy.BBOX()]
});
```

As you can see, the only thing to do is set the protocol to be used by the layer. In this case, we use an instance of the `OpenLayers.Protocol.WFS` class.

The WFS protocol constructor has many parameters but the most important ones are as follows:

- `url`: The URL to the WFS server
- `featureType`: The feature to be queried
- `featureNS`: The namespace of the feature

Other important options and, more or less, the commonly used ones are as follows:

- `geometryName`: Specifies the name of the attribute that stores the feature's geometry information. By default it is `the_geom`.
- `srsName`: The spatial reference system used in the requests. By default it is `"EPSG:4326"`.

Finally, the vector layer uses an `OpenLayers.Strategy.BBOX` strategy, which is responsible to refresh the content of the layer every time the map's viewport changes.

There's more...

Many times the map server that supports WMS and WFS protocols, can serve the same information both in raster and vector formats.

Imagine a set of regions stored in PostgreSQL/PostGIS and a map server, such as GeoServer, with a layer of countries configured to be served both as raster images via WMS requests, or as vector GML format using WFS requests.

In these cases, if we have previously created an `OpenLayers.Layer.WMS` layer, there is an easy way to create a new WFS protocol instance with the static method `OpenLayers.Protocol.WFS.fromWMSLayer`.

Given a WMS layer and some options, the method initializes an `OpenLayers.Protocol.WFS` instance, supposing the WFS `url`, `srsName`, and other properties are the same as in the WMS instance.

See also

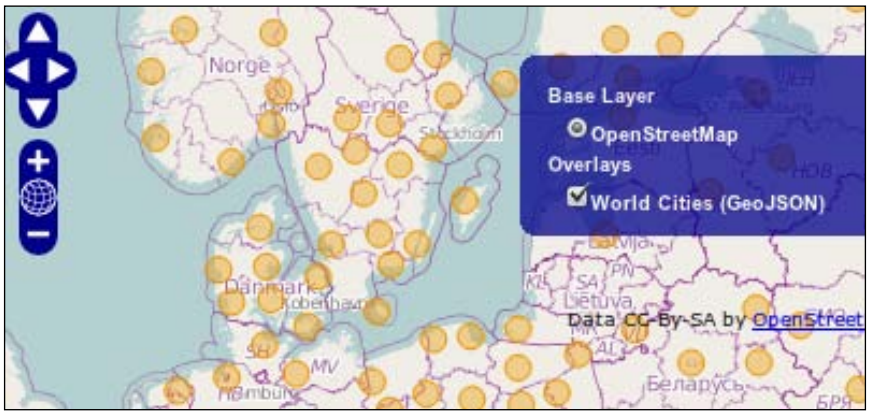
- The *Filtering features in WFS requests* recipe
- The *Working with popups* recipe
- The *Using point features as markers* recipe
- The *Reading features directly using Protocols* recipe

Using the cluster strategy

As we have seen in the chapter's introduction, the behavior of vector layers is determined by the strategies we attach to them.

Imagine a scenario where we want to show all the museums in every city around the world. What will happen when the user navigates within the map and sets a zoom level to see the whole world? We simply see a cloud of points, all at the same place.

The solution to this problem is to cluster the features on each zoom level.



This recipe shows how easy it is to use the cluster strategy on a vector layer, which is responsible for clustering the features to avoid a situation similar to the one we just mentioned.

How to do it...

1. Create an HTML file and insert the following code in it:

```
<!-- Map DOM element -->
<div id="ch3_cluster" style="width: 100%;
    height: 100%;"></div>

<!-- The magic comes here -->
<script type="text/javascript">
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch3_cluster");

    layer = new OpenLayers.Layer.OSM("OpenStreetMap");
    map.addLayer(layer);

    map.addControl(new
       OpenLayers.Control.LayerSwitcher());
    map.setCenter(new OpenLayers.LonLat(0,0), 2);
```

2. As you can see the vector layer is using two strategies:

```
    // World Cities
    var citiesLayer = new OpenLayers.Layer.Vector("World
        Cities (GeoJSON)", {
        protocol: new OpenLayers.Protocol.HTTP({
            url: "http://localhost:8080/
```

```
                openlayers-cookbook/recipes/
                data/world_cities.json",
                format: new OpenLayers.Format.GeoJSON()
            }),
            strategies: [
                new OpenLayers.Strategy.Fixed(),
                new OpenLayers.Strategy.Cluster({distance:
                    15})
            ]
        });
        map.addLayer(citiesLayer);
    </script>
```

How it works...

A vector layer can have more than one strategy associated with it. In this recipe we have added the `OpenLayers.Strategy.Fixed` strategy, which loads the layer content only once, and the `OpenLayers.Strategy.Cluster` strategy, which automatically clusters the features to avoid an ugly cloud of features caused by overlapping:

```
strategies: [
    new OpenLayers.Strategy.Fixed(),
    new OpenLayers.Strategy.Cluster({distance: 15})
]
```

Every time we change the zoom level, the cluster strategy computes the distance among all features and adds all the features that conform to some parameters of the same cluster.

The main parameters we can use to control the behavior of the cluster strategy are as follows:

- `distance`: The distance in pixels between features to be considered that they are in the same cluster. By default it is set to 20 pixels.
- `threshold`: If the number of features in a cluster is less than the threshold, then they will be added directly to the layer instead of the cluster

There's more...

OpenLayers has a set of basic but very common strategies that we can combine in vector layers:

- The *Box* strategy, to request features every time the map's viewport changes
- The *Refresh* strategy, to update the layer features periodically after some time
- The *Filter* strategy to limit the features the layer must request

We encourage those more advanced JavaScript readers, to take a close look at the OpenLayers source code and learn more about how strategies work.

See also

- The *Creating features programmatically* recipe
- The *Adding features from a WFS server* recipe

Filtering features in WFS requests

A key concept when working against a WFS server is the concept of filters.

Among many other specifications, the OGC has defined a standard that defines the notation to be used for filtering, the *Filter Encoding Specification*.

Filters are similar to the `WHERE` clause in SQL and allow us to select features that meet some conditions.

You can find the Filter Encoding Specification on the OGC website available at `http://www.opengeospatial.org/standards/filter`.

As we will see in *Chapter 7, Styling Features*, filters are not only used to query features but are also used to define rules to style them.

OpenLayers offers a set of classes suited to work with the filters the specification defines: property filters (`PropertyIsEqualTo`, `PropertyIsLessThan`, and so on), logical filters, and spatial filters (`Intersects`, `Within`, and so on).

This recipe shows a basic usage of the filter classes to restrict the features queried on a WFS server.

Getting ready

We are going to query a remote WFS server, so we will require a proxy script that was configured in our own server to make the real WFS request.

See the *Adding features from a WFS server* recipe in this chapter for more information about proxy scripts.

How to do it...

1. Create an HTML file and insert the following code:

```
<!-- Map DOM element -->
<div id="ch3_filtering" style="width: 100%;
    height: 100%;"></div>

<!-- The magic comes here -->
<script type="text/javascript">
```

2. The first step in the JavaScript code is to set the proxy script required to solve the cross domain request policy:

```
OpenLayers.ProxyHost = "./utils/proxy.php?url=";

// Create the map using the specified DOM element
var map = new OpenLayers.Map("ch3_filtering");
```

3. Set OSM as the base layer:

```
var baseLayer = new
    OpenLayers.Layer.OSM("OpenStreetMap");
map.addLayer(baseLayer);

map.addControl(new
    OpenLayers.Control.LayerSwitcher());
```

4. To center the map's viewport in a concrete location we need to transform the desired location from latitude/longitude to the projection used by the base layer, that is, the projection used by the map:

```
var center = new OpenLayers.LonLat(-100, 41);
center.transform(new
    OpenLayers.Projection("EPSG:4326"),
    map.getProjectionObject());
map.setCenter(center, 4);
```

5. Add a vector layer, which requests some states:

```
        // Filter features with the query.
        var statesLayer = new
            OpenLayers.Layer.Vector("States", {
            protocol: new OpenLayers.Protocol.WFS({
                url: "http://demo.opengeo.org/geoserver/wfs",
                featureType: "states",
                featureNS: "http://www.openplans.org/topp"
            }),
            strategies: [new OpenLayers.Strategy.BBOX()],
            filter: new OpenLayers.Filter.Logical({
                type: OpenLayers.Filter.Logical.AND,
                filters: [
                    new OpenLayers.Filter.Comparison({
                        type: OpenLayers.Filter.
                            Comparison.GREATER_THAN,
                        property: "MALE",
                        value: "700000"
                    }),
                    new OpenLayers.Filter.Spatial({
                        type:
                            OpenLayers.Filter.Spatial.WITHIN,
                        value: OpenLayers.Bounds.fromArray
                            ([-120, 10,-90,50])
                    })
                ]
            })
        });

        map.addLayer(statesLayer);
    </script>
```

How it works...

The main part of this recipe is the code involved in the instantiation of the vector layer. The constructor receives two parameters, the name and an options object. Within the options object we have set three properties:

```
var statesLayer = new OpenLayers.Layer.Vector("States", {
    protocol: ...,
    strategies: ...,
    filter: ...
});
```

Let's take a look at the protocol, strategies, and filter used in the layer. We are querying a WFS server, so we need to use an `OpenLayers.Protocol.WFS` instance to talk to it:

```
protocol: new OpenLayers.Protocol.WFS({
    url: "http://demo.opengeo.org/geoserver/wfs",
    featureType: "states",
    featureNS: "http://www.openplans.org/topp"
})
```

In the same way as WMS, the WFS server has the `GetCapabilities` action, which allows the client to know the capabilities it provides: kind of features, available operations, and so on.

Check the response obtained from the server used in the recipe: `http://demo.opengeo.org/geoserver/wfs?request=GetCapabilities`.

As a strategy, we want the layer to refresh the features every time the map's viewport is modified, so `OpenLayers.Strategy.BBOX` is the right instance:

```
strategies: [new OpenLayers.Strategy.BBOX()],
```

Finally, there is a `filter` property, which performs all the magic in this recipe. We have tried to use a more or less complete filter, which includes one logical filter, one comparison filter, and one spatial filter:

```
filter: new OpenLayers.Filter.Logical({
    type: OpenLayers.Filter.Logical.AND,
    filters: [
        new OpenLayers.Filter.Comparison({
            type: OpenLayers.Filter.
                Comparison.GREATER_THAN,
            property: "MALE",
            value: "700000"
        }),
        new OpenLayers.Filter.Spatial({
            type: OpenLayers.Filter.Spatial.WITHIN,
            value: OpenLayers.Bounds.fromArray
                ([-120, 10,-90,50])
        })
    ]
})
```

Depending on the kind of filter, they can have different properties with different values allowed.

Our filter queries for all states in the WFS server, on the specified layer, that are within the bounding box defined by `[-120, 10,-90,50]` and have a `MALE` population greater than `700,000`.

There's more...

The `OpenLayers.Protocol` class has a `defaultFilter` property, which allows us to set a default filter for the requests.

The filters specified in the vector layer, will be the logical AND operator, which is merged before making the request.

See also

- The *Adding features from a WFS server* recipe
- The *Reading features directly using Potocols* recipe

Reading features directly using Protocols

OpenLayers allows us to read data from different origins and sources. As we have described in the chapter's introduction, OpenLayers offers the helper classes: protocols and formats.

Protocols are designed to simplify the task of retrieving data from different origins: via HTTP, from an WFS server, and so on.

On the other hand, formats simplifies the task of reading from (or writing to) a given data format. It is very common to load data from different origins and know how to work directly with protocols that can incredibly simplify this task.

As an example, this recipe shows how we can add features from different data sources in the same vector layer, by working directly with the protocol instances.

How to do it...

1. Create an HTML file and add the OpenLayers dependencies. Then create a `DOM` element to hold the map:

```
<!-- Map DOM element -->
<div id="ch3_protocol" style="width: 100%;
    height: 100%;"></div>
```

2. Next, initialize the map, add some base layer, and center the viewport:

```
<script type="text/javascript">
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch3_protocol");

    var baseLayer = new
        OpenLayers.Layer.OSM("OpenStreetMap");
    map.addLayer(baseLayer);

    map.addControl(new
        OpenLayers.Control.LayerSwitcher());
    map.setCenter(new OpenLayers.LonLat(0,0), 2);
```

3. Now, create a vector layer:

```
    var vectorLayer = new
        OpenLayers.Layer.Vector("Vector Layer");
    map.addLayer(vectorLayer);
```

4. Create two protocols pointing to the desired remote files:

```
    // Create HTTP protocol to read GML file
    var gmlReq = new OpenLayers.Protocol.HTTP({
        url: "http://localhost:8080/
        openlayers-cookbook/recipes/data/
        world_cities.json",
        format: new OpenLayers.Format.GeoJSON(),
        callback: addFeaturesFromResponse
    });
    gmlReq.read();

    // Create HTTP protocol to read KML file
    var kmlReq = new OpenLayers.Protocol.HTTP({
        url: "http://localhost:8080/
```

```
            openlayers-cookbook/recipes/data/
            global_undersea.kml",
            format: new OpenLayers.Format.KML({
                extractStyles: true,
                extractAttributes: true
            }),
            callback: addFeaturesFromResponse
        });
        kmlReq.read();
```

5. Finally, add the callback function to be executed when the protocol instances load data from remote files:

```
        // Translate features from EPSG:4326 to OSM
        // projection and add to the layer only
        // the Point geometry features.
        function addFeaturesFromResponse(response) {
            for(var i=0; i<response.features.length; ++i) {
                if(response.features[i].
                geometry.CLASS_NAME ==
                "OpenLayers.Geometry.Point") {
                    response.features[i].geometry.transform
                    (vectorLayer.projection,
                    map.getProjectionObject());
                    vectorLayer.addFeatures
                        ([response.features[i]]);
                }
            }
        }
    </script>
```

How it works...

The goal of this recipe is to show how we can work directly with a protocol and load content from different data sources on the same vector layer.

Because of this we have created an empty vector layer, without specifying the protocol and strategy to use:

```
var vectorLayer = new OpenLayers.Layer.Vector("Vector Layer");
```

After that, we have created an `OpenLayers.Protocol.HTTP` instance that reads a remote `GeoJSON` file:

```
var gmlReq = new OpenLayers.Protocol.HTTP({
    url: "http://localhost:8080/
        openlayers-cookbook/recipes/data/world_cities.json",
```

```
            format: new OpenLayers.Format.GeoJSON(),
            callback: addFeaturesFromResponse
        });
```

Note how we can specify a callback function that will be called once the file is loaded and read it using the desired format. The function receives one parameter of type `OpenLayers.Protocol.Response`, which among others, contains a `features` array property with the set of features read from the file.

To make the protocol start the reading process we simply need to call:

```
gmlReq.read();
```

Finally, let's take a look at the callback function. This function is called when both the protocols finish reading the data. We have implemented it to transform the features to the right projection and add to the vector layer only those of type `OpenLayers.Geometry.Point`:

```
        function addFeaturesFromResponse(response) {
            for(var i=0; i<response.features.length; ++i) {
                if(response.features[i].geometry.CLASS_NAME ==
                    "OpenLayers.Geometry.Point") {
                    response.features[i].geometry.transform
                      (vectorLayer.projection, map.getProjectionObject());
                    vectorLayer.addFeatures([response.features[i]]);
                }
            }
        }
```

As we can see, this is another way to filter the content we put in a vector layer, but take into account that the filtering is made on the client side and not on the server side. That means the entire data is transferred from the server to the client.

There's more...

We would like to mention that in this recipe we do not set the `OpenLayers.ProxyHost` variable. This is because the files we are requesting via AJAX are in the same domain the HTML file is loaded from.

See also

- The *Adding a GML layer* recipe
- The *Adding features from a WFS server* recipe
- The *Filtering features in WFS requests* recipe

4
Working with Events

In this chapter we will cover:

- Creating a side-by-side map comparator
- Implementing a work in progress indicator for map layers
- Listening for vector layer features' events
- Listening for non-OpenLayers events

Introduction

This chapter is focused on events, which is an important concept in any JavaScript program. Although this chapter is brief, the concepts explained here are very important to understand when working with OpenLayers.

Events are the heart of JavaScript. They are the impulses that allow us to produce a reaction. As programmers of a mapping application, we are interested in reacting when the map zoom changes, when a layer is loaded, or when a feature is added to a layer. Every class susceptible to emit events is responsible for managing its listeners (those interested in being notified when an event is fired) and also to emit events under certain circumstances.

For example, we can register a function listening for the `zoomend` event on the `OpenLayers.Map` instance. Every time the map instance changes its zoom, it is responsible to trigger the `zoomend` event, so all its listeners will be notified by the new event.

To help in all this process, OpenLayers has the `OpenLayers.Events` class, that takes care of registering listeners and simplifying the action of firing an event to all of them. In concrete, it allows to:

- Define event
- Register listeners
- Trigger events to notify all listeners

Many classes, such as `OpenLayers.Map` and `OpenLayers.Layer`, have an `events` property, which is an instance of `OpenLayers.Events` that takes care of registering the listeners interested to be notified on their event.

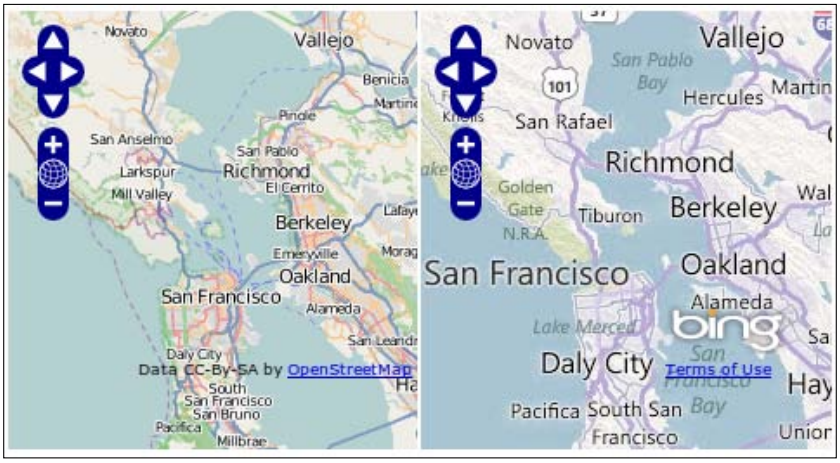
In addition, these classes commonly define an `EVENT_TYPES` array property (which is constant) and list the available events you can register for that class. For example, for the `OpenLayers.Map` class the `EVENT_TYPES` is set as follows:

```
EVENT_TYPES: [
    "preaddlayer", "addlayer","preremovelayer", "removelayer",
    "changelayer", "movestart",
    "move", "moveend", "zoomend", "popupopen", "popupclose",
    "addmarker", "removemarker", "clearmarkers", "mouseover",
    "mouseout", "mousemove", "dragstart", "drag", "dragend",
    "changebaselayer"]
```

As a programmer you need to look at the OpenLayers API documentation (`http://dev.openlayers.org/releases/OpenLayers-2.11/doc/apidocs/files/OpenLayers/Map-js.html`) or you can also refer to the source code to know the available events that you can register on each class.

Creating a side-by-side map comparator

We are going to create a map comparator. The goal is to have two maps side- by-side from different providers and using some of the events that `OpenLayers.Map` instance provides to keep the maps synchronized at the same position and zoom level.



How to do it...

To have two maps side-by-side, perform the following steps:

1. Start creating an HTML with OpenLayers library dependency.
2. Now, add the HTML code required to have two maps side-by-side. Here we are using a table with a row and two columns:

```
<table style="width: 100%; height: 95%;">
    <tr>
        <td>
            <div id="ch04_map_a" style="width: 100%;
            height: 100%;"></div>
        </td>
        <td>
            <div id="ch04_map_b" style="width: 100%;
            height: 100%;"></div>
        </td>
    </tr>
</table>
```

3. Now, let's write the JavaScript code. Create the two maps and initialize with the desired image provider. Here we have used `OpenStreetMap` and `Bing`:

```
<script type="text/javascript">
    // Create left hand side map
    var map_a = new OpenLayers.Map("ch04_map_a");
    var layer_a = new OpenLayers.Layer.OSM("OpenStreetMap");
    map_a.addLayer(layer_a);
    map_a.setCenter(new OpenLayers.LonLat(0,0), 2);

    // Create right hand side map
    var map_b = new OpenLayers.Map("ch04_map_b");
    var bingApiKey = "AvcVU_Eh1H2_
    xVcK0EeRO70MD7Zm6qwLhrVC12C3D997DylUewCWaKR9XTZgWwu6";
    var layer_b = new OpenLayers.Layer.Bing({
        name: "Road",
        type: "Road",
        key: bingApiKey
    });
    map_b.addLayer(layer_b);
    map_b.setCenter(new OpenLayers.LonLat(0,0), 2);
```

4. Now, register the `move` and `zoomend` events on both layers:

```
// Register events on map_a using 'on':
map_a.events.on({
    "move": moveListener,
    "zoomend": zoomListener
});

// Register events on map_a using 'register':
map_b.events.register("move", null, moveListener);
map_b.events.register("zoomend", null, zoomListener);
```

5. Finally, implement the `listener` functions that are called every time an event occurs:

```
// Listener functions

function moveListener(event) {

    if(event.object == map_a) {

        map_b.setCenter(map_a.getCenter());

    } else {

        map_a.setCenter(map_b.getCenter());

    }

}

function zoomListener(event) {

    if(event.object == map_a) {

        map_b.zoomTo(map_a.getZoom()-1);

    } else {

        map_a.zoomTo(map_b.getZoom()+1);

    }

}
</script>
```

How it works...

To keep the two maps always in synchronization at the same position and zoom level, we need to know when the map has moved and when the zoom level has changed.

The `move` event is triggered every time the map is moved. Additionally, there are the `movestart` and `moveend` events, which are fired only when the `move` action starts or ends, but they are not useful here because we need to catch every movement.

The `zoomend` event is triggered when the map's zoom level changes. So, how can we listen for events in the map? This is achieved through the `events` property, which is an instance of `OpenLayers.Events`.

There are two ways (really there is also a third one that we will see in the *There's more* section) to register event listeners for the map events—using the `on` or the `register` methods.

On the first map, we have used the `on` method to register multiple events at once:

```
map_a.events.on({
    "move": moveListener,
    "zoomend": zoomListener
});
```

The `on` method requires an object in which its properties' names are the event names and the values are the listener functions to be called when events are triggered.

The `on` method accepts a special property called `scope`. This allows us to register all the specified events to be executed within the same context. That is, when the `listener` function is executed, the `this` keyword will point to the object specified in the `scope` property.

Contexts can be an advanced topic for someone who has just initiated in JavaScript. An interesting conversation can be found at `http://stackoverflow.com/questions/1798881/javascript-context`.

In the second map, we have used the `register` method, which allows us to register an event listener one at a time:

```
map_b.events.register("move", null, moveListener);
map_b.events.register("zoomend", null, zoomListener);
```

The `events.register()` function accepts four parameters:

- `type`: This is the event we want to listen for.
- `object`: This is the context where the function is executed (similar to the `scope` property in the `on` method).
- `function`: This is the function to be executed when the event is triggered.
- `priority`: This is a Boolean value. If it is `true`, the listener is queued at the front of the event's queue instead of at the end.

Now, we will be notified for any `move` or `zoomend` event that any of the two maps will produce.

It is important to note that OpenLayers event's mechanism always calls the `listener` function by passing an `event` parameter. This `event` object contains any information that is written by the `source` object that triggers the event, plus the following three properties that are always added automatically:

- `type`: Contains the event name (`move`, `zoomend`, and so on)
- `object`: Points to the object that fires the event
- `element`: The `DOM` element related to the event

Let's take a look at our listener functions. The `moveListener` function checks which map has fired the event, then gets the map's `center`, and assigns the same `center` to the other map:

```
function moveListener(event) {
    if(event.object == map_a) {
        map_b.setCenter(map_a.getCenter());
    } else {
        map_a.setCenter(map_b.getCenter());
    }
}
```

As you can see, we can get a reference to the map that triggers the event with `event.object`.

Similarly, the `zoomListener` function gets the zoom level on the source event map and applies it on the other map.

```
function zoomListener(event) {
    if(event.object == map_a) {
        map_b.zoomTo(map_a.getZoom()-1);
    } else {
        map_a.zoomTo(map_b.getZoom()+1);
    }
}
```

Bing maps have different resolution levels on their imagery than `OpenStreetMap`. We can say it differs by one zoom level with respect to other imagery providers because we are adding or subtracting this to/from the zoom level.

There's more...

As we can be interested in listening events, in the same way, we can also be interested in stopping the notifications.

The `OpenLayers.Events` class has the `un` and `unregister` methods, which allow us to unregister our listener functions from notifying when certain events are triggered.

Similar to the `on` method, the `un` method allows to unregister multiple listeners, while the `unregister` method allows to unregister only one listener at a time. Taking this recipe as a sample, we could unregister events on maps as follows:

```
map_a.events.un({
    "move": moveListener,
    "zoomend": zoomListener
});
map_b.events.unregister("move", null, moveListener);
```

Another way to register an event listener

In addition to the `on` and `register` methods, there is a third way to register event listeners.

When creating `OpenLayers.Map`, `OpenLayers.Layer`, and `OpenLayers.Control` instances, we can use the `eventListeners` property, in the same way as we use the `on` method to register a set of listeners. For example:

```
map = new OpenLayers.Map('map', {
    eventListeners: {
        "move": moveListener,
        "zoomend": zoomListener
    }
});
```

What really happens is the object passed to the `eventListener` property is directly used to initialize the listeners by using the `on` method.

See also

- The *Using Bing imagery* recipe in *Chapter 2*, *Adding Raster Layers*
- The *Implementing a work in progress indicator for map layers* recipe
- The *Listening for vector layer features' events* recipe

Implementing a work in progress indicator for map layers

In the art of creating great applications, the most important thing to take into account is the user experience. A good application does what it must do, but by making the user feel comfortable.

When working with remote server, most of the time the user is waiting for data retrieval. For example, when working with a WMS layer, every time we change the zoom level, the user has to wait for some seconds till data is obtained from the server and the tiles start rendering.

It would be great to show some feedback to the users by using an icon, a progress bar, and so on, to inform that the application is working but needs some time.

This recipe shows how we can give some feedback to the user by informing when the application is loading content from different servers, making use of some layer events.

Like in many other recipes in this book, we have used the Dojo toolkit framework (`http://dojotoolkit.org`) for a better user experience. The main difference we can see is that a basic HTML page is the set of rich widgets (buttons, toolbar, progress bar, and so on) it offers. Do not worry if something on the HTML page is not clear, the goal of the book is not teaching Dojo, and that does not alter the explanations about OpenLayers concepts.

How to do it...

Perform the following steps:

1. Create an HTML file with OpenLayers dependency.
2. First we are going to add the HTML code required to show a progress bar. Note how simply it can be created by using the Dojo framework. Tag a normal `span` element with the `data-dojo-type` and `data-dojo-props` attributes.

```
<span data-dojo-type="dijit.ProgressBar" style="width: 100px;"
id="progress"
      data-dojo-props="'indeterminate': true,
      label:''"></span>
```

3. As always, place the `div` element to hold the map:

```
<div id="ch04_work_progress" style="width: 100%; height: 100%;"></
div>
```

4. For starting the JavaScript section code, we need to take into account that we are requesting features from a remote WFS server, because this is the first thing we need to do for setting the proxy URL to be used:

```
<!-- The magic comes here -->
<script type="text/javascript">
    OpenLayers.ProxyHost = "./utils/proxy.php?url=";
```

5. Now, create the map and two layers—a WMS layer, which is the base layer, and a WFS layer:

```
    // Create left map
    var map = new OpenLayers.Map("ch04_work_progress");
    var wms = new OpenLayers.Layer.WMS("Basic",
    "http://labs.metacarta.com/wms/vmap0",
    {
        layers: 'basic'
    });
    var wfs = new OpenLayers.Layer.Vector("States", {
        protocol: new OpenLayers.Protocol.WFS({
            url: "http://demo.opengeo.org/geoserver/wfs",
            featureType: "states",
            featureNS: "http://www.openplans.org/topp"
        }),
        strategies: [new OpenLayers.Strategy.BBOX()]
    });
    map.addLayers([wms, wfs]);
```

6. Add a layer switcher control and centralize the map:

```
    map.addControl(new
    OpenLayers.Control.LayerSwitcher());
    map.setCenter(new OpenLayers.LonLat(-100, 41), 8);
```

7. Register event listeners on WMS and WFS layers:

```
    // Register events on layers using 'on':
    wms.events.on({
        "loadstart": updateLoader,
        "loadend": updateLoader,
        "loadcancel": updateLoader
    });
    wfs.events.on({
        "loadstart": updateLoader,
        "loadend": updateLoader,
        "loadcancel": updateLoader
    });
```

8. Finally, implement the listener function to show the progress bar when any of the two layers is loading its content:

```
    // Listener functions
    var wmsLoading = false;
    var wfsLoading = false;
    function updateLoader(event) {
        var progress = dijit.byId('progress');
        if(event.type == "loadstart") {
            if(event.object == wms) {
                wmsLoading = true;
            }
            if(event.object == wfs) {
                wfsLoading = true;
            }

            var label = "";
            if(wmsLoading) {
                label += "WMS ";
            }
            if(wfsLoading) {
                label += "+ WFS";
            }

            progress.set('value', 'Infinity');
            progress.set('label', label);
            dojo.style(progress.domNode, "visibility", "visible");
        } else {
            if(event.object == wms) {
                wmsLoading = false;
            }
            if(event.object == wfs) {
                wfsLoading = false;
            }
            progress.set('value', '0');
            dojo.style(progress.domNode, "visibility", "hidden");
        }
    }
</script>
```

How it works...

After creating the map and the two layers, register our listener functions for the events `loadstart`, `loadend`, and `loadcancel` on both layers:

```
wms.events.on({
    "loadstart": updateLoader,
    "loadend": updateLoader,
    "loadcancel": updateLoader
});
wfs.events.on({
    "loadstart": updateLoader,
    "loadend": updateLoader,
    "loadcancel": updateLoader
});
```

These are common events to all layers, because they are inherited from the `OpenLayers.Layer` class.

The `loadstart` event is triggered when the layer starts the process of loading data, while `loadend` or `loadcancel` are triggered because the process ends or is canceled.

With this in mind, the cumbersome `updateLoader` listener function is responsible for showing an indeterminate progress bar with a text message when any of the two layers is loading data. The text message can be WMS, WFS, or WMS WFS, depending on the layers that are loading the content.

There's more...

As we mentioned earlier, the events used in this recipe are common for all layers.

Concrete subclasses of the `OpenLayers.Layer` class can have their own events, as in the case of `OpenLayers.Layer.Vector` that has events to notify when features are added, removed, and so on.

See also

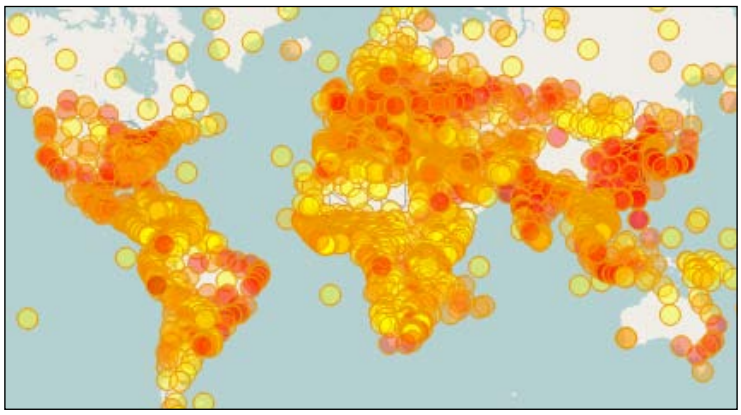
- The *Adding WMS layer* recipe in *Chapter 2*, *Adding Raster Layers*
- The *Adding features from a WFS server* recipe in *Chapter 3*, *Working with Vector Layers*
- The *Creating a side-by-side map comparator* recipe
- The *Listening for vector layer features' events* recipe

Listening for vector layer features' events

When working with vector layers, it is common to find a situation where you need to know what is happening, that is, when a new feature is going to be added to the layers or when a feature has been modified, deleted, and so on. Fortunately, vector layer has the capability to trigger a great fan of events.

The goal of this recipe is to show how easy it is to listen for events in a vector layer and know what is happening on it.

We are going to load a GML file with some cities around the world, and we will style its fill color depending on some feature attribute.



How to do it...

1. Create an HTML file and add the OpenLayers library dependency files. Then, add a `div` element to define where to hold the map instance:

```
<div id="ch04_vector_layer_listener" style="width: 100%; height:
100%;"></div>
```

2. Initialize the map instance, add a base layer, and centralize the viewport:

```
<!-- The magic comes here -->
<script type="text/javascript">
    // Create map
    var map = new
    OpenLayers.Map("ch04_vector_layer_listener");
    var layer = new
    OpenLayers.Layer.OSM("OpenStreetMap");
    map.addLayer(layer);
    map.setCenter(new OpenLayers.LonLat(0,0), 4);
```

3. Create a vector layer to read a GML file. Also, initialize it by registering an event listener for the `beforefeatureadded` event:

```
var vectorLayer = new
OpenLayers.Layer.Vector("States", {
    protocol: new OpenLayers.Protocol.HTTP({
        url: "http://localhost:8080/openlayers-
        cookbook/recipes/data/world_cities.json",
        format: new OpenLayers.Format.GeoJSON()
    }),
    strategies: [new OpenLayers.Strategy.Fixed()],
    eventListeners: {
        "beforefeatureadded": featureAddedListener
    }
});
map.addLayer(vectorLayer);
```

4. Write the code for the listener function. Define a color palette that assigns a fill color to every feature depending on the `POP_RANK` attribute:

```
// Define color palette
var colors = [
    "#CC0000",
    "#FF0000",
    "#FF3300",
    "#FF6600",
    "#FF9900",
    "#FFCC00",
    "#FFFF00"
];
function featureAddedListener(event){
    // Set feature color depending on the rank attribute
    var feature = event.feature;
    var rank = feature.attributes.POP_RANK;
    feature.style = OpenLayers.Util.extend({},
      OpenLayers.Feature.Vector.style['default']);
    feature.style.fillColor = colors[rank-1];
}
</script>
```

How it works...

After initializing the map and the base layer, we have to create a vector layer:

```
        var vectorLayer = new OpenLayers.Layer.Vector("States", {
            protocol: new OpenLayers.Protocol.HTTP({
                url: "http://localhost:8080/openlayers-cookbook/recipes/
    data/world_cities.json",
                format: new OpenLayers.Format.GeoJSON()
            }),
            strategies: [new OpenLayers.Strategy.Fixed()],
            eventListeners: {
                "beforefeatureadded": featureAddedListener
            }
        });
```

As a protocol, we are using the `OpenLayers.Protocol.HTTP` instance that will get data from the specified URL, via HTTP protocol and will read it by using the `OpenLayers.Format.GeoJSON` format reader.

The vector layer uses `OpenLayers.Strategy.Fixed`, which means the content is loaded only once, no matter whether we move the map's viewport or not.

There are some ways to register the event listeners. One of those ways is using the `on` or `register` methods, but we have chosen to register the event listener at the same time when we initialize the layer by using the `eventListener` property.

This way, every time when a feature is going to be added to the layer (before it was added), the listener function will be called by receiving an `event` object as a parameter, with some information related to the layer's event:

```
        function featureAddedListener(event){
            var feature = event.feature;
            var rank = feature.attributes.POP_RANK;
            feature.style = OpenLayers.Util.extend({},
              OpenLayers.Feature.Vector.style['default']);
            feature.style.fillColor = colors[rank-1];
        }
```

From the event, we can get a reference to the feature and its attributes. Here we are using the `POP_RANK` attribute to select the fill color of the feature.

More information about the feature style properties, which we can change, is available at `http://dev.openlayers.org/releases/OpenLayers-2.11/doc/apidocs/files/OpenLayers/Feature/Vector-js.html#OpenLayers.Feature.Vector.OpenLayers.Feature.Vector.style`.

There's more...

In this recipe, we can use the `OpenLayers.Util.extend` method to set the initial style of the feature and then set the desired fill color:

```
feature.style = OpenLayers.Util.extend(
    {},
    OpenLayers.Feature.Vector.style['default']
);
```

The `OpenLayers.Util.extend` method requires two parameters—the destination and the source objects. Its function is to copy all the properties found in the source object to the destination.

The `OpenLayers.Util.extend` method is very important to create hierarchy and inheritance in OpenLayers. However, its namespace is `OpenLayers.Util` and it is located in the `OpenLayers/BaseTypes/Class.js` file, which talks about its importance.

On the other hand, `OpenLayers.Feature.Vector.style` is an object with some predefined styles for features such as `default`, `selected`, `delete`, and so on.

So, the preceding line means that a new object extending an empty object with all the properties in the `OpenLayers.Feature.Vector.style['default']` object can be created.

See also

- The *Styling features using symbolizers* recipe in *Chapter 7*, *Styling Features*
- The *Adding a GML layer* recipe in *Chapter 3*, *Working with Vector Layers*.
- The *Creating a side-by-side map comparator* recipe
- The *Listening for non-OpenLayers events* recipe

Listening for non-OpenLayers events

When developing a web mapping application, the use of OpenLayers is only a piece among the set of tools that we need to use. Adding other components, such as buttons, images, lists, and so on, and interacting with them are other tasks that we must work on.

Interacting with a `OpenLayers.Map` instance or `OpenLayers.Layer` subclass is easy because they trigger specific events, but what if we want to listen for events on a button or any `DOM` element?

For this purpose, OpenLayers offers us the `OpenLayers.Event` class (do not get confused with the plural `OpenLayers.Events` class). This is a helper class, which, among other things, allows us to listen for events in non-OpenLayers elements in a browser-independent way.

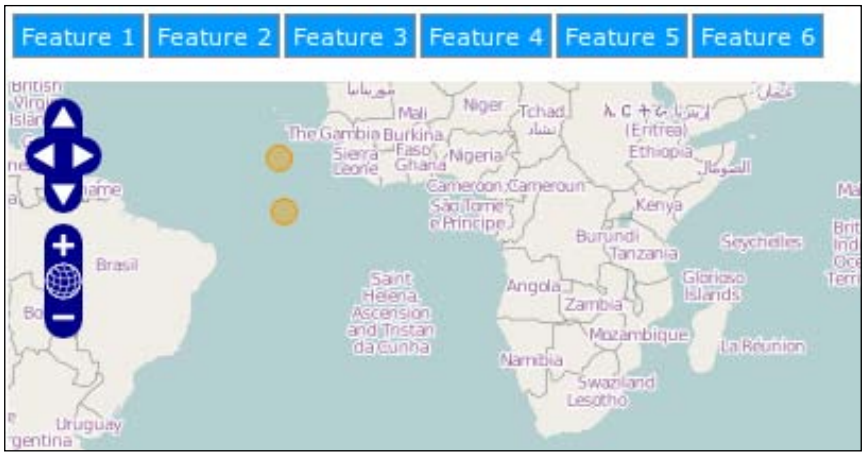
Unfortunately the way to register event listeners in JavaScript is not the same in all browsers. Also, Microsoft differs from W3C (the WWW Consortium) in the way to register listeners. You can find more information at `http://www.quirksmode.org/js/events_advanced.html`.

If your project uses a library or framework such as jQuery, Dojo, or ExtJS, you will probably use their features to access DOM elements, register for events, and so on.

If you are working on a simpler project without the aforementioned libraries, it is a good idea to register events through the `OpenLayers.Event` class, because it is browser-independent, which means your application will be compatible with more browsers.

In addition, there is one more reason to read this recipe and the reason is that OpenLayers uses the `OpenLayers.Event` class internally to implement many handlers and controls, which we will see in the future chapters.

Let's have a look at how we can listen for events on HTML elements through the `OpenLayers.Event` class.



The idea is to create six buttons and add six point features to a vector layer. Then highlight the feature when mouse enters a button or unselect if mouse leaves it.

How to do it...

To listen for non-OpenLayers events, follow the next steps:

1. Create an HTML with OpenLayers library dependency. Start adding some CSS styles for the buttons. The following code defines a style when the buttons are not selected (the mouse is out) and also a style with different background color when mouse is hovered over the buttons:

```
<style>
    .square {
        border: 1px solid #888;
        background-color: #0099FF;
        color: #fff;
        padding: 3px;
    }
    .square:hover {
        background-color: #0086d2;
    }
</style>
```

2. Create a table to hold the six buttons. A button will be represented by a `span` element with an identifier:

```
<table>
    <tr>
        <td><span id="f0" class="square">Feature
        1</span></td>
```

```
            <td><span id="f1" class="square">Feature
            2</span></td>
            <td><span id="f2" class="square">Feature
            3</span></td>
            <td><span id="f3" class="square">Feature
            4</span></td>
            <td><span id="f4" class="square">Feature
            5</span></td>
            <td><span id="f5" class="square">Feature
            6</span></td>
        </tr>
    </table>
    <br/>
```

3. Add a `div` element to hold the map:

```
<div id="ch04_dom_events" style="width: 100%; height: 100%;"></
div>
```

4. Now, add the JavaScript code required to instantiate the map object, set a base layer, and add a vector layer:

```
<!-- The magic comes here -->
<script type="text/javascript">
    // Create left map
    var map = new OpenLayers.Map("ch04_dom_events");
    var osm = new OpenLayers.Layer.OSM();
    // Create a vector layer with one feature for each
    previous SPAN element
    var vectorLayer = new
    OpenLayers.Layer.Vector("Features");
```

5. Populate the vector layer with six features. Each one will contain the identifier of the button that represents it:

```
    var pointFeatures = [];
    for(var i=0; i< 6; i++) {
        // Create the ID
        var id = "f"+i;
        // Regiter listeners to handle when mouse enters
        and leaves the DOM element
        OpenLayers.Event.observe(OpenLayers.Util.
        getElement(id), 'mouseover', mouseOverListener);
        OpenLayers.Event.observe(OpenLayers.Util.
        getElement(id), 'mouseout', mouseOutListener);

        // Create a random point
        var px = Math.random()*360-180;
        var py = Math.random()*160-80;
```

```
            var pointGeometry = new
            OpenLayers.Geometry.Point(px, py);
            OpenLayers.Projection.transform(pointGeometry,
            new OpenLayers.Projection("EPSG:4326"), new
            OpenLayers.Projection("EPSG:900913"));
            var pointFeature = new
            OpenLayers.Feature.Vector(pointGeometry, {
                elem_id: id
            });
            pointFeatures.push(pointFeature);
        }
        vectorLayer.addFeatures(pointFeatures);

        map.addLayers([osm, vectorLayer]);
        map.setCenter(new OpenLayers.LonLat(0, 0), 1);
```

6. Finally, add the code that implements the event listeners:

```
        // Listeners
        function mouseOverListener(event) {
            var id = event.target.id;
            var feature = vectorLayer.
            getFeaturesByAttribute('elem_id', id);
            vectorLayer.drawFeature(feature[0], "select");
        }
        function mouseOutListener(event) {
            var id = event.target.id;
            var feature = vectorLayer.
            getFeaturesByAttribute('elem_id', id);
            vectorLayer.drawFeature(feature[0], "default");
        }
    </script>
```

How it works...

We have created six buttons, identified from `f0` to `f5`, and we want to create six features that represent them. To do this, in the `for` loop, first we create a string with an identifier:

```
var id = "f"+i;
```

Then, register an event listener function for the `mouseover` and `mouseout` events:

```
OpenLayers.Event.observe(OpenLayers.Util.getElement(id),
'mouseover', mouseOverListener);
OpenLayers.Event.observe(OpenLayers.Util.getElement(id),
'mouseout', mouseOutListener);
```

This is done by using the `OpenLayers.Event.observe` method, which requires three parameters. These parameters are as follows:

- `elementParam`: The `DOM` element reference, or its identifier, which we want to listen to for `tis` events
- `name`: The event you want to listen to
- `observer`: The function that will act as a listener

Because we need to pass the `DOM` element reference, we need to get it first. To get an element reference when its identifier is available, we can use the helper method `OpenLayers.Util.getElement`.

From the `elementParam` definition, you can see that the use of `OpenLayers.Util.getElement` is not strictly necessary. If we pass an ID, the `OpenLayers.Event.observe` method will internally use the `OpenLayers.Util.getElement` function to get the element reference, so the next two lines will have the same result:

```
        OpenLayers.Event.observe(id, 'mouseover', mouseOverListener);
        OpenLayers.Event.observe(OpenLayers.Util.getElement(id),
'mouseover', mouseOverListener);
```

The `OpenLayers.Util` class has plenty of methods to help in working with the `DOM` elements, arrays, and many more functions. We encourage you to take a look.

Once the listeners are registered, we create a random point feature and add it to the vector layer:

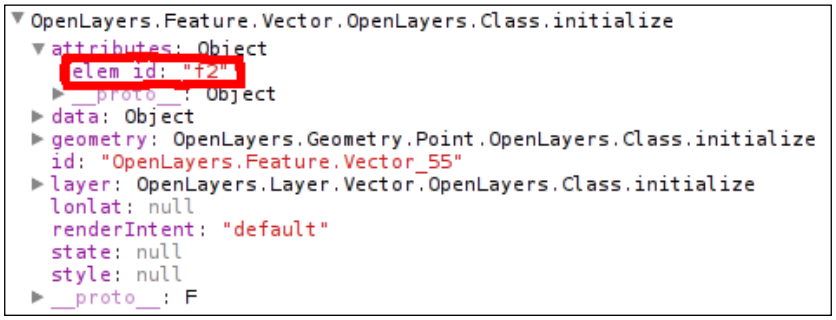
```
var px = Math.random()*360-180;
var py = Math.random()*160-80;
var pointGeometry = new OpenLayers.Geometry.Point(px, py);
```

Remember to transform the point coordinates to the projection used by the map. In this case, because the base layer is OSM and the map has no specified projection property, the OSM projection will be used:

```
OpenLayers.Projection.transform(pointGeometry, new
OpenLayers.Projection("EPSG:4326"), new
OpenLayers.Projection("EPSG:900913"));
var pointFeature = new
OpenLayers.Feature.Vector(pointGeometry, {
    elem_id: id
});
pointFeatures.push(pointFeature);
```

We have created the feature by passing a custom attribute `elem_id`, which will store the identifier of the button that represents the feature. This way we have a reference to connect the feature and the button.

The following screenshot shows how custom attributes are stored within the feature `attributes` property:



At this point we have six buttons and six features, which store the corresponding button identifiers as the custom attributes. Now, the task is to implement the listener function. Let's have a look at the `mouseOverListener` function.

```
function mouseOverListener(event) {
    var id = event.target.id;
    var feature =
    vectorLayer.getFeaturesByAttribute('elem_id', id);
    vectorLayer.drawFeature(feature[0], "select");
}
```

From the event, which is a browser `MouseEvent`, we get the identifier of the target element that has triggered the event:

```
var id = event.target.id;
```

Next, using the `OpenLayers.Layers.Vector.getFeatureByAttribute` method, we get an array of features within the vector layer that has the `elem_id` with the value `id`. Of course, here it will always return an array with only one element:

```
var feature = vectorLayer.getFeaturesByAttribute('elem_id', id);
```

Now, we have the feature. Simply redraw it with a different render intent. Select to highlight the feature as selected and put its style back to default:

```
vectorLayer.drawFeature(feature[0], "select");
```

We will see more about styling features in *Chapter 7*, *Styling Features*. Meanwhile, consider render intents as predefined styles to render features.

There's more...

OpenLayers defines a global variable `$`, which points to the `OpenLayers.Util.getElement` function, if it does not exist. This way we can get a reference to an element in a short way.

For example, the next two lines have the same result:

```
$("some_ID")
OpenLayers.Util.getElement("some_ID")
```

Be careful with the use of the `$` function. Many JavaScript libraries, one of the most known is jQuery library (`http://jquery.com`), also define the global `$` object as a common way to operate with it. So, check twice the order in which you have imported libraries on your application and where the `$` function really points.

As a curiosity, while getting an element reference by its identifier with `OpenLayers.Util.getElement` written:

```
$("some_ID")
```

jQuery library requires you to use the # character:

```
$("#some_ID")
```

Stop observing

We can be interested in observing some event, in the same way, we can also have a desire to stop observing it.

Similar to the `OpenLayers.Event.observe` method, given an element reference or a string identifier, the `OpenLayers.Event.stopObservingElement` method allows us to stop observing some `DOM` element.

See also

- The *Creating features programmatically* recipe in *Chapter 3*, *Working with Vector Layers*
- The *Styling features using symbolizers* recipe in *Chapter 7*, *Styling Features*
- The *Creating a side-by-side map comparator* recipe
- The *Listening for vector layer features' events* recipe

5
Adding Controls

In this chapter we will cover:

- Adding some visual controls
- Adding the NavigationHistory control
- Working with geolocation
- Placing controls outside the map
- Editing features on multiple vector layers
- Modifying features
- Measuring distances and areas
- Getting feature information from a data source
- Getting information from a WMS server

Introduction

This chapter explores from the basics, the most important and common controls that OpenLayers offers us as developers. Controls allow us to navigate through the map, play with layers, zoom in or out, perform actions such as editing features, measuring distances, and the like. In essence, controls allow us to interact.

The `OpenLayers.Control` class is the base class for all the controls and contains the common properties and methods that a control can have. We can summarize this as follows:

- A control is attached to a map
- A control can trigger events
- A control can be activated or deactivated
- A control can have a visual representation (such as a button) or have no visual representation (such as the drag action)

Controls are closely related to the **handlers**. While controls are designed to contain the logic of the action, they delegate to the handlers *the low-level tasks*, such as to know about the mouse or keyboard events. For example, the `OpenLayers.Control.DragPan` control is responsible for dragging the map by reacting to the mouse events. While the task, to listen to the mouse events, is delegated to an internal instance of the `OpenLayers.Handler.DragPan` class, the task to move the map is made by the control itself.

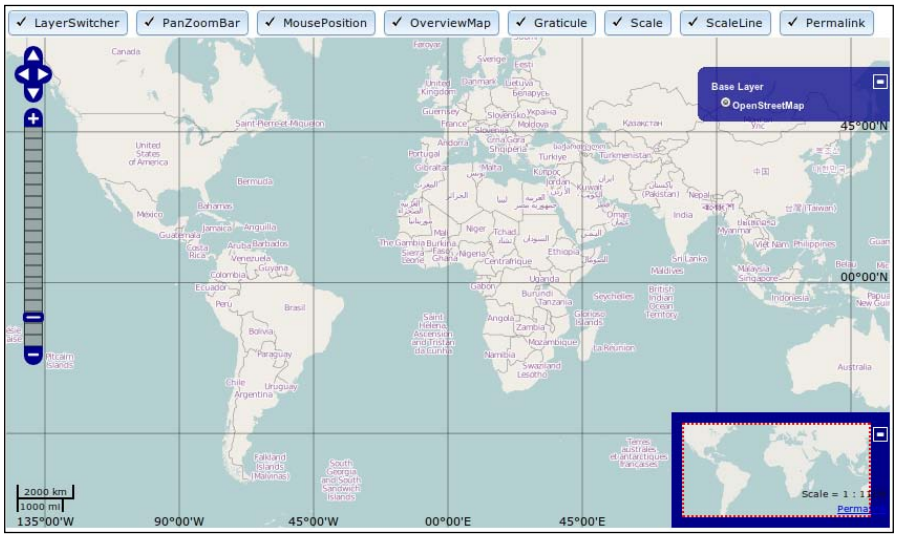
In a similar way as with the controls, the class `OpenLayers.Handler` is the base class for all the existing handlers used by the controls.

Let's see some recipes that will help us to understand the controls better.

Adding and removing controls

OpenLayers offers a great number of controls, commonly used on mapping applications.

This recipe shows how to use the most common controls that have a visual representation. The list includes the OverviewMap control, the Scale and ScaleLine controls, the Graticule control, the LayerSwitcher control, the PanZoomBar control, the MousePosition control, and the Permalink control:



How to do it...

1. First add the code for the buttons:

```
<button data-dojo-type="dijit.form.ToggleButton" data-dojo-props="
iconClass:'dijitCheckBoxIcon', checked: true, onChange: layerSwitc
herChanged">LayerSwitcher</button>
```

```
<button data-dojo-type="dijit.form.ToggleButton" data-dojo-pr
ops="iconClass:'dijitCheckBoxIcon', checked: true, onChange:
panZoomBarChanged">PanZoomBar</button>
<button data-dojo-type="dijit.form.ToggleButton" data-dojo-props="
iconClass:'dijitCheckBoxIcon', checked: true, onChange: mousePosit
ionChanged">MousePosition</button>
<button data-dojo-type="dijit.form.ToggleButton" data-dojo-props="
iconClass:'dijitCheckBoxIcon', checked: true, onChange: overviewMa
pChanged">OverviewMap</button>
<button data-dojo-type="dijit.form.ToggleButton" data-dojo-pr
ops="iconClass:'dijitCheckBoxIcon', checked: true, onChange:
graticuleChanged">Graticule</button>
<button data-dojo-type="dijit.form.ToggleButton" data-dojo-pr
ops="iconClass:'dijitCheckBoxIcon', checked: true, onChange:
scaleChanged">Scale</button>
<button data-dojo-type="dijit.form.ToggleButton" data-dojo-pr
ops="iconClass:'dijitCheckBoxIcon', checked: true, onChange:
scaleLineChanged">ScaleLine</button>
<button data-dojo-type="dijit.form.ToggleButton" data-dojo-pr
ops="iconClass:'dijitCheckBoxIcon', checked: true, onChange:
permalinkChanged">Permalink</button>
```

We are using the **Dojo Toolkit** (`http://dojotoolkit.org/`) to create the richest user interface, thanks to the beautiful components it offers. The goal of the recipe is not to teach Dojo, but to teach OpenLayers, so we are free to change the code related to HTML, to use checkbox elements for input, instead of the Dojo toggle buttons, and work with the `onclick` event.

The importance of the recipe is that the reader learns about creating different controls, attaching them to the map, and activating or deactivating them.

2. Next, add the `div` element to hold the map:

```
<div id="ch05_visual_controls" style="width: 100%; height:
90%;"></div>
```

3. Create the map instance and add a base layer:

```
<!-- The magic comes here -->
<script type="text/javascript">
    // Create map
    var map = new OpenLayers.Map("ch05_visual_controls", {
        controls: []
    });
    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);
```

4. Add the set of controls:

```
    // Add controls
    var layerSwitcher = new OpenLayers.Control.LayerSwitcher({'asc
ending':false});
    var panZoomBar = new OpenLayers.Control.PanZoomBar();
    var mousePosition = new OpenLayers.Control.MousePosition();
    var overviewMap = new OpenLayers.Control.
OverviewMap({maximized: true});
    var graticule = new OpenLayers.Control.Graticule({displayInLay
erSwitcher: false});
    var scale = new OpenLayers.Control.Scale();
    var scaleline = new OpenLayers.Control.ScaleLine();
    var permalink = new OpenLayers.Control.Permalink();

    map.addControls([layerSwitcher, panZoomBar, mousePosition,
overviewMap,
        graticule, scale, scaleline, permalink]);

    map.setCenter(new OpenLayers.LonLat(0, 0), 2);
```

5. Finally, add the code to add or remove the controls depending on the state of its corresponding buttons:

```
    function layerSwitcherChanged(checked) {
        if(checked) {
            layerSwitcher = new OpenLayers.Control.LayerSwitcher({
'ascending':false});
            map.addControl(layerSwitcher);
        } else {
            map.removeControl(layerSwitcher);
            layerSwitcher.destroy();
        }
    }
    function panZoomBarChanged(checked) {
        if(checked) {
            panZoomBar = new OpenLayers.Control.PanZoomBar();
            map.addControl(panZoomBar);
        } else {
            map.removeControl(panZoomBar);
            panZoomBar.destroy();
        }
    }
    function mousePositionChanged(checked) {
        if(checked) {
            mousePosition = new OpenLayers.Control.
MousePosition();
```

```
            map.addControl(mousePosition);
        } else {
            map.removeControl(mousePosition);
            mousePosition.destroy();
        }
    }
```

6. Each function receives a `checked` parameter that indicates if the button is pressed or not. Depending on its value, we simply add or remove the control from the map:

```
function overviewMapChanged(checked) {
        if(checked) {
            overviewMap = new OpenLayers.Control.
OverviewMap({maximized: true});
            map.addControl(overviewMap);
        } else {
            map.removeControl(overviewMap);
            overviewMap.destroy();
        }
    }
    function graticuleChanged(checked) {
        if(checked) {
            graticule = new OpenLayers.Control.Graticule({displayI
nLayerSwitcher: false});
            map.addControl(graticule);
        } else {
            map.removeControl(graticule);
            graticule.destroy();
        }
    }
    function scaleChanged(checked) {
        if(checked) {
            scale = new OpenLayers.Control.Scale();
            map.addControl(scale);
        } else {
            map.removeControl(scale);
            scale.destroy();
        }
    }
    function scaleLineChanged(checked) {
        if(checked) {
            scaleline = new OpenLayers.Control.ScaleLine();
            map.addControl(scaleline);
        } else {
            map.removeControl(scaleline);
```

```
                scaleline.destroy();
            }
        }
        function permalinkChanged(checked) {
            if(checked) {
                permalink = new OpenLayers.Control.Permalink();
                map.addControl(permalink);
            } else {
                map.removeControl(permalink);
                permalink.destroy();
            }
        }
    </script>
```

How it works...

The first thing we have done is to create the map instance, forcing it to have no controls attached to it. This is done by setting the `controls` property to an empty array:

```
// Create map
var map = new OpenLayers.Map("ch05_visual_controls", {
    controls: []
});
```

When we create an `OpenLayers.Map` instance without specifying the `controls` property, OpenLayers automatically adds the next set of default controls to it: Navigation, PanZoom, ArgParser, and Attribution.

Next, we have created the controls and added all of them to the map using the `addControls` method:

```
    var layerSwitcher = new OpenLayers.Control.LayerSwitcher({'ascend
ing':false});
    …
    …
    ...
    var permalink = new OpenLayers.Control.Permalink();

    map.addControls([layerSwitcher, panZoomBar, mousePosition,
overviewMap, graticule, scale, scaleline, permalink]);
```

Before continuing, let's take a look at the properties used in some of the controls' instantiation.

On the layer switcher, we have set the property `ascending` to `false`. This means the layers will be sorted in descending order, that is they will be added to the map in the reverse order:

```
    var layerSwitcher = new OpenLayers.Control.LayerSwitcher({'ascend
ing':false});
```

On the OverviewMap control, the `maximized` property allows us to expand the control created:

```
    var overviewMap = new OpenLayers.Control.OverviewMap({maximized:
true});
```

Finally, for the Graticule control, the `displayInLayerSwitcher` property allows to switch it on or off in the LayerSwitcher control:

```
    var graticule = new OpenLayers.Control.Graticule({
displayInLayerSwitcher: false});
```

Thanks to the Dojo Toolkit, the buttons we have created have the behavior of a toggle button. Each button has a function associated with it that is executed every time the button state changes from checked to unchecked. In the case of the overview map button, the associated function is `overviewMapChanged` that is specified in the `onChange` event within the `data-dojo-props` attribute:

```
<button data-dojo-type="dijit.form.ToggleButton" data-dojo-props="icon
Class:'dijitCheckBoxIcon', checked: true, onChange: overviewMapChanged
">OverviewMap</button>
```

The function that acts as the listener for the `onChange` event receives a boolean parameter, indicating if the button is checked or unchecked.

All the listener functions are similar. Depending on the value of the checked parameter, it removes (and destroys) the control from the map or creates a new one:

```
    function overviewMapChanged(checked) {
        if(checked) {
            overviewMap = new OpenLayers.Control.
OverviewMap({maximized: true});
            map.addControl(overviewMap);
        } else {
            map.removeControl(overviewMap);
            overviewMap.destroy();
        }
    }
```

In the same way as layers, removing a control from the map instance with the `removeControl()` method does not free the possible resources used by the control. We need to explicitly do it with the `destroy()` method.

See also

- The *Placing controls outside the map* recipe
- The *Understanding how themes work using img folder* recipe (theming the PanZoomBar control) in *Chapter 6*, *Theming*

Adding a navigation history control

Probably the most commonly used control in our mapping applications will be the Navigation control. `OpenLayers.Control.Navigation` control integrates (makes use of) some other controls, such as `OpenLayers.Control.DragPan`, `OpenLayers.Control.ZoomBox`, or a wheel handler, which allows us to pan and zoom the map.

While navigating, moving, or zooming, it can be interesting to store a history of the navigation actions made by the user, so he/she can go back or forward to previous places. Fortunately, we don't need to reinvent the wheel. OpenLayers offers us the `OpenLayers.Control.NavigationHistory` control.

This recipe shows how easy it is to add it to our applications and benefit from its features.

As you can see in the screenshot, we are going to add a button above the map that will enable or disable the Navigation component.

How to do it...

1. Create an HTML file with the required OpenLayers dependencies. Add the code for the toggle button that will enable/disable the navigation control:

```
<button data-dojo-type="dijit.form.ToggleButton" data-dojo-pr
ops="iconClass:'dijitCheckBoxIcon', checked: true, onChange:
navigationChanged">Navigation</button>
```

2. Next, add a `div` element to hold the map:

```
<div id="ch05_nav_history" style="width: 100%; height: 90%;"></
div>
```

3. Now, create the map instance and add a base layer:

```
<script type="text/javascript">
    // Create map
    var map = new OpenLayers.Map("ch05_nav_history", {
        controls: []
    });
    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);
```

4. Add the Navigation and NavigationHistory controls:

```
    // Add controls
    var navigation = new OpenLayers.Control.Navigation();
    var history = new OpenLayers.Control.NavigationHistory();
    var panel = new OpenLayers.Control.Panel();
    panel.addControls([history.next, history.previous]);

    map.addControls([navigation, history, panel]);
    map.setCenter(new OpenLayers.LonLat(0, 0), 4);
```

5. Implement the function responsible to enable/disable the navigation control:

```
    function navigationChanged(checked) {
        if(checked) {
            navigation.activate();
        } else {
            navigation.deactivate();
        }
    }
</script>
```

How it works...

First let's talk about the navigation control. Using it is not a mystery. Simply create a control instance and add it to the map:

```
var navigation = new OpenLayers.Control.Navigation();
….
….
map.addControls([navigation, ...]);
```

The button created at the beginning makes use of the Dojo Toolkit, which allows us to easily convert it to a toggle button. In addition, we have added a listener function to check when the button's state changes between checked and unchecked. The `navigationChanged` function activates or deactivates the control depending on the `checked` value:

```
function navigationChanged(checked) {
    if(checked) {
        navigation.activate();
    } else {
        navigation.deactivate();
    }
}
```

Each control has an `activate()` and `deactivate()` method. They are defined in the base class, `OpenLayers.Control`, and all concrete controls inherit or override these methods.

The use of `activate` and `deactivate` is preferred over removing and adding the control from/to the map. This way, there is no need to either create or attach instances of the control. The control is simply in standby until we activate it again.

That is all related to the navigation control, let's take a look at how to add the navigation history control, because this is just a two-step process.

The `OpenLayers.Control.NavigationHistory` control is a bit more special. It contains stacks to store the previous and next visited places and, among others, also contains references to two buttons (instances of the `OpenLayers.Control.Button` control class), which allows us to go back and forward in the navigation history. The references to these buttons can be found in the `previous` and `next` properties.

By default, after adding a NavigationHistory control to the map, no button appears. It is our responsibility to show the previous and next buttons on the map. For this, and other similar purposes, OpenLayers offers us the `OpenLayers.Control.Panel` control class. It is a special kind of control that can contain or group together other controls. So, with all this in mind, we can now explain the way the Navigation History control is added to the map.

First we need to create the `OpenLayers.Control.NavigationHistory` instance and add it to the map. Second, we need to add a panel to show the two buttons and add the two buttons:

```
var history = new OpenLayers.Control.NavigationHistory();
var panel = new OpenLayers.Control.Panel();
panel.addControls([history.next, history.previous]);
```

Finally, the panel itself must be added to the map as a new control:

```
map.addControls([navigation, history, panel]);
```

As you can see, we have added the navigation, the navigation history, and the panel with the buttons as map controls, simply because all three are controls.

In *Chapter 6*, *Theming*, we will see how we can change the icons used by this control.

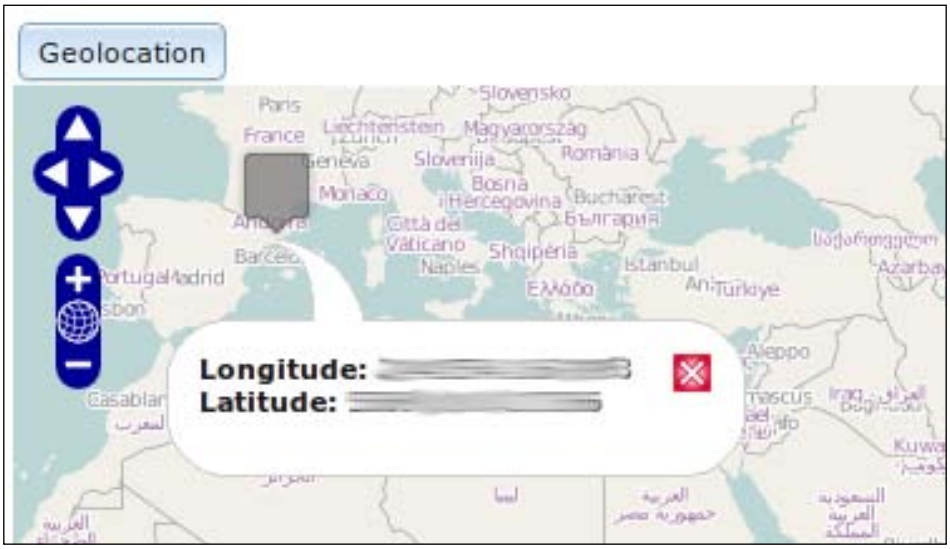
See also

- The *Adding and removing controls* recipe
- The *Placing controls outside the map* recipe
- The *Understanding how themes work using the theme folder* recipe in *Chapter 6*, *Theming*

Working with geolocation

With the arrival of **HTML5**, one of the many new APIs and concepts introduced in the specification is the possibility to identify the location of the client that is loading the web page, through the **Geolocation API** (`http://dev.w3.org/geo/api/spec-source.html`). Of course, in the world of web mapping applications, this opens new and great possibilities.

In this recipe, we are going to show how easily we can identify the current location of the user and center the map's viewport to it:



Every time the user clicks on the **Geolocation** button, the map's viewport will be moved to the current user's location and a marker will be placed on it. Also, when the mouse goes over the marker, a popup with the current location will be shown.

Getting ready

As we mentioned at the beginning of the recipe, Geolocation is a feature that the browser must implement, so we need an HTML5 compliant browser to make this control work.

How to do it...

1. First create the HTML file with OpenLayers dependencies, then add the HTML code for the button and map element:

```
<button data-dojo-type="dijit.form.Button" data-dojo-
props="onClick: geolocationClick">Geolocation</button>
<div id="ch05_geolocating" style="width: 100%; height: 90%;"></
div>
```

2. Then, initialize the map instance and add a base layer:

```
<script type="text/javascript">
    // Create map
    var map = new OpenLayers.Map("ch05_geolocating");
    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);
```

3. Now, add the `OpenLayers.Control.Geolocate` control to the map:

```
    // Add controls
    var geolocate = new OpenLayers.Control.Geolocate({
        eventListeners: {
            "locationupdated": locateMarker,
            "locationfailed": function() {
                console.log('Location detection failed');
            }
        }
    });
    map.addControl(geolocate);
```

4. Create and add to the map, the marker layer, where the marker will be placed:

```
    var markers = new OpenLayers.Layer.Markers("Markers");
    map.addLayer(markers);
```

5. Set an initial place for the view:

```
    map.setCenter(new OpenLayers.LonLat(0, 0), 6);
```

6. Implement the listener function associated to the **Geolocation** button:

```
    function geolocationClick() {
        geolocate.deactivate();
        geolocate.activate();
    }
```

7. Finally, implement the function that is executed each time the location of the client is detected. The purpose of this function is to add a marker to the map at the current client's location, and show a popup with the coordinates when the mouse goes over the marker:

```
    function locateMarker(event) {
```

8. Start by removing any previous markers:

```
        // Remove any existing marker
        markers.clearMarkers();
```

9. Then, create the icon to be used by the marker:

```
        var size = new OpenLayers.Size(32, 37);
        var offset = new OpenLayers.Pixel(-(size.w/2), -size.h);
        var icon = new OpenLayers.Icon('./recipes/data/icons/
symbol_blank.png', size, offset);
        icon.setOpacity(0.7);

        // Create a lonlat instance from the event location.
        // NOTE: The coordinates are transformed to the map's
projection by
        // the geolocate control.
        var lonlat = new OpenLayers.LonLat(event.point.x, event.
point.y);

        // Add the marker
        var popup = null;
        var marker = new OpenLayers.Marker(lonlat, icon);
```

10. Then, register a listener for the mouseover event that will show the popup:

```
        marker.events.on({
            "mouseover": function() {
                if(popup) {
                    map.removePopup(popup);
                }

                var content = "<strong>Longitude:</strong> " +
lonlat.lon + "<br/>" + "<strong>Latitude:</strong> " + lonlat.lat;

                popup = new OpenLayers.Popup.FramedCloud(
                "popup", lonlat, new OpenLayers.Size(250, 100),
content,
                null, true, null);

                map.addPopup(popup);
```

```
                }
            });

            markers.addMarker(marker);
        }
    </script>
```

How it works...

The first step is to create the `OpenLayers.Control.Geolocate` control instance and add it to the map:

```
var geolocate = new OpenLayers.Control.Geolocate({
    eventListeners: {
        "locationupdated": locateMarker,
        "locationfailed": function() {
            console.log('Location detection failed');
        }
    }
});
```

The control can trigger three events:

- `locationupdated`: This event is fired when the browser returns a new position
- `locationfailed`: This event is fired if the geolocation fails
- `locationuncapable`: This event is fired if you activate the control in a browser that does not support geolocation.

In this recipe, we attached an event listener function for the events `locationupdated` and `locationfailed`.

To use the Geolocate control, we need to invoke its `activate()` method. Then, OpenLayers will request the browser to get the current user's location and the browser will ask if we want to share our location. If we accept, then a `locationupdated` event will be triggered with the current location as an argument.

In the recipe, the `geolocationClick` function is called when the button is clicked and forces the activation of the control:

```
function geolocationClick() {
    geolocate.deactivate();
    geolocate.activate();
}
```

Then, when the `locationupdated` event is triggered, the `locateMarker` function is executed, passing an `event` parameter with all the related event information, including the client coordinates:

```
function locateMarker(event) {…}
```

The coordinates stored at `event.point` are transformed by the **Geolocate** control, to be in the same coordinate system as the map.

The purpose of this function is to add a marker to the map at the current client's location and show a popup with the coordinates when the mouse goes over the marker.

There's more...

The `OpenLayers.Control.Geolocate` control has a couple of interesting properties.

First the `bind` property, by default set to `true`, allows us to specify if the map's center must be updated to the location detected by the control.

The `watch` property, by default set to `false`, allows updating the position regularly.

In addition, we can pass to the control a `geolocationOptions` object, defined in the specification (see `http://dev.w3.org/geo/api/spec-source.html#position_options_interface`) for better configuration of the control.

See also

- The *Adding and removing controls* recipe
- The *Modifying features* recipe

Placing controls outside the map

By default, all the controls are placed on the map. This way, controls such as the PanPanel, EditingToolbar, or MousePosition are rendered on top of the map and over any layer. This is the default behavior, but OpenLayers is flexible enough to allow us to put controls outside the map:

In this recipe we are going to create a map where the navigation toolbar and the mouse position controls are placed outside and above the map.

How to do it...

1. Create an HTML file and add the OpenLayers dependencies. Add the following CSS code required to redefine some aspects of the controls we are going to use:

```
<style>
    .olControlNavToolbar {
        top: 0px;
        left: 0px;
        float: left;
    }
    .olControlNavToolbar div {
        float: left;
    }
</style>
```

2. Now, add the HTML code to place the two controls above the map:

```
<table>
    <tr>
        <td>
            Navigation: <div id="navigation"
class="olControlNavToolbar"></div>
        </td>
        <td>
            Position: <div id="mouseposition" style="font-size:
smaller;"></div>
        </td>
    </tr>
</table>

<div id="ch05_control_outside" style="width: 100%; height:
90%;"></div>
```

3. Create the map instance and add a base layer:

```
<script type="text/javascript">
    // Create map
    var map = new OpenLayers.Map("ch05_control_outside");
    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);
    map.setCenter(new OpenLayers.LonLat(0, 0), 3);
```

4. Now, add the mouse position and navigation toolbar controls:

```
    var mousePosition = new OpenLayers.Control.MousePosition({
        div: document.getElementById('mouseposition')
    });
    map.addControl(mousePosition);

    var navToolbarControl = new OpenLayers.Control.NavToolbar({
        div: document.getElementById("navigation")
    });
    map.addControl(navToolbarControl);
</script>
```

How it works...

The previous code seems pretty simple. We have added two controls to our map: an `OpenLayers.Control.MousePosition` control, which shows the current coordinates of the mouse on the map, and `OpenLayers.Control.NavToolbar`.

The `OpenLayers.Control.NavToolbar` control is nothing more than a panel control that contains other controls: an `OpenLayers.Control.Navigation` control, the hand icon (to move the map), and an `OpenLayers.Control.ZoomBox` control, the magnifying glass icon (to zoom on a given box).

So, where is the secret in the recipe for placing the controls outside the map? The answer is in the construction of each control.

The base class `OpenLayers.Control` has a `div` property that points to the `div` element that will be used to hold the control. By default, no `div` element is specified in the constructor, so the control creates a new one to be used.

If you specify a `div` element in the control instantiation, then it is used as the place where the control will be rendered.

For the MousePosition control, we have used the following code:

```
Position: <div id="mouseposition" style="font-size: smaller;"></div>
…
var mousePosition = new OpenLayers.Control.MousePosition({
    div: document.getElementById('mouseposition')
});
```

This means we are placing the control on the previously created `div` element, identified by the `mouseposition` string.

For the navigation toolbar, it differs a bit:

```
Navigation: <div id="navigation" class="olControlNavToolbar"></div>
...
var navToolbarControl = new OpenLayers.Control.NavToolbar({
    div: document.getElementById("navigation")
});
```

In this case we have set the CSS class `olControlNavToolbar`, defined by OpenLayers. Why?

When we do not specify the `div` property, the control creates one and applies some default CSS classes that set the control icon, borders, background color, and so on. Remove the `div` property from the navigation toolbar and see the results. A `div` element will be created and placed on the map with some classes, such as `olControlNavToolbar`, attached to it and will contain some other elements representing the buttons for the pan and zoom actions.

When we specify the `div` property to be used, no style is automatically created and, because of this, controls can disappear or not be rendered nicely if we do not specify some CSS.

Once this is clear, we can say we have not used the CSS class with the mouse position control because it only contains some text. Well, we have only set the font size.

The navigation control is a more complex control, it contains two other controls and we need to tune up its style a bit.

As we will see in *Chapter 6*, *Theming*, most of the OpenLayers flexibility when working with controls is due to the use of the CSS classes. All the controls have, by default, a CSS class associated that defines its position, icons, color, and so on.

In the CSS code that we have set at the beginning of the recipe, we are redefining the place of the navigation toolbar within the `div` and indicating that we want the contained elements, buttons, and flows in the `left` direction:

```
<style>
    .olControlNavToolbar {
        top: 0px;
        left: 0px;
        float: left;
    }
    .olControlNavToolbar div {
        float: left;
    }
</style>
```

See also

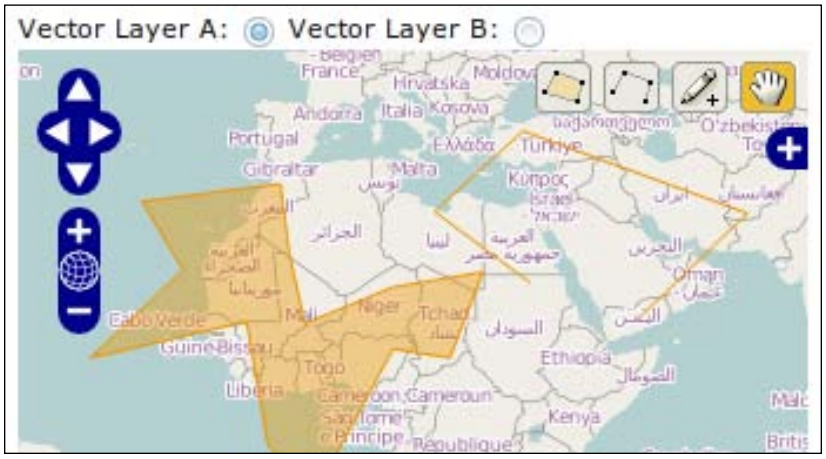
- *Chapter 6*, *Theming*

Editing features on multiple vector layers

When working with vector information, most probably, one of the most common things we can do in a GIS application is: add new features.

OpenLayers has plenty of controls, so there is no need to reinvent the wheel. We have a set of tools and the only thing we need to do is learn how to use each one.

For this concrete purpose, add new features. OpenLayers has the `OpenLayers.Control.EditingToolbar` control that shows a toolbar with some buttons to add polygons, polylines, and points:



Because we can have many vector layers in the map, the control needs us to specify the layer it must work on.

In addition to showing how easy is to use the control, the goal of this recipe is to show how we can use the same control to add features to more than one layer.

This way, this little application will consist of a map with two vector layers. Thanks to the radio buttons, we will be able to chose the layer on which we want to create the new features.

How to do it...

1. First, add the HTML code to create a couple of radio buttons that will allow us to select the vector layer on which we want to draw:

```
<form action="">
    Vector Layer A: <input id="rbA" type="radio" dojoType="dijit.
form.RadioButton" onChange="layerAChanged" name="layer"
value="layerA" checked/>
    Vector Layer B: <input id="rbB" type="radio" dojoType="dijit.
form.RadioButton" onChange="layerBChanged" name="layer"
value="layerB"/>
</form>
<div id="ch05_editing_vector" style="width: 100%; height:
100%;"></div>
```

2. Now, create a map instance and add a base layer:

```
<script type="text/javascript">
    // Create map
    var map = new OpenLayers.Map("ch05_editing_vector");
    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);
    map.addControl(new OpenLayers.Control.LayerSwitcher());
    map.setCenter(new OpenLayers.LonLat(0, 0), 3);
```

3. Add the two vector layers:

```
    var vectorLayerA = new OpenLayers.Layer.Vector("Vector layer
A");
    var vectorLayerB = new OpenLayers.Layer.Vector("Vector layer
B");
    map.addLayers([vectorLayerA, vectorLayerB]);
```

4. Add the editing toolbar control, initially associated to the first vector layer:

```
    var editingToolbarControl = new OpenLayers.Control.
EditingToolbar(vectorLayerA);
    map.addControl(editingToolbarControl);
```

5. Finally, implement the code to handle the radio button changes. It will change the layer associated to the editing toolbar control:

```
    function layerAChanged(checked) {
        if(checked) {
            var controls = editingToolbarControl.
getControlsByClass("OpenLayers.Control.DrawFeature");
            for(var i=0; i< controls.length; i++) {
                controls[i].layer = vectorLayerA;
            }
        }
    }
    function layerBChanged(checked) {
        if(checked) {
            var controls = editingToolbarControl.
getControlsByClass("OpenLayers.Control.DrawFeature");
            for(var i=0; i< controls.length; i++) {
                controls[i].layer = vectorLayerB;
            }
        }
    }
</script>
```

How it works...

The use of the `OpenLayers.Control.EditingToolbar` control has not much mystery. In the constructor, we need to indicate the vector layer we want to add the new features to.

The control will show some buttons on top of the map, allowing us to create new polygons, polylines, or points. Those new features will be added to the specified layer.

So, the secret to add features to other vector layers is about how to change the layer referenced by the control.

The `OpenLayers.Control.EditingToolbar` control is nothing more than a panel that contains four controls. We encourage the reader to take a look at its `initialize` method.

The editor toolbar contains a navigation control, which is represented by the hand icon, and three instances of the `OpenLayers.Control.DrawFeature` control.

The DrawFeature control is the essence of the editor toolbar control. Given a vector layer and a handler, the control allows drawing features on the layer.

As we mentioned at the beginning of this chapter, controls are closely related to handlers. Here, we can see how handlers are responsible for detecting the mouse events and translate it to the point, path, or polygon creation events. On the other side, the draw feature control listens for these events and creates the appropriate map features.

Let's summarize the key points:

- The editor toolbar, as a panel, contains a list of controls: one Navigation control and three DrawFeature controls
- In addition, the EditingToolbar control needs a reference to the vector layer to be edited
- The vector layer reference is passed to the three DrawFeature controls, so they can add new features on the layer

Now, we can see that by changing the layer reference in the draw feature controls, we change the layer where features are added. And this is exactly what the functions that listen for radio buttons' events do:

```
    function layerAChanged(checked) {
        if(checked) {
            var controls = editingToolbarControl.
getControlsByClass("OpenLayers.Control.DrawFeature");
            for(var i=0; i< controls.length; i++) {
                controls[i].layer = vectorLayerA;
            }
        }
    }
```

As we can see in the code, from the editing toolbar we get all the draw feature controls with the call to the `getControlsByClass` method, and then for each one we change the reference to the layer by changing the `layer` property.

There's more...

If we look at the code of the `initialize()` method of the `OpenLayers.Control.EditingToolbar` class:

```
          var controls = [
            new OpenLayers.Control.DrawFeature(layer, OpenLayers.
Handler.Point, {'displayClass': 'olControlDrawFeaturePoint'}),
            new OpenLayers.Control.DrawFeature(layer, OpenLayers.
Handler.Path, {'displayClass': 'olControlDrawFeaturePath'}),
            new OpenLayers.Control.DrawFeature(layer, OpenLayers.
Handler.Polygon, {'displayClass': 'olControlDrawFeaturePolygon'})
          ];
```

We can see it is passing a `displayClass` property to the `OpenLayers.Control.DrawFeature` controls.

This property is also common to all controls inherited from the `OpenLayers.Control` class, and specifies the CSS class that must be applied to the `div` element that will be used to draw the control.

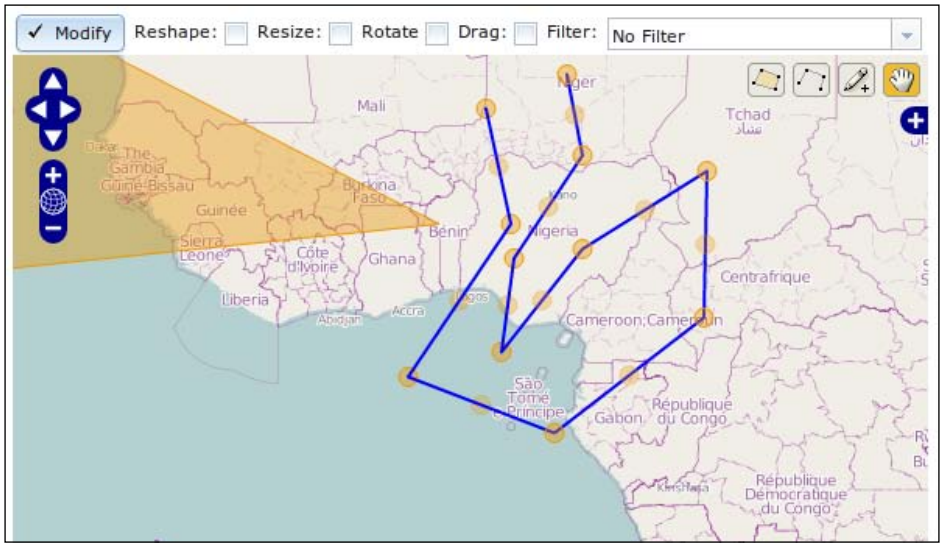
See also

- The *Placing controls outside the map* recipe
- The *Modifying features* recipe

Modifying features

When working on the web mapping application, most probably, the capability to allow the users to add new features would be a desired requirement, but what about modifying features such as move vertex, rotate features, scale, and so on?

Again, OpenLayers simplifies our lives as developers, giving us the powerful `OpenLayers.Control.ModifyFeature` control:



This time we are going to create a little application that will provide us with two important controls: first, to add new features and second, to modify them. For this purpose, we will use the `OpenLayers.Control.EditingToolbar` and `OpenLayers.Control.ModifyFeature` controls.

In concrete, we will see how we can reshape, resize, rotate, and drag features. In addition, we will see how to filter what kind of features can be affected by the modifications.

How to do it...

1. Let's start with creating the controls required for managing the control to modify a feature:

```
<form action="">
    <button data-dojo-type="dijit.form.ToggleButton" data-dojo-
props="iconClass:'dijitCheckBoxIcon', checked: false, onChange:
modifyChanged">Modify</button>
    Reshape: <input id="reshape" dojoType="dijit.form.
CheckBox"onChange="changeMode" name="layer"/>
    Resize: <input id="resize" dojoType="dijit.form.CheckBox"
onChange="changeMode" name="layer"/>
    Rotate <input id="rotate" dojoType="dijit.form.CheckBox"
onChange="changeMode" name="layer"/>
    Drag: <input id="drag" dojoType="dijit.form.CheckBox"
onChange="changeMode" name="layer"/>
```

```
    Filter: <select dojoType="dijit.form.Select" id="filter"
onChange="changeFilter" name="filter" style="width: 200px;">
        <option value="ALL" selected>No Filter</option>
        <option value="POINT">POINT</option>
        <option value="PATH">PATH</option>
        <option value="POLYGON">POLYGON</option>
    </select>
</form>
```

2. Add the element to hold the map:

```
<div id="ch05_modify" style="width: 100%; height: 100%;"></div>
```

3. Start the JavaScript coding by initializing the map and adding a base layer:

```
<script type="text/javascript">
    // Create map
    var map = new OpenLayers.Map("ch05_modify");
    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);
    map.addControl(new OpenLayers.Control.LayerSwitcher());
    map.setCenter(new OpenLayers.LonLat(0, 0), 3);
```

4. Now, add a vector layer to add and modify its features:

```
    var vectorLayer = new OpenLayers.Layer.Vector("Vector layer");
    map.addLayer(vectorLayer);
```

5. Attach an editing toolbar control to the previous layer and add it to the map:

```
    var editingToolbarControl = new OpenLayers.Control.
EditingToolbar(vectorLayer);
    map.addControl(editingToolbarControl);
```

6. Similarly, attach a ModifyFeature control to the vector layer and add it to the map:

```
    var modifyControl = new OpenLayers.Control.
ModifyFeature(vectorLayer);
    map.addControl(modifyControl);
```

7. Add the listener functions that modify the behavior of the modify feature control. First add the function that activates or deactivates the control:

```
    function modifyChanged(checked) {
        if(checked) {
            modifyControl.activate();
        } else {
            modifyControl.deactivate();
        }
    }
```

8. Next, add the function that changes the way the modifications are made:

```
function changeMode() {
    var reshape = dijit.byId("reshape").get("checked");
    var resize = dijit.byId("resize").get("checked");
    var rotate = dijit.byId("rotate").get("checked");
    var drag = dijit.byId("drag").get("checked");

    var mode = null;
    if(reshape) {
        mode |= OpenLayers.Control.ModifyFeature.RESHAPE;
    }
    if(resize) {
        mode |= OpenLayers.Control.ModifyFeature.RESIZE;
    }
    if(rotate) {
        mode |= OpenLayers.Control.ModifyFeature.ROTATE;
    }
    if(drag) {
        mode |= OpenLayers.Control.ModifyFeature.DRAG;
    }

    modifyControl.deactivate();
    modifyControl.mode = mode;
    modifyControl.activate();
}
```

9. Finally add the function to filter the type of geometries the control affects:

```
function changeFilter(value) {

    modifyControl.deactivate();
    map.removeControl(modifyControl);
    modifyControl.destroy();

    var geometryTypes = null;
    if(value=="POINT") {
        geometryTypes = ["OpenLayers.Geometry.Point"];
    } else if(value=="PATH") {
        geometryTypes = ["OpenLayers.Geometry.LineString"];
    } else if(value=="POLYGON") {
        geometryTypes = ["OpenLayers.Geometry.Polygon"];
    }
    modifyControl = new OpenLayers.Control.
ModifyFeature(vectorLayer, {
        geometryTypes: geometryTypes
```

```
            });
            map.addControl(modifyControl);
            modifyControl.activate();
        }
    </script>
```

How it works...

The editing toolbar allows us to draw points, paths, and polygons. Once some features are added to the layer, we can click on the **Modify** toggle button to activate or deactivate the modify feature control. This action is handled by the `modifyChanged` function:

```
function modifyChanged(checked) {
    if(checked) {
        modifyControl.activate();
    } else {
        modifyControl.deactivate();
    }
}
```

By default, the modify feature control allows to reshape any kind of feature, no matter whether it is a point, path, or polygon, that is we can move or add a new vertex to the feature.

With the checkboxes, we can modify the behavior of the control, for example, allowing resizing or dragging of the selected feature.

The function `changeMode` listens for changes on any of the checkboxes and is responsible to modify the action that the control handles.

The action in question is specified through the `mode` property of the control. We can set it at the time of instantiation, or later by modifying the property, as we are doing in this recipe.

In addition, the control allows to handle many actions at a time. We can specify all of them using the logical OR operator. For example:

```
mode = OpenLayers.Control.ModifyFeature.RESHAPE | OpenLayers.Control.
ModifyFeature.RESIZE;
```

As you can see, we force the deactivation and later the activation of the control so that the new `mode` value takes effect.

Finally, we can control one more thing of the control's behavior and that is, the kind of features we can modify. Using the select box, we can choose the kind of geometries that can be modified by the control. The function `changeFilter` is listening for changes on the select box and changes the configuration of the modify feature control.

This geometry filter is done by using the `geometryType` property of the control.

Unfortunately, this property can only be set at instantiation time, changes made later have no effect. So, we need to remove the control from the map and create a new one with the desired geometries to be filtered.

Removing a control from the map does not free the possible resources used by the control. We need to destroy it to free the resources.

There's more...

After reading this recipe, we know how to modify the features. But, what if we want to listen for events while features are being modified? How to know when a modification is going to be made?

The answer is simple, we need to listen for events in the vector layer we are modifying.

Registering for events, such as `beforefeaturemodified`, `featureselected`, or `vertexremoved` allows us to know what exactly is happening and react according to our requirements.

See also

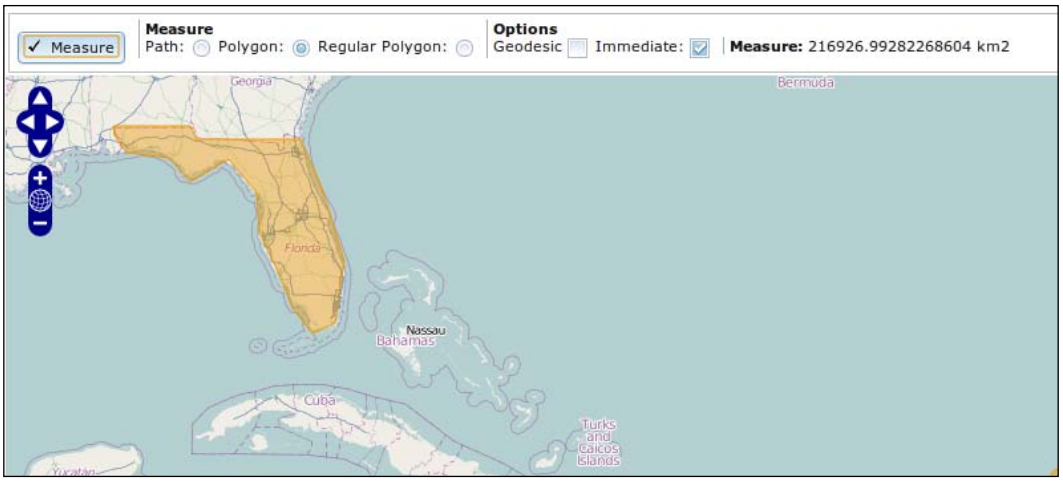
- The *Editing features on multiple vector layers* recipe
- The *Adding and removing controls* recipe
- The *Listening for vector layer features' event* recipe in *Chapter 4*, *Controls*

Measuring distances and areas

The capability to measure distances or areas is an important thing on many GIS applications.

In this recipe, we are going to see in action what the Measure control in OpenLayers offers to the developer.

The application will show a simple map with some buttons on top, as shown in the following screenshot. The **Measure** toggle button activates or deactivates the control, while the radio buttons bring us the possibility to select what to measure: a path or an area:



In addition, we can set two control options. The **Geodesic** control indicates if the distance of the area computation must be in geodesic metrics instead of planar. The **Immediate** option is useful to update the measure every time we move the mouse.

How to do it...

Here, we are going to write not the whole source code but only those pieces of code that are important for the recipe. So, we are avoiding putting here the HTML code required to build the measure button, checkboxes, options radio buttons, and the `div` element that holds the map instance.

1. Let's take a look at the JavaScript code. First, instantiate the map, add a base layer, and center the map display:

```
var map = new OpenLayers.Map("ch05_measure");
var osm = new OpenLayers.Layer.OSM();
map.addLayer(osm);
map.addControl(new OpenLayers.Control.LayerSwitcher());
map.setCenter(new OpenLayers.LonLat(0, 0), 3);
```

2. Now, add the Measure control. Note that we are registering two listener functions for the events `measure` and `measurepartial`:

```
    var measureControl = new OpenLayers.Control.
Measure(OpenLayers.Handler.Path, {
        persist: true,
        eventListeners: {
            'measure': measure,
            'measurepartial': measurepartial
        }
    });
```

3. Next, place the code for the **Measure** toggle button that activates or deactivates the control:

```
function measureClick(checked) {
    var path = dijit.byId('path').get('checked');
    var polygon = dijit.byId('polygon').get('checked');
    var regular = dijit.byId('regular').get('checked');

    if(checked){
        if(path) {
            measureControl.updateHandler(OpenLayers.Handler.
Path, {persist: true});
        } else if(polygon) {
            measureControl.updateHandler(OpenLayers.Handler.
Polygon, {persist: true});
        } else if(regular) {
            measureControl.updateHandler(OpenLayers.Handler.
RegularPolygon, {persist: true});
        }
        map.addControl(measureControl);
        measureControl.activate();
    } else {
        measureControl.deactivate();
        map.removeControl(measureControl);
    }

    dojo.byId('value').innerHTML = "";
}
```

4. Implement the listener functions for the `measure` and `measurepartial` control events:

```
function measure(event) {
    var message = event.measure + " " + event.units;
    if(event.order>1) {
        message += "2";
    }
    dojo.byId('value').innerHTML = message;
}

function measurepartial(event) {
    var message = event.measure + " " + event.units;
    dojo.byId('value').innerHTML = message;
}
```

5. Finally, place the code for the functions that change the **Geodesic** and **Immediate** options:

```
function changeImmediate(checked) {
    measureControl.setImmediate(checked);
}
function changeGeodesic(checked) {
    measureControl.geodesic = checked;
}
```

How it works...

Let's start analyzing how we initialized the measure control:

```
    var measureControl = new OpenLayers.Control.Measure(OpenLayers.
Handler.Path, {
        persist: true,
        eventListeners: {
            'measure': measure,
            'measurepartial': measurepartial
        }
    });
```

The only parameter we need to pass to the control is a handler to be used.

Like many other controls, the `OpenLayers.Control.Measure` class makes use of handlers to interact with the map. In this case, the measure control can make use of any handler that allows to draw geometries. To summarize, the flow is as follows:

- The control is activated and it delegates to a handler the task of drawing some geometry in the map
- Once the handler has drawn the desired geometry, (such as a path or a polygon) the feature is returned to the control
- The control computes the distance or area of the geometry and triggers an event

Actually, we have a limitation to use handlers that return geometries that implement the `getArea()` or `getLength()` methods. For example, if you try to use the `OpenLayers.Handler.Box` handler with the measure control, once you activate the control and draw a box, you will get an error in the browser console. This is because the box handler returns an `OpenLayers.Bounds` instance that neither has a `getLength` nor `getArea` method.

In our code we have initialized the measure control setting as follows:

- The `persist` property to `true`. This property indicates that the geometry created by the handler must remain on the map until a new measure starts.
- Two event listeners, for the events `measure` and `measurepartial`. The `measure` event is triggered once the measure action has been finished. The `measurepartial` is triggered on any measure update (only if the `immediate` property is true).

When the **Measure** toggle button is pressed, the `measureClick` function is executed. This function checks what kind of handler must be used for the measurement and sets it on the control.

This can be done by the `updateHandler` method on the Measure control. For example:

```
measureControl.updateHandler(OpenLayers.Handler.Polygon, {persist:
true});
```

In addition, the `measureClick` function adds the control to the map and activates when the button is toggled on, or deactivates and removes the control from the map when the button is toggled off.

For the control options buttons, we have set two listening functions associated to the checkboxes.

When the **Immediate** checkbox changes, the `changeImmediate` function is executed. This, using the `setImmediate` method, changes the `immediate` property of the control, which allows triggering events every time the measure updates with a mouse movement:

```
function changeImmediate(checked) {
    measureControl.setImmediate(checked);
}
```

The **Geodesic** checkbox sets the value of the `geodesic` property. This time we can modify the property directly without the need of a setter function:

```
function changeGeodesic(checked) {
    measureControl.geodesic = checked;
}
```

With the `geodesic` property set to `true`, the control will use a geodesic metric instead of a planar metric to compute the measures.

There's more...

An important part of the measurement is done with the geometric instances.

All the geometry classes, such as `OpenLayers.Geometry.Polygon` or `OpenLayers.Geometry.LineString`, contain methods to compute their area or length.

Looking at the measure control source code, we can see how once its associated handler returns a geometry, it simply calls the geometry methods to get the area or length and triggers an event.

See also

- The *Working with geolocation* recipe
- The *Editing features on multiple vector layers* recipe
- The *Modifying features* recipe

Getting feature information from data source

We work on web mapping applications almost every day. We know how to create a map and add raster and vector layers. More than that, we know how to get vector data from different data sources: GeoJSON file, KML file, or from a WFS server.

At this point, and related to vector layers, one of the possible questions we could have is: how can we retrieve the feature's information? Fortunately, OpenLayers offers us some controls that can answer this question.

In this recipe, we are going to see in action the `OpenLayers.Control.GetFeature` control class that has the ability to query the feature's data source.

We are going to create a map with a base layer and two vector layers. One from a WFS server, with the USA, and the other from a GML file with Europe's countries.

On top of the map, a button allows us to activate/deactivate the `GetFeature` control and two radio buttons allow us to select between the USA or Europe layers to be queried.

How to do it...

1. Let's start creating the HTML file with the OpenLayers dependencies. Then add the HTML code for the map and buttons:

```
<button dojoType="dijit.form.ToggleButton" id="getfeatureButton"
onChange="getFeatureClick" iconClass='dijitCheckBoxIcon'
checked="false">Activated</button>
Get Information from:
USA <input id="usa" dojoType="dijit.form.RadioButton"
onChange="changeHandler" checked name="layer"/>
Europe <input id="europe" dojoType="dijit.form.RadioButton"
onChange="changeHandler" name="layer"/>

<div id="ch05_getfeature" style="width: 100%; height: 100%;"></
div>
```

2. In the JavaScript section, set the proxy script to be used:

```
OpenLayers.ProxyHost = "./utils/proxy.php?url=";
```

3. Initialize the map and add a base layer:

```
var map = new OpenLayers.Map("ch05_getfeature");
// Add a WMS layer
var wms = new OpenLayers.Layer.WMS("Basic", "http://labs.
metacarta.com/wms/vmap0",
{
    layers: 'basic'
});
map.addLayer(wms);
map.addControl(new OpenLayers.Control.LayerSwitcher());
map.setCenter(new OpenLayers.LonLat(0, 40), 3);
```

4. Add the two vector layers, the first from a WFS server and the second from a GML file:

```
var statesLayer = new OpenLayers.Layer.Vector("States", {
    protocol: new OpenLayers.Protocol.WFS({
        url: "http://demo.opengeo.org/geoserver/wfs",
        featureType: "states",
        featureNS: "http://www.openplans.org/topp"
    }),
    strategies: [new OpenLayers.Strategy.BBOX()]
});
map.addLayer(statesLayer);

var europeLayer = new OpenLayers.Layer.Vector("Europe (GML)",
{
    protocol: new OpenLayers.Protocol.HTTP({
        url: "http://localhost:8080/openlayers-cookbook/
recipes/data/europe.gml",
        format: new OpenLayers.Format.GML()
    }),
    strategies: [new OpenLayers.Strategy.Fixed()]
});
map.addLayer(europeLayer);
```

5. Add a third layer that will serve to show the selected features:

```
var selected = new OpenLayers.Layer.Vector("Selected", {
    styleMap: new OpenLayers.Style(OpenLayers.Feature.Vector.
style["temporary"])
});
map.addLayer(selected);
```

6. Now add the `GetFeature` control:

```
var getFeature = new OpenLayers.Control.GetFeature({
    protocol: statesLayer.protocol,
    box: true,
    hover: false,
    multipleKey: "shiftKey",
    toggleKey: "ctrlKey",
    eventListeners: {
        "featureselected": function(event) {
            selected.addFeatures([event.feature]);
        },
        "featureunselected": function(event) {
            selected.removeFeatures([event.feature]);
        }
    }
});
map.addControl(getFeature);
```

7. Insert the code to activate/deactivate the control:

```
function getFeatureClick(checked) {
    if(checked) {
        getFeature.activate();
    } else {
        getFeature.deactivate();
    }
}
```

8. And finally, add the code that will change which layer to be queried by the control:

```
function changeHandler() {
    var usa = dijit.byId('usa').get('checked');
    if(usa) {
        getFeature.protocol = statesLayer.protocol;
    } else {
        getFeature.protocol = europeLayer.protocol;
    }
}
```

How it works...

Because we are working with the WFS layer, and also the later queries made by the `GetFeature` control are using AJAX, we need to configure a proxy script to be used.

After initializing the map, we have added a base WMS layer simply using the `OpenLayers.Layer.WMS` class and specifying a URL to the server and the WMS layer name:

```
        var wms = new OpenLayers.Layer.WMS("Basic", "http://labs.
    metacarta.com/wms/vmap0",
        {
            layers: 'basic'
        });
```

The two vector layers have not much secret. Thanks to the `OpenLayers.Protocol` subclasses, we can easily create vector layers from different data sources by simply specifying the right protocol:

```
        var statesLayer = new OpenLayers.Layer.Vector("States", {
            protocol: new OpenLayers.Protocol.WFS(...),
            strategies: [new OpenLayers.Strategy.BBOX()]
        });

        var europeLayer = new OpenLayers.Layer.Vector("Europe (GML)", {
            protocol: new OpenLayers.Protocol.HTTP(...),
            strategies: [new OpenLayers.Strategy.Fixed()]
        });
```

In addition, we have created a third vector layer called `selected`. Why? Let's explain first how the `OpenLayers.Control.GetFeature` works:

```
        var getFeature = new OpenLayers.Control.GetFeature({
            protocol: statesLayer.protocol,
            box: true,
            hover: false,
            multipleKey: "shiftKey",
            toggleKey: "ctrlKey",
            eventListeners: {
                "featureselected": function(event) {
                    selected.addFeatures([event.feature]);
                },
                "featureunselected": function(event) {
                    selected.removeFeatures([event.feature]);
                }
            }
        });
```

By default, `OpenLayers.Control.GetFeature` starts working when a click event is made. Then, using the specified `protocol` instance, it queries the data source for the features under the click location.

In addition, by using the `box` property, we can allow selecting features using a selection binding box. A third selection is done with the hover action but we have disabled it setting the `hover` property to `false`.

As the name implies, the properties `multipleKey` and `toggleKey` are used to define the control keys that are used to select multiple features and toggle their selection state.

Finally, the `eventListeners` property allows us to register at the time the constructor is called and the events we want to listen for. In this case we will be notified when a feature will be selected or unselected.

Let's go back to the third vector layer, the `selected` layer.

In contrast to the other controls, such as the `OpenLayers.Control.SelectFeature`, `OpenLayers.Control.GetFeature` didn't modify the visual style of the feature. That is, if you use the `SelectFeature` control, you will see that each time you select a feature, its color and border will change to indicate it is selected.

On the other hand, with `OpenLayers.Control.GetFeature`, nothing happens when a feature is selected. The control makes a query to the data source and an event is triggered. You are responsible to perform something with that event.

In this recipe, we are retrieving the feature from the event and adding it to the selected layer:

```
"featureselected": function(event) {
    selected.addFeatures([event.feature]);
},
"featureunselected": function(event) {
    selected.removeFeatures([event.feature]);
}
```

If we look at the instantiation code of the selected layer, it looks something like the following code:

```
    var selected = new OpenLayers.Layer.Vector("Selected", {
        styleMap: new OpenLayers.Style(OpenLayers.Feature.Vector.
style["temporary"])
    });
```

We have supplied a different style for the layer. In this case, we are getting the `"temporary"` style defined at `OpenLayers.Feature.Vector` and applying it to the layer so the features have a different look.

More on styling is discussed in *Chapter 7*, *Styling Features*.

There's more...

It is important to note that the `OpenLayers.Control.SelectFeature` control is similar to the `OpenLayers.control.GetFeature` control, but they have important differences.

First, the `OpenLayers.control.GetFeature` control makes a query to the data source that returns the features involved by the selection. The `openLayers.Control.SelectFeature` control makes no request, it works on the client-side and retrieves the selected feature from the specified vector layer.

Second, every time a feature is selected with the `OpenLayers.Control.SelectFeature` control, a `featureselected` event is triggered by the vector layer. So we can register listeners in the vector layer to be notified for the selection events.

With the `OpenLayers.control.GetFeature` control, no event is triggered by the vector layer. The events are triggered by the control and because of this we need to register listeners in the control.

Finally, with the `OpenLayers.control.GetFeature` control, we can use any protocol that supports spatial filters. Because of this, we can use the `GetFeature` control against a WFS server or a GML file.

See also

- The *Selecting and transforming features* recipe
- The *Getting information from the WMS server* recipe
- The *Adding GML layer* recipe in *Chapter 3*, *Vector Layers*
- The *Filtering features in WFS requests* in *Chapter 3*, *Vector Layers*
- The *Listening for vector layer features' event* in *Chapter 4*, *Controls*

Getting information from the WMS server

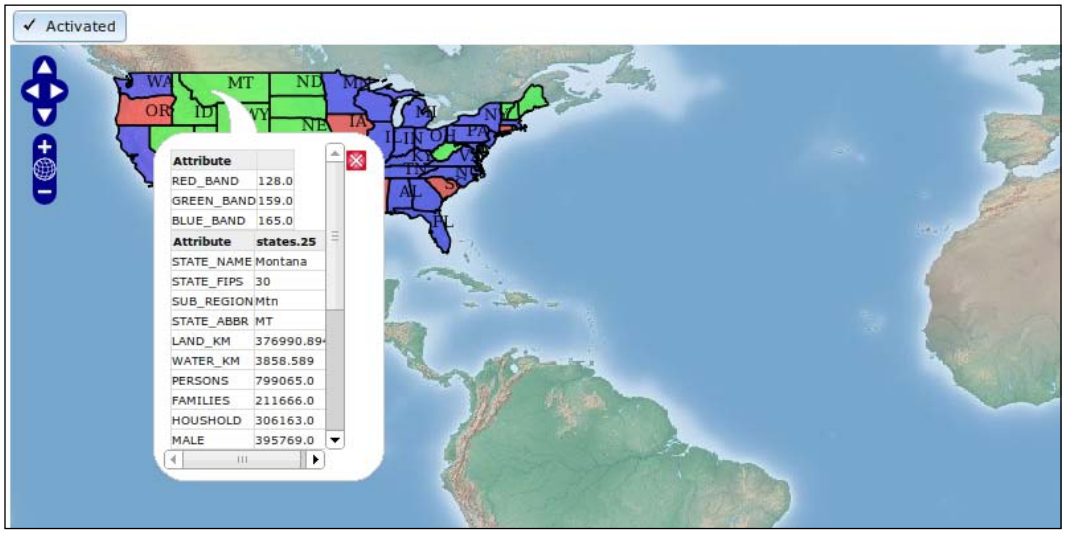
Nowadays, the **Web Map Service** (**WMS**) has an important role in the GIS world, mainly because rendering tons of vector data at the client-side, no matter if its browser on a desktop consumes many resources.

If we think, in the OpenStreetMap project, where we have tons of vector data about streets, places, and so on, we can see that the main way to render data is in a raster way.

In this scenario, WMS servers allow us to get vector or raster data, from a shapefile, from a `.geotiff` file, from a spatial database, and so on, and render all together as a single image. Not only that, if properly configured, a WMS server allows us to query information of a feature at a given point.

With OpenLayers, this can be easily done using the `OpenLayers.Control.WMSGetFeatureInfo` control.

In the following screenshot, we can see what our current recipe looks like. Given some vector information about USA states, the server returns a raster image.



Once the control is activated, any click event on the map will trigger a request to the WMS server to get the feature information.

How to do it...

1. Create an HTML file with the OpenLayers library dependencies and add the code for the button and the map's `div` element:

```
<button dojoType="dijit.form.ToggleButton" id="featureInfoButton"
onChange="featureInfoChange" iconClass='dijitCheckBoxIcon'
checked="false">Activated</button>
<div id="ch05_wmsfeatureinfo" style="width: 100%; height:
100%;"></div>
```

2. Set the proxy script, and initialize the map instance:

```
OpenLayers.ProxyHost = "./utils/proxy.php?url=";
// Create map
var map = new OpenLayers.Map("ch05_wmsfeatureinfo");
```

3. Now, add two WMS layers. The first will act as the base layer while the second will be an overlay layer with the USA states:

```
    var wms = new OpenLayers.Layer.WMS("Basic", "http://demo.
opengeo.org/geoserver/wms",
    {
        layers: 'topp:naturalearth'
    });
    map.addLayer(wms);
     var wms2 = new OpenLayers.Layer.WMS("Basic", "http://demo.
opengeo.org/geoserver/wms",
    {
        layers: 'topp:states',
        transparent: true
    },{
        isBaseLayer: false
    });
    map.addLayer(wms2);
```

4. Add the layer switcher control and center the map's viewport:

```
    map.addControl(new OpenLayers.Control.LayerSwitcher());
    map.setCenter(new OpenLayers.LonLat(-90, 40), 4);
```

5. Then add the code for the `WMSGetFeatureInfo` control:

```
    var featureInfo = new OpenLayers.Control.WMSGetFeatureInfo({
        url: 'http://demo.opengeo.org/geoserver/wms',
        title: 'Identify features by clicking',
        queryVisible: true,
        eventListeners: {
            "getfeatureinfo": function(event) {
                map.addPopup(new OpenLayers.Popup.FramedCloud(
                "chicken",
                map.getLonLatFromPixel(event.xy),
                null,
                event.text,
                null,
                true
            ));
            }
        }
    });
    map.addControl(featureInfo);
```

6. Finally, add the code to activate/deactivate the control when the button is clicked:

```
function featureInfoChange(checked) {
    if(checked) {
        featureInfo.activate();
    } else {
        featureInfo.deactivate();
    }
}
```

How it works...

A WMS server implements different request types. The most important is the `GetMap` request, which allows us to get an image given some parameters, such as a bounding box, the name of the layers, and so on.

All this explanation is more close to understanding the WMS standard than working with OpenLayers. So, invest your time and learn what the WMS standard offers and how it works. You can find a very brief description at wikipedia: `http://en.wikipedia.org/wiki/Web_Map_Service`, and the whole specification at OGC: `http://www.opengeospatial.org/standards/wms`.

In addition, the WMS server can implement the `GetFeatureInfo` request. This type of request allows us to, given a point and some layer names configured at the WMS server, retrieve information from a feature, that is, we can get a feature attribute from a layer which is rendered as a raster image.

Let's describe the code of this recipe, which is the goal of this book and not to explain how a WMS server works.

Because, the control will make an AJAX request, we need to set a proxy script to be used:

```
OpenLayers.ProxyHost = "./utils/proxy.php?url=";
```

The WMS layer comes from a public server from the awesome **OpenGeo** project (`http://opengeo.org`). The first layer acts as a base layer. The second one must be an overlay layer, because we have set the `isBaseLayer` property to `false`. In addition, to avoid the layer hiding the base layer, we have set the `transparent` property, which is used in the WMS request, to `true`:

```
 var wms2 = new OpenLayers.Layer.WMS("Basic", "http://demo.
opengeo.org/geoserver/wms",
    {
        layers: 'topp:states',
        transparent: true
```

```
},{
    isBaseLayer: false
});
```

Adding the `WMSGetFeatureInfo` control is easy, we need to set the WMS server URL, some desired properties and register some event listeners to make something with the returned information:

```
var featureInfo = new OpenLayers.Control.WMSGetFeatureInfo({
    url: 'http://demo.opengeo.org/geoserver/wms',
    queryVisible: true,
    eventListeners: {...}
});
```

Because we want to show a popup with the data, we have registered a function on the `getfeatureinfo` event, which is triggered when the control obtains the server data:

```
        eventListeners: {
            "getfeatureinfo": function(event) {
                map.addPopup(new OpenLayers.Popup.FramedCloud(
                "chicken",
                map.getLonLatFromPixel(event.xy),
                null,
                event.text,
                null,
                true
            ));
            }
        }
```

To query information of a layer in a WMS server, it must be configured as a queryable layer. If we request for a layer which is not queryable, then a `nogetfeatureinfo` event will be triggered by the control.

By default, the control requests data for all WMS layers in the map. With the `queryVisible` property, we can limit the query to those layers which are currently visible and forget those hidden layers.

There's more...

The `WMSGetFeatureInfo` control has other interesting properties.

With the `hover` property set to `true` we can force the control to query the server, not only when the mouse clicks on the map, but also on the mouse hover event.

Using the `layers` property, which accepts an array of `OpenLayers.Layer.WMS` layers, we can control which layers must be queried on the server. If not specified, the layers are obtained from the map.

In addition, if a layer has been configured to work with more than one server, only the first one is used for the queries.

Also, it is important to note that a WMS server can return the data in different formats, for example, a plain text, an HTML response, or also in GML format.

With the `infoFormat` property, we can indicate to the server the kind of response we desire. By default it is HTML.

See also

- The *Getting feature information from data source* recipe
- The *Selecting and transforming features* recipe
- The *Adding WMS layer* recipe

6
Theming

In this chapter we will cover:

- Understanding how themes work using the `img` folder
- Understanding how themes work using the `theme` folder
- Delimiting tiles in a raster layer
- Creating a new OpenLayers theme
- Starting actions outside the controls

Introduction

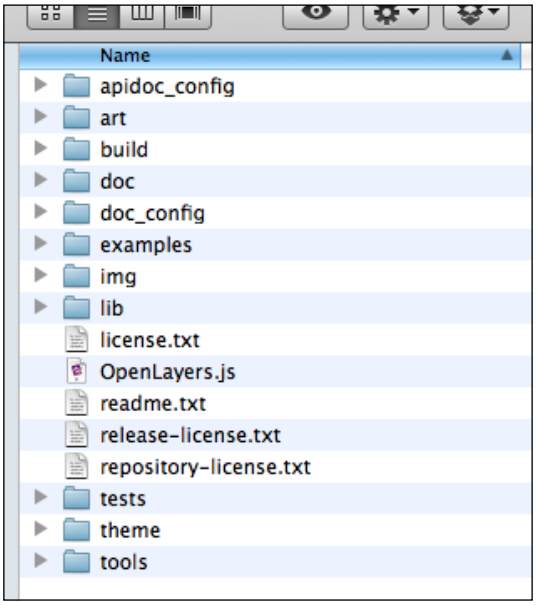
It is worth mentioning that in software applications, the first impressions are the most important things, and they are given by two factors: the look and feel.

This chapter is all oriented to show how we can improve the look and feel of our web mapping application by theming OpenLayers.

As many other web applications, the look and feel of the OpenLayers library is controlled using images and CSS classes, which define the position, dimensions, and visual aspects of any OpenLayers component.

At this moment, with the Version 2.11, we can find the `img` and `theme` folders within the bundle distribution, and both are used to control the look of the OpenLayers applications.

Remember to place these folders in your project when using OpenLayers as mentioned in the *Different ways to include OpenLayers* recipe in *Chapter 1*, *Web Mapping Basics*.



The `theme` folder contains CSS files, as well as some images used within the CSS, while the `img` folder contains only images, used by some controls in a more hardcoded (and not recommended) way.

We can say, the use of the `theme` folder with CSS styles is the preferred way to implement controls, while the use of the `img` folder remains for those controls that are not updated to work with CSS styles.

Becoming a *great* web designer is out of the scope of this book, but it is true that if we want to tune up the OpenLayers appearance a bit, we need to have some knowledge of HTML and how CSS works.

We can find a description of the CSS standard at `http://en.wikipedia.org/wiki/Cascading_Style_Sheets` but we can find tons of great tutorials looking on the Net, such as `http://www.csstutorial.net`.

The browsers work with three main technologies: HTML, CSS, and JavaScript. Summarized in really short sentences, we can say:

- HTML defines the content of a web page using paragraphs, titles, sections, and so on.
- CSS defines the visual aspects of the HTML elements, that is, which text color a paragraph must use, the text size of a title, and so on.
- Finally, JavaScript is a programming language processed by the browser that can be used to manipulate dynamically any aspect of the page. For example, we can add new HTML elements, change CSS, and check if the fields in a form are valid before sending it to the server.

The **Document Object Model** (**DOM**) is a standard for accessing and manipulating HTML documents. We can think it decomposes an HTML document like a tree of elements and attributes.

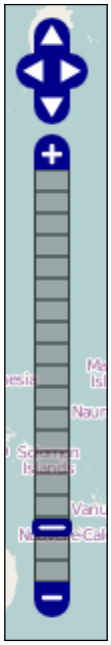
See: `http://www.w3schools.com/htmldom/default.asp`

OpenLayers belongs to the third category. It is a JavaScript library that allows us to create web mapping applications using concepts such as maps, layers, or features, but abstracting us from the HTML DOM elements and the CSS aspects required to render them.

When we create an OpenLayers component, such as a map, a layer, or a control instance, it also creates the required HTML elements to render them and puts them at the right place of the DOM structure of our HTML page.

The goal of this chapter is to show how to theme the most important OpenLayers components. So, using many of the next recipes, we will be able to create a fresh web-mapping application with a customized look.

Understanding how themes work using the img folder



As explained in the *Introduction* section of this chapter, there are controls that are themed simply using the images stored in the `img` folder.

This way is the oldest way to theme a control, and for newer implementations, the preferred way of theming is using CSS, that is, using the `theme` folder.

Until its update, we could make use of controls that work with any of the two forms of theming, so it is important to know how to theme both.

In this recipe, we are going to describe how to theme the `PanZoomBar` control that uses the old way based on the images in the `img` folder.

How to do it...

1. Create an HTML file with OpenLayers dependencies and start adding in the body element of the document, the `div` element to hold the map:

```
<div id="ch06_theming_img" style="width: 100%; height: 90%;"></
div>
```

2. Now, add the following JavaScript code to initialize the map and add a base layer:

```
var map = new OpenLayers.Map("ch06_theming_img ");
var osm = new OpenLayers.Layer.OSM();
map.addLayer(osm);
map.setCenter(new OpenLayers.LonLat(0, 0), 2);
```

3. Finally, simply create an `OpenLayers.Control.PanZoomBar` control instance and add to the map:

```
var panZoomBar = new OpenLayers.Control.PanZoomBar();
map.addControl(panZoomBar);
```

How it works...

When we create the control, what really happens is OpenLayers automatically creates a set of HTML elements and places them in the page's DOM structure.

From the OpenLayers' JavaScript API perspective, it is simply adding a control component to the map but from the HTML code point of view, it means that a complex set of elements is created on the page to represent all the buttons and images required to look like a nice pan and zoom control:

```
<div id="OpenLayers.Control.PanZoomBar_71" style="... left: 4px; top:
4px; ..." class="olControlPanZoomBar olControlNoSelect" ...>
    <div id="OpenLayers.Control.PanZoomBar_71_panup" ...>
        <img id="OpenLayers.Control.PanZoomBar_71_panup_innerImage"
... src="./js/OpenLayers-2.11/img/north-mini.png">
    </div>
    <div id="OpenLayers.Control.PanZoomBar_71_panleft" ...>
        <img id="OpenLayers.Control.PanZoomBar_71_panleft_innerImage"
... src="./js/OpenLayers-2.11/img/west-mini.png">
    </div>
     <div id="OpenLayers.Control.PanZoomBar_71_panright" ...>
        <img id="OpenLayers.Control.PanZoomBar_71_panright_innerImage"
... src="./js/OpenLayers-2.11/img/east-mini.png">
    </div>
     <div id="OpenLayers.Control.PanZoomBar_71_pandown" ...>
        <img id="OpenLayers.Control.PanZoomBar_71_pandown_innerImage"
... src="./js/OpenLayers-2.11/img/south-mini.png">
    </div>
    <div id="OpenLayers.Control.PanZoomBar_71_zoomin" ...>
        <img id="OpenLayers.Control.PanZoomBar_71_zoomin_innerImage"
... src="./js/OpenLayers-2.11/img/zoom-plus-mini.png">
    </div>
</div>
```

Looking at the generated code we can say that:

- The main control element uses a CSS class named `olControlPanZoomBar`
- All the images used are loaded from the OpenLayers `img` folder, such as `img/north-mini.png`, `img/east-mini.png` and `img/zoom-minus-mini.png`
- The position of the control, and its buttons, are set in the `style` attribute instead of using a CSS class

The conclusion is trivial: a change in the look of this control can only be done by changing the images in the `img` folder.

In addition, if we want to place the control at a different position, we need to play with the `OpenLayers.Control.PanZoom.X` and `OpenLayers.Control.PanZoom.Y` properties, which are used to set the value of the `top` and `left` properties in the `style` attribute of the main control HTML element. For example, setting:

```
OpenLayers.Control.PanZoom.X = 50;
```

Produces an HTML code as follows:

```
<div id="OpenLayers.Control.PanZoomBar_71" style="position: absolute;
left: 50px; top: 4px; ...
```

This means, although you redefine the properties `top` and `left` in the `olControlPanZoomBar` class, they will not take effect because the properties specified in the `style` attribute take precedence:

```
.olControlPanZoomBar {
    top: 50px;
    left: 50px;
}
```

There's more...

There are a couple of important things to note.

First, every OpenLayers instance has an ID property. We can set it manually when creating the instance or leave OpenLayers to compute one for us, but take into account that the ID must be unique.

In this case, the ID for the `PanZoomBar` control is the string `OpenLayers.Control.PanZoomBar_71` and it is also used to identify the HTML elements.

Second, if a control makes use of a CSS class, by convention, the name of the class will be `olControl` followed by the name of the control, such as: `olControlPanZoomBar`.

See also

- The *Understanding how themes work using the theme folder* recipe
- The *Adding and removing controls* recipe in *Chapter 5*, *Adding Controls*

Understanding how themes work using the theme folder

As we have explained in the chapter's introduction, there are some OpenLayers controls strongly based on CSS classes to be the theme.

In this group, we can find the `PanPanel` control, a small control formed by a set of four buttons that allows the user to pan the map in four directions:

How to do it...

1. Create an HTML page and add the OpenLayers library dependencies:

```
<script type="text/javascript" src="./js/OpenLayers-2.11/lib/
OpenLayers.js"></script>
```

2. After this, we need to include the CSS file with the theme to be used. Here we are using the default theme:

```
<link rel="stylesheet" href="./js/OpenLayers-2.11/theme/default/
style.css" type="text/css">
```

3. Within the `body` element of the document, add the `div` element to hold the map:

```
<div id="ch06_theming_theme" style="width: 100%; height: 90%;"></
div>
```

4. Within a `script` element, add the code to create the map with a base layer:

```
var map = new OpenLayers.Map("ch06_theming_img ");
var osm = new OpenLayers.Layer.OSM();
map.addLayer(osm);
map.setCenter(new OpenLayers.LonLat(0, 0), 2);
```

5. Finally, create an `OpenLayers.Control.PanPanel` instance and add it to the map:

```
var panControl = new OpenLayers.Control.PanPanel();
map.addControl(panControl);
```

How it works...

When the `OpenLayers.Control.PanPanel` instance is added to the map, what really happens is a set of new HTML elements are added to the DOM page structure:

```
<div id="OpenLayers.Control.PanPanel_71" style="position: absolute;
z-index: 1006; " class="olControlPanPanel olControlNoSelect"
unselectable="on">
    <div class="olControlPanNorthItemInactive"></div>
    <div class="olControlPanSouthItemInactive"></div>
    <div class="olControlPanEastItemInactive"></div>
    <div class="olControlPanWestItemInactive"></div>
</div>
```

There is one main element for the control that contains other elements representing the four buttons.

The main HTML element has an attached CSS class with the name `olControlPanPanel`. This class name is automatically created by OpenLayers and follows this convention: `olControl` plus the control name.

All the CSS classes used in the previous HTML code can be found in the source code in the `theme/default/style.css` theme file.

There's more...

Looking at the CSS classes used by the control, we can understand a bit better how it works.

First, we change the position of the control by modifying the properties of the CSS class:

```
.olControlPanPanel {
    top: 10px;
    left: 5px;
}
```

Next, CSS code sets the image to be used and the size of the buttons:

```
.olControlPanPanel div {
    background-image: url(img/pan-panel.png);
    height: 18px;
    width: 18px;
```

```
        cursor: pointer;
        position: absolute;
    }
```

We can see how the image sprite is taken from the file `theme/default/img/pan-panel.png`:

An **image sprite** is a collection of images put into the same file. Later, using the CSS properties we can get once piece of this image sprite to be used on an element.

Image sprites reduce the number of requests to the server when a page loads.

Next, each button defines the required properties to extract the piece of image to be used as the button:

```
.olControlPanPanel .olControlPanSouthItemInactive {
    top: 36px;
    left: 9px;
    background-position: 18px 0;
}
```

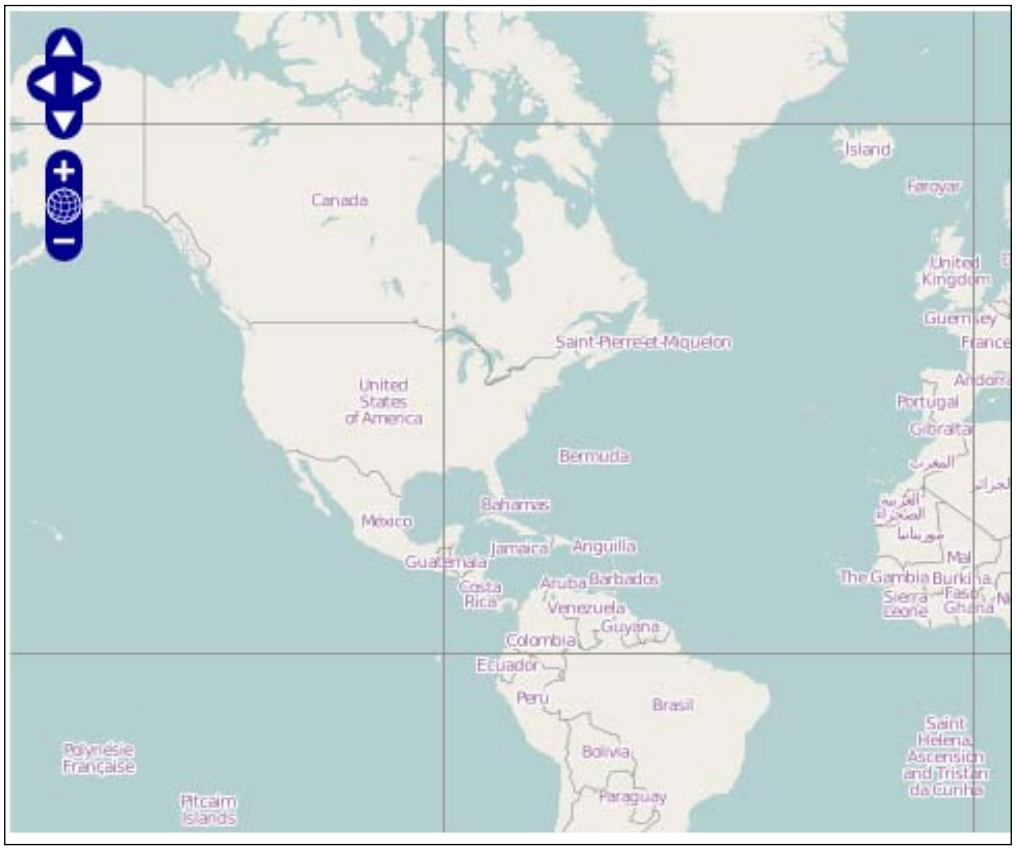
We can see how with little CSS knowledge we can modify almost any desired thing of the control.

See also

- The *Understanding how themes work using the img folder* recipe
- The *Adding and removing controls* recipe in *Chapter 5*, *Adding Controls*
- The *Creating a simple full screen map* recipe in *Chapter 1*, *Web Mapping Basics*
- The *Different ways for including OpenLayers* recipe in *Chapter 1*, *Web Mapping Basics*

Delimiting tiles in a raster layer

To show how easy it is to change the appearance of an element using CSS, in this recipe we are going to add a border to all the tiles from any raster layer to show where the limits of each tile are:



How to do it...

1. Create an HTML file with OpenLayers dependencies and add within the `head` section a `style` element with the following CSS code:

```
<style>
    .olTileImage {
        border: 1px solid #999;
    }
</style>
```

2. Next, in the `body` element of the document, add the `div` element to hold the map:

```
<div id="ch06_tile_borders" style="width: 100%; height: 90%;"></
div>
```

3. Now, add the following JavaScript code to initialize the map and add a base layer:

```
var map = new OpenLayers.Map("ch06_tile_borders");
var osm = new OpenLayers.Layer.OSM();
map.addLayer(osm);
map.setCenter(new OpenLayers.LonLat(0, 0), 2);
```

How it works...

The code to create the map instance and layer is pretty simple, we have simply created an instance of both and added the layer to the map. Finally, we have centered the map's viewport.

Even though it seems incredible, all the magic of this recipe is in the CSS code at the top:

```
.olTileImage {
    border: 1px solid #999;
}
```

Every raster layer class uses images to render the tiles of data. To do so, the layer creates some HTML elements and adds them to the DOM structure as follows:

```
<div id="OpenLayers.Layer.OSM_315" ... class="olLayerDiv">
     <div ...>
        <img id="OpenLayersDiv335" style="position: relative; width:
256px; height: 256px; " class="olTileImage" ...>
    </div>
    <div ...>
        <img id="OpenLayersDiv337" style="position: relative; width:
256px; height: 256px; " class="olTileImage" ...>
    </div>
</div>
```

Every OpenLayers component is transformed in one or more HTML elements that use the CSS classes to define the way they are visualized.

As you can see, a `div` element is created for the whole layer identified by `OpenLayers.Layer.OSM_315`, which has the class parameter set to `olLayerDiv` CSS. Within it we can find `img` elements that point to the tiles to be rendered. These elements have applied the `olTileImage` class.

Thanks to the CSS classes in this recipe, we have set a border on each tile by simply specifying the appropriate property.

See also

- The *Understanding how themes work using the theme folder* recipe

Creating a new OpenLayers theme

There can be situations where we desire a completely different look for the OpenLayers theme.

As we mentioned in the chapter's introduction and other recipes (*Understanding how themes work using the img folder* and *Understanding how themes work using the theme folder*), OpenLayers theming is based on images and CSS files:

In this recipe we are going to see how we can create a new OpenLayers theme based on the default theme we can find in the `theme/default` folder. We are going to change some aspects of the most common controls, such as scale or scale line, overview map, or layer switcher.

Getting ready

To create a new theme, we need to create a replica for the content of the `img` and `theme` folders of the OpenLayers distribution. Both folders contain images to be used in controls, so it is easy to see why a good graphic design is important to create a good theme.

The theme called **green theme**, used in this chapter, is divided into folders `recipes/data/green_img` and `recipes/data/green_theme`.

How to do it...

1. Create an HTML file. In the `head` section we need to attach, in addition to the dependencies to the OpenLayers library, the CSS stylesheets for our custom theme:

```
    <head>
        <title>Creating a new OpenLayers theme</title>
        <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">

        <!-- Include OpenLayers library -->
        <script type="text/javascript" src="../../js/
OpenLayers-2.11/lib/OpenLayers.js"></script>

        <!-- OpenLayers Theme -->
        <link rel="stylesheet" href="../../js/OpenLayers-2.11/
theme/default/style.css" type="text/css">

        <!-- Out customer Green Theme -->
        <link rel="stylesheet" href="../data/green_theme/style.
css" type="text/css">
```

2. Before continuing to see the code at the `head` section, let's write the following code within the `body` section that will hold the map instance and the navigation history tool:

```
    <body onload="init()">
        <div id="ch06_theme" style="width: 100%; height: 100%;"></
div>
        <div id="history" class="historyClass"></div>
    </body>
```

3. Again, within the head section, add the following piece of styling code necessary for the previous `div` element used to hold the navigation history tool:

```
        <style>
            .historyClass {
                position: absolute;
                top: 5px;
                right: 125px;
                z-index: 9999;
            }
        </style>
```

4. Within the `script` section, add the following JavaScript code that specifies where OpenLayers can find the folder `img`:

```
            OpenLayers.ImgPath = "http://localhost:8080/
openlayers-cookbook/recipes/data/green_img/";
```

5. Implement the `init` function, which is executed when the `body` is loaded. It creates the map instance, and adds a base layer and a set of most common controls:

```
            function init() {
                // Create map
                var map = new OpenLayers.Map("ch06_theme", {
                    controls: []
                });
                var osm = new OpenLayers.Layer.OSM();
                map.addLayer(osm);
                map.setCenter(new OpenLayers.LonLat(0, 0), 2);

                var vectorLayer = new OpenLayers.Layer.
Vector("Vector Layer");
                map.addLayer(vectorLayer);

                // Add controls
                map.addControl(new OpenLayers.Control.
Navigation());
                map.addControl(new OpenLayers.Control.
LayerSwitcher({roundedCorner: false}));
                map.addControl(new OpenLayers.Control.
PanZoomBar({zoomWorldIcon: true}));
                map.addControl(new OpenLayers.Control.
MousePosition());
                map.addControl(new OpenLayers.Control.
OverviewMap());
                map.addControl(new OpenLayers.Control.Scale());
                map.addControl(new OpenLayers.Control.
ScaleLine());
                map.addControl(new OpenLayers.Control.
Permalink());
                map.addControl(new OpenLayers.Control.
EditingToolbar(vectorLayer));

                var history = new OpenLayers.Control.
NavigationHistory();
                var panel = new OpenLayers.Control.Panel({
                    div: document.getElementById('history')
                });
                panel.addControls([history.next, history.
previous]);
                map.addControls([history, panel]);
            }
```

How it works...

Let's start briefly by describing the JavaScript code. We have created a map, added a base layer, and finally added a set of controls.

Specially take a look at how we have added the NavigationHistory control, because this is the first themed point of the application:

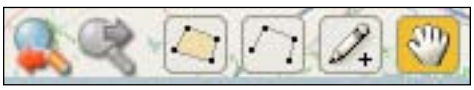
```
var history = new OpenLayers.Control.NavigationHistory();
var panel = new OpenLayers.Control.Panel({
    div: document.getElementById('history')
});
panel.addControls([history.next, history.previous]);
map.addControls([history, panel]);

…
…
<div id="history" class="historyClass"></div>
…
...
```

We have instantiated the control and placed its buttons on a `Panel` control. In addition to rendering the `Panel` control in a specific `div` element of the web page, we have set its `div` property pointing to the desired element.

The CSS `historyClass` is a class that allows us to place the control that is floating on the right-hand side (close to the editing toolbar control):

```
.historyClass {
    position: absolute;
    top: 5px;
    right: 125px;
    z-index: 9999;
}
```

The output will be as shown in the following screenshot:

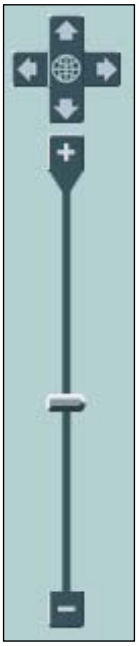


The rest of the theme is based on two important folders.

Because some controls are based on the images contained in the `img` folder, we have set the path to this folder at the beginning of the JavaScript code. Controls such as `PanZoomBar` or the `LayerSwitcher` require the following folder to get their icons:

```
OpenLayers.ImgPath = "http://localhost:8080/openlayers-cookbook/
recipes/data/green_img/";
```

The following screenshot shows the `PanZoomBar` control's icon:



On the other hand, controls such as the `Scale` or `ScaleLine`, the `MousePosition`, or also many aspects of the `LayerSwitcher` control are defined via CSS through the `theme` folder, by its `style.css` file and their images. This is included with a `link` tag in the document's header:

```
<!-- OpenLayers Theme -->
<link rel="stylesheet" href="../../js/OpenLayers-2.11/theme/default/
style.css" type="text/css">

<!-- Out customer Green Theme -->
<link rel="stylesheet" href="../data/green_theme/style.css"
type="text/css">
```

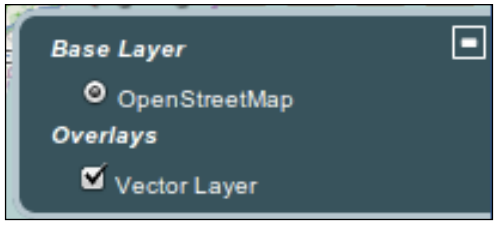
The `green_theme/style.css` file does not contain a complete redefinition of the classes we found in the default theme of OpenLayers. We have simply redefined some classes that affect the color or position of some controls. For this, first we have included the `default/style.css` file and later our custom `green_theme/style.css` file, which only redefines some classes by adding or changing the styles.

There is no magic recipe on how to theme CSS-based controls. We need to check the generated HTML code for the controls and see which CSS classes they use and which other we can apply. Let's see some themed controls.

On the `LayerSwitcher` control, we have changed the font size, the background and border color, added a border radius (only valid for CSS3 compatible browsers), and changed the title for the base and overlay sections to use italics:

```
.olControlLayerSwitcher {
    font-size: x-small;
    font-weight: normal;
}
.olControlLayerSwitcher .layersDiv {
    background-color: #38535c;
    border-radius: 1em;
    border-width: 3px 0 3px 3px;
    border-style: solid;
    border-color: #b6c6ce;
}
.olControlLayerSwitcher .layersDiv .baseLbl,
.olControlLayerSwitcher .layersDiv .dataLbl {
    font-style: italic;
    font-weight: bolder;
}
```

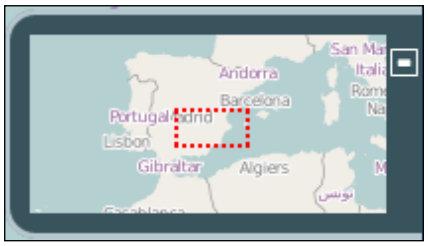
The following screenshot shows the `LayerSwitcher` control with the changes done using the previous code:



For the `OverviewMap` control, we have added a background and border color, a border radius, and moved it a bit to the top to leave space for the following controls:

```
.olControlOverviewMapContainer {
    bottom: 20px;
}
.olControlOverviewMapElement {
    background-color: #38535c;
    border-radius: 1em 0 0 1em;
    border-width: 3px 0 3px 3px;
    border-style: solid;
    border-color: #b6c6ce;
}
```

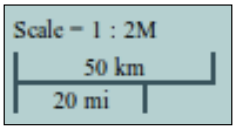
The following screenshot shows the `OverviewMap` control with the changes done using the previous code:



The `Scale` and `ScaleLine` controls have been moved to the left-hand side and the color has been changed to follow the green theme:

```
.olControlScaleLine {
    bottom: 10px;
    font-size: x-small;
}
.olControlScaleLineTop {
    border: solid 2px #38535c;
    border-top: none;
}
.olControlScaleLineBottom {
    border: solid 2px #38535c;
    border-bottom: none;
}
.olControlScale {
    left: 10px;
    bottom: 40px;
    font-size: x-small;
}
```

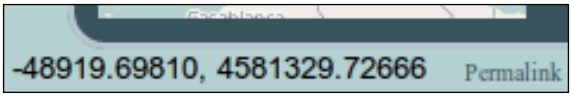
The following screenshot shows the `Scale` and `ScaleLine` controls with the changes done using the previous code:



For the `MousePosition` and `Permalink` controls, we have slightly moved them and changed the text color:

```
div.olControlMousePosition {
    bottom: 5px;
    right: 60px;
    font-size: small;
}

.olControlPermalink {
    bottom: 5px;
    font-size: x-small;
}
.olControlPermalink a {
    color: #38535c;
    text-decoration: none;
}
```

The following screenshot shows the `MousePosition` and `Permalink` controls with the changes done using the previous code:



Finally, for the `NavigationHistory` control, we need to redefine a CSS class so the buttons are ordered horizontally, instead of vertically, which is the default mode:

```
.olControlNavigationHistory {
    float: right;
}
```

There's more...

As we mentioned, there is no easy way to theme a component. We need to take into account the HTML code that will render the component, the images it uses, and the possible CSS styles applied.

The simplest solution to change the look of your mapping applications is to play with the icons and CSS as we did here. More drastic improvements include creating your own controls or placing the controls on external buttons, offered by an external framework such as Dojo (`http://dojotoolkit.org`), jQueryUI (`http://jqueryui.com`), or ExtJS (`http://www.sencha.com/products/extjs`), and writing the required code to activate or deactivate the controls. A good sample of this is the GeoExt project (`http://geoext.org`) that offers rich components based on the previous ExtJS project.

See also

- The *Understanding how themes work using the img folder* recipe
- The *Understanding how themes work using the theme folder* recipe
- The *Adding a navigation history control* recipe in *Chapter 5*, *Adding Controls*
- The *Place controls outside the map* recipe in *Chapter 5*, *Adding Controls*

Starting actions outside the controls

Another different and drastic way to change the look of our applications is to place the controls outside the map and attach them to our own components.

Most of the OpenLayers controls have two features:

- They realize some action (edit features, create a line, and so on)
- They know how to render themselves on top of the map

To achieve the goal of this recipe, the idea is to separate the visualization from the action that the control does. This way, we can create some buttons and activate or deactivate a control depending on the button that is pressed:

As we can see in the screenshot, we are going to create a toolbar and place:

- The same set of controls we can find in `OpenLayers.Control.EditingToolbar`, which will allow us to draw points, lines, and polygons
- A dropdown button that will allow us to start the `OpenLayers.Control.Measure` action

Getting ready

We are going to use the Dojo Toolkit framework (`http://dojotoolkit.org/`) which we have used along with the source code of this book, but you can make do with plain HTML buttons or `div` elements if preferred.

How to do it...

1. Create an HTML file and add the OpenLayers dependencies, both JavaScript and CSS. Add the following CSS classes within the `style` element that will be used to style our custom buttons:

```
<style>
    .pointer { background-image: url(./recipes/data/gis_icons/
pointer.png); }
    .point { background-image: url(./recipes/data/gis_icons/point.
png); }
    .line { background-image: url(./recipes/data/gis_icons/line.
png); }
    .polygon { background-image: url(./recipes/data/gis_icons/
polygon.png); }
    .area { background-image: url(./recipes/data/gis_icons/area-
measure.png); }
```

```
    .length { background-image: url(./recipes/data/gis_icons/
length-measure.png); }
</style>
```

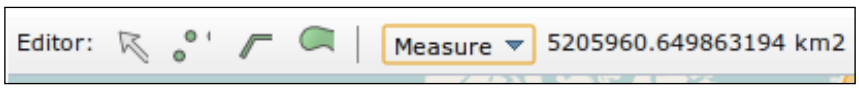
2. Now, let's go to create the toolbar. It will consist of four toggle buttons to select the editing action:

```
<div data-dojo-type="dijit.Toolbar">
    Editor:
    <div data-dojo-type="dijit.form.ToggleButton"
         data-dojo-props="iconClass:'dijitEditorIcon pointer',
showLabel:false, onClick:pointerAction, checked:true">Pan</div>
    <div data-dojo-type="dijit.form.ToggleButton"
         data-dojo-props="iconClass:'dijitEditorIcon point',
showLabel:false, onClick:pointAction">Point</div>
    <div data-dojo-type="dijit.form.ToggleButton"
         data-dojo-props="iconClass:'dijitEditorIcon line',
showLabel:false, onClick:lineAction">Line</div>
    <div data-dojo-type="dijit.form.ToggleButton"
         data-dojo-props="iconClass:'dijitEditorIcon polygon',
showLabel:false, onClick:polygonAction">Polygon</div>
```

3. And one drop-down button to choose the kind of measure to do:

```
    <span data-dojo-type="dijit.ToolbarSeparator"></span>

    <div data-dojo-type="dijit.form.DropDownButton">
        <span>Measure</span>
        <div data-dojo-type="dijit.DropDownMenu">
            <div data-dojo-type="dijit.MenuItem" data-dojo-props="
iconClass:'dijitEditorIcon length', onClick:measureLengthAction">D
istance</div>
            <div data-dojo-type="dijit.MenuItem" data-dojo-props=
"iconClass:'dijitEditorIcon area', onClick:measureAreaAction">Ar
ea</div>
        </div>
    </div>
```

The following screenshot shows the drop-down button created:



4. In addition, the toolbar will hold a span element to show the measured values:

```
    <span id="value"></span>
</div>
```

5. Now, we can place the `div` element that will hold the map:

```
<div id="ch06_external" style="width: 100%; height: 90%;"></div>
```

6. Now, add the required JavaScript code to initialize the map, and add a base layer and a vector layer to add features to the map:

```
<script type="text/javascript">
    // Create map
    var map = new OpenLayers.Map("ch06_external");
    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);

    map.setCenter(new OpenLayers.LonLat(0, 0), 2);

    var vectorLayer = new OpenLayers.Layer.Vector("VectorLayer");
    map.addLayer(vectorLayer);
```

7. Next, add the controls to the map. First, add the controls related to the `DrawFeature` control:

```
    // Add controls
    var pointControl = new OpenLayers.Control.
DrawFeature(vectorLayer, OpenLayers.Handler.Point);
    var lineControl = new OpenLayers.Control.
DrawFeature(vectorLayer, OpenLayers.Handler.Path);
    var polygonControl = new OpenLayers.Control.
DrawFeature(vectorLayer, OpenLayers.Handler.Polygon);
```

8. Then add the `Measure` control, that allows us to measure distances and areas:

```
    var measureControl = new OpenLayers.Control.
Measure(OpenLayers.Handler.Path, {
        persist: true,
        immediate: true,
        eventListeners: {
            'measure': updateMeasure,
            'measurepartial': updateMeasure
        }
    });
    map.addControls([pointControl, lineControl, polygonControl,
measureControl]);
```

9. Implement the functions to handle the buttons that represent the actions of the `EditingToolbar` (hand, draw point, draw path, and draw polygon):

```
    // Functions to control button actions
    var currentControl = null;
    function pointerAction() {
        _unselectButtons(this);
```

```
            _selectControl(null);
        }
        function pointAction(){
            _unselectButtons(this);
            _selectControl(pointControl);
        }
        function lineAction(){
            _unselectButtons(this);
            _selectControl(lineControl);
        }
        function polygonAction(){
            _unselectButtons(this);
            _selectControl(polygonControl);
        }
    Implement the actions for the measure control:
        function measureLengthAction(){
            _unselectButtons(this);
            measureControl.updateHandler(OpenLayers.Handler.Path,
    {persist: true});
            _selectControl(measureControl);
        }
        function measureAreaAction(){
            _unselectButtons(this);
            measureControl.updateHandler(OpenLayers.Handler.Polygon,
    {persist: true});
            _selectControl(measureControl);
        }
```

10. And finally, add the code for the two helper functions:

```
        function _selectControl(control) {
            if(currentControl) {
                currentControl.deactivate();
            }
            if(control) {
                currentControl = control;
                currentControl.activate();
            }
        }
        function _unselectButtons(context) {
            dijit.registry.byClass('dijit.form.ToggleButton').
    forEach(function(button){
                if(context==button) return;
                button.set('checked', false);
            });
```

```
        }
        function updateMeasure(event) {
            var message = event.measure + " " + event.units;
            if(event.order>1) {
                message += "2";
            }
            dojo.byId('value').innerHTML = message;
        }
    </script>
```

How it works...

Every control must be attached to the map, but like in this recipe, it is not necessary for the control to have a visible representation.

In the same way, we can invoke methods on the `OpenLayers.Map` instance to zoom in or zoom out. We can programmatically activate or deactivate a control without the need for a panel or an icon to interact with it.

For the three buttons that allow us to create new features (points, lines, and polygons), we have created three controls based on the `OpenLayers.Control.DrawFeature` control.

This control requires two arguments: the vector layer (to add the new feature to) and a handler (used to interact with the map while we are creating the feature):

```
    var pointControl = new OpenLayers.Control.DrawFeature(vectorLayer,
OpenLayers.Handler.Point);
    var lineControl = new OpenLayers.Control.DrawFeature(vectorLayer,
OpenLayers.Handler.Path);
    var polygonControl = new OpenLayers.Control.
DrawFeature(vectorLayer, OpenLayers.Handler.Polygon);
```

Before these three controls (represented as buttons) in the toolbar, there is a pointer button that allows us to deactivate the current control and pan the map. It does not require any control associated with it. When the button is clicked, we simply deactivate the current control allowing the map to be panned again.

Although visually there are two options in the drop-down button, internally, both correspond to the same control, the `OpenLayers.Control.Measure` control. When one of the measure options is clicked, we set the handler to be used by the control:

```
    function measureLengthAction(){
        _unselectButtons(this);
        measureControl.updateHandler(OpenLayers.Handler.Path,
```

```
{persist: true});
        _selectControl(measureControl);
    }
    function measureAreaAction(){
        _unselectButtons(this);
        measureControl.updateHandler(OpenLayers.Handler.Polygon,
{persist: true});
        _selectControl(measureControl);
    }
```

We have specified the control property `persist` of `Measure` to `true`. This makes the line or polygon that is rendered, to show the measure and stay visible on the map when the measure ends.

The `immediate` property allows the measure process to trigger an event every time the mouse moves.

Finally, we have specified to the event listeners, the `measure` event that is triggered when the measure finishes, and the `measurepartial` event that is triggered on every mouse movement.

Both events execute the `updateMeasure` function, which is responsible to update the measure value shown in the toolbar:

```
function updateMeasure(event) {
    var message = event.measure + " " + event.units;
    if(event.order>1) {
        message += "2";
    }
    dojo.byId('value').innerHTML = message;
}
```

The code responsible for handling the logic to hold only one button or a control active at a time is implemented in the `pointAction` function and makes use of the `currentControl` variable.

The goal is simple, each time a button is pressed, the current selected button is toggled and the related control is deactivated before activating the new selected control.

There's more...

The **GeoExt** project (`http://geoext.org`), is a toolkit based on ExtJS (`http://www.sencha.com/products/extjs/`) with a rich set of user interface components to simplify the creation of rich web applications.

Creating a layer tree or a grid to edit a feature's attributes is pretty simple with GeoExt.

The ESRI's JavaScript API used to build web applications is based on the Dojo Toolkit framework.

Anyway, both are great frameworks.

See also

- The *Placing controls outside the map* recipe in *Chapter 5*, *Adding Controls*
- The *Adding and removing controls* recipe in *Chapter 5*, *Adding Controls*

7
Styling Features

In this chapter we will cover:

- Styling features using symbolizers
- Improving style using StyleMap and the replacement of the feature's attributes
- Playing with StyleMap and the render intents
- Working with unique value rules
- Defining custom rules to style features
- Styling clustered features

Introduction

Once we know how to work with vector layers, such as adding new features or modifying the existing ones, the question we can have in mind is: how to style them?

The visual representation of features, the **style**, is one of the most important concepts in GIS applications. It is not only important from the user's experience or designer's perspective but also as an information requirement, for example, to identify features that match certain rules.

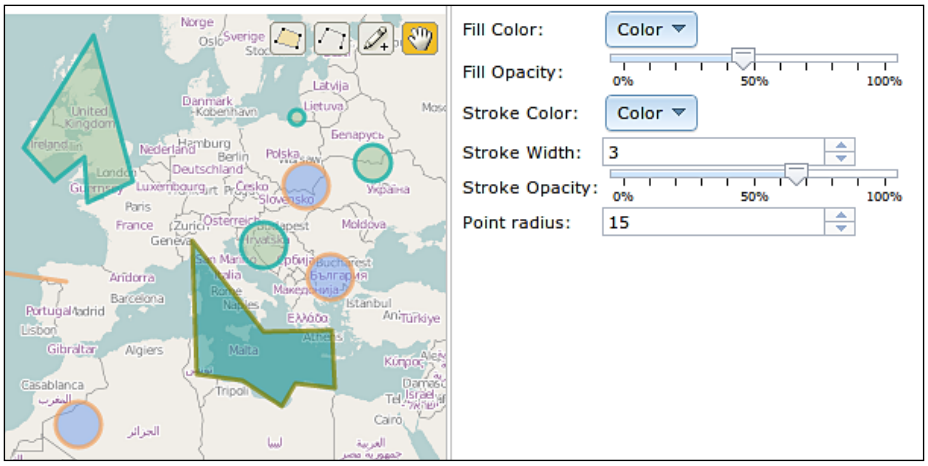
The way we visualize features is not only important to make our application much more attractive, but also to improve the way we bring information to the user. For example, given a set of points that represent some temperatures, if we are interested on the hottest zones, we could represent them with different radius and color values. This way, a lesser radius and a color near to blue means a cold zone while a greater radius and a color near to red means a hot zone.

OpenLayers offers us a great degree of flexibility when styling features that can initially seem a bit complex. Concepts such as symbolizers, StyleMap, rules, or filters are all related with the process of styling.

Let's see all this in the following recipes.

Styling features using symbolizers

To see the most basic form of styling a feature, we are going to create a little map editor that allows adding new features by specifying some few style properties:



Each `OpenLayers.Feature.Vector` instance can have a style associated with it. This style is called **symbolizer**, which is nothing more than a JavaScript object with some fields that specify the fill color, stroke, and so on. For example:

```
{
    fillColor: "#ee9900",
    fillOpacity: 0.4,
    strokeColor: "#ee9900",
    strokeOpacity: 1,
    strokeWidth: 1
}
```

In the code, every time a feature is going to be added to the map, the code will get the fill and stroke properties from the controls on the left-hand side and will create a new symbolizer hash to be used by the new feature.

Getting ready

The source code has two main sections, one for HTML, where all the controls are placed, and a second one for the JavaScript code.

The HTML section has plenty of codes related with the controls used to select the fill and stroke properties. These controls come from the Dojo Toolkit project (`http://dojotoolkit.org`) and because they are not the goal of this recipe, we have not covered it here. We encourage the reader to take a look at it in the code bundle of the book.

Let's see the JavaScript code.

How to do it...

1. After creating the HTML file including OpenLayers dependencies (see the *Getting ready* section for the HTML code), create the `map` instance in the `div` element identified by `ch07_using_symbolizers` and add a base layer:

```
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch07_using_symbolizers");

    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);

    map.setCenter(new OpenLayers.LonLat(0,0), 3)
```

2. Now, add a vector layer where new features will be placed:

```
    var vectorLayer = new OpenLayers.Layer.Vector("Features");
    vectorLayer.events.register('beforefeatureadded', vectorLayer,
setFeatureStyle);
    map.addLayer(vectorLayer);
```

3. Add the `OpenLayers.Control.EditingToolbar` control that allows to add new features to the previous vector layer:

```
    var editingControl = new OpenLayers.Control.
EditingToolbar(vectorLayer);
    map.addControl(editingControl);
```

4. Add the code responsible, to get and apply the style to the new features:

```
    function setFeatureStyle(event) {
        var fillColor = dijit.byId('fillColor').get('value');
        var fillOpacity = dijit.byId('fillOpacity').
get('value')/100;
        var strokeColor = dijit.byId('strokeColor').get('value');
        var strokeWidth = dijit.byId('strokeWidth').get('value');
        var strokeOpacity = dijit.byId('strokeOpacity').
get('value')/100;
        var pointRadius = dijit.byId('pointRadius').get('value');
```

```
            var style = OpenLayers.Util.extend({},
              OpenLayers.Feature.Vector.style['default']);
            style.fillColor = fillColor;
            style.fillOpacity = fillOpacity;
            style.strokeColor = strokeColor;
            style.strokeWidth = strokeWidth;
            style.strokeOpacity = strokeOpacity;
            style.pointRadius = pointRadius;

            event.feature.style = style;
        }
```

How it works...

The idea is, each time a feature is added to the layer using the EditingToolbar control, create a symbolizer and apply it to the new feature.

The first step is to register a `beforefeatureadded` event listener in the vector layer so that we are notified each time a new feature is going to be added:

```
vectorLayer.events.register('beforefeatureadded',
  vectorLayer, setFeatureStyle);
```

The function `setFeatureStyle` is called every time a new feature is added. An `event` parameter is passed on each call, pointing to the feature to be added (`event.feature`) and a reference to the vector layer (`event.object`).

The `event.object` references the object passed as the `object` parameter in the `event.register(event_type, object, listener)` method.

```
function setFeatureStyle(event) {
    var fillColor = dijit.byId('fillColor').get('value');
    var fillOpacity = dijit.byId('fillOpacity').get('value')/100;
    var strokeColor = dijit.byId('strokeColor').get('value');
    var strokeWidth = dijit.byId('strokeWidth').get('value');
    var strokeOpacity = dijit.byId('strokeOpacity').
get('value')/100;
    var pointRadius = dijit.byId('pointRadius').get('value');

    var style = OpenLayers.Util.extend({}, OpenLayers.Feature.
Vector.style['default']);
    style.fillColor = fillColor;
    style.fillOpacity = fillOpacity;
    style.strokeColor = strokeColor;
```

```
        style.strokeWidth = strokeWidth;
        style.strokeOpacity = strokeOpacity;
        style.pointRadius = pointRadius;

        event.feature.style = style;
    }
```

Once we obtain the property values from the Dojo widgets, we create a new symbolizer.

The `OpenLayers.Feature.Vector` class defines some style symbolizers in the `style` array property, so the quickest way is to create a copy of one of those styles (actually we have extended it) and then modify some of its properties.

There's more...

The question that can arise here is: what takes precedence when styling, a rule applied to a vector layer or a symbolizer applied to a single feature?

The answer is: styles goes from bottom to top, that is, if we have specified a symbolizer in a feature then it will be used to render it, otherwise, any rule or StyleMap assigned to the vector layer will be applied to its features.

See also

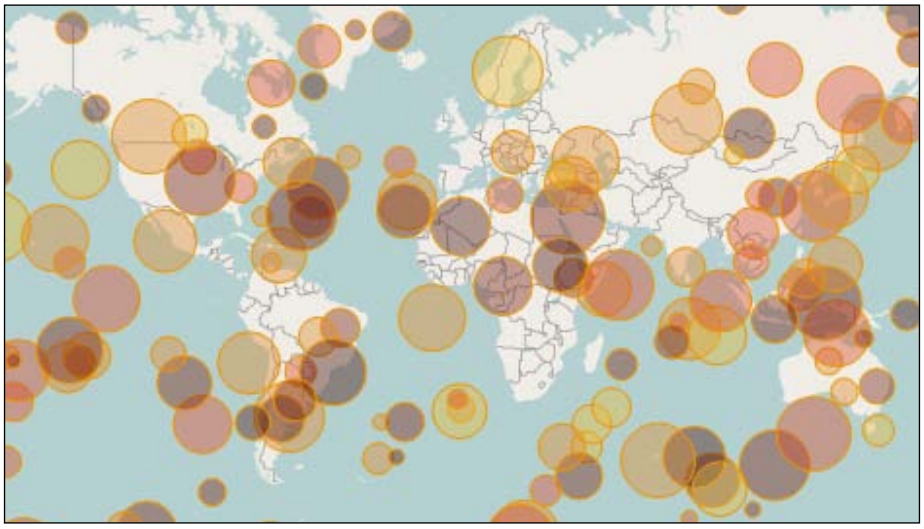
- The *Improving style using StyleMap and the replacement of feature's attributes* recipe
- The *Playing with StyleMap and the render intents* recipe

Improving style using StyleMap and the replacement of feature's attributes

We can summarize that there are two ways to style a feature. The first is applying a symbolizer hash directly to the feature (see the *Styling features using symbolizers* recipe). The second is applying the style to the layer so every feature contained in it becomes styled.

The second one is the preferred way in many situations. It is a generic way to style all the features in a layer by setting some styles and rules.

This recipe shows how we can use the `StyleMap` instances and how easily we can style all the points of a layer without applying a style on each feature. The output of this recipe should look similar to the following screenshot:



In addition, the technique we will use allows us to involve the feature's attributes to select a point radius and color, creating them all together more dynamically.

How to do it...

1. Once we have created the HTML file with OpenLayers dependencies, start creating the `div` element that will hold the map instance:

```
<div id="ch07_styleMap" style="width: 100%; height: 95%;"></div>
```

2. Now, create the map instance and add a base layer:

```
<script type="text/javascript">
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch07_styleMap");

    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);

    map.setCenter(new OpenLayers.LonLat(0,0), 2)
```

3. Now, let's start defining the style for the whole layer. First create a color palette for the points:

```
    // Create stylemap for the layer
    var colors = ['#EBC137','#E38C2D','#DB4C2C','#771E10','#4811
0C'];
```

4. Create a style instance from a previous symbolizer hash:

```
    var style = OpenLayers.Util.extend({}, OpenLayers.Feature.
Vector.style["default"]);
    style.pointRadius = "${radius}";
    style.fillColor = '${colorFunction}';

    var defaultStyle = new OpenLayers.Style(style, {
        context: {
            colorFunction: function(feature) {
                return colors[feature.attributes.temp];
            }
        }
    });
```

5. Create a vector layer applying the desired `StyleMap`:

```
    // Create the vector layer
    var vectorLayer = new OpenLayers.Layer.Vector("Features", {
        styleMap: new OpenLayers.StyleMap(defaultStyle)
    });
    map.addLayer(vectorLayer);
```

6. Finally, create some random points. Each feature will have two attributes `radius` and `temp` with random values:

```
    // Create random feature points.
    var pointFeatures = [];
    for(var i=0; i< 150; i++) {
        var px = Math.random() * 360 - 180;
        var py = Math.random() * 170 - 85;

        // Create a lonlat instance and transform it to the map
projection.
        var lonlat = new OpenLayers.LonLat(px, py);
        lonlat.transform(new OpenLayers.Projection("EPSG:4326"),
new OpenLayers.Projection("EPSG:900913"));

        var pointGeometry = new OpenLayers.Geometry.Point(lonlat.
lon, lonlat.lat);
```

```
            var pointFeature = new OpenLayers.Feature.
    Vector(pointGeometry);

            // Add random attributes
            var radius = Math.round(Math.random() * 15 + 4);
            var temp = Math.round(Math.random() * 4);
            pointFeature.attributes.radius = radius;
            pointFeature.attributes.temp = temp;

            pointFeatures.push(pointFeature);
        }
        // Add features to the layer
        vectorLayer.addFeatures(pointFeatures);

    </script>
```

How it works...

Let's go to describe first the random point features we have added to the vector layer.

The idea is to create some random points at random places. Because of this, we create some random x-y values, transform to map coordinates, create geometry, and finally create a feature with that geometry:

```
        var px = Math.random() * 360 - 180;
        var py = Math.random() * 170 - 85;

        // Create a lonlat instance and transform it to the map
projection.
        var lonlat = new OpenLayers.LonLat(px, py);
        lonlat.transform(new OpenLayers.Projection("EPSG:4326"), new
OpenLayers.Projection("EPSG:900913"));

        var pointGeometry = new OpenLayers.Geometry.Point(lonlat.lon,
lonlat.lat);
        var pointFeature = new OpenLayers.Feature.
Vector(pointGeometry);
```

In addition, we are setting in each feature, a couple of attributes (`radius` and `temp`) with random values:

```
        // Add random attributes
        var radius = Math.round(Math.random() * 15 + 4);
        var temp = Math.round(Math.random() * 4);
        pointFeature.attributes.radius = radius;
        pointFeature.attributes.temp = temp;
```

These attributes will be used later in the feature's style definition.

Let's go to describe the creation of the style for the vector layer.

We want each feature to be represented as a point using the attribute `radius` for the point's radius and the `temp` attribute for the point's color.

The first step is to create a symbolizer by hash copying (actually extending) that is defined at `OpenLayers.Feature.Vector.style["default"])`.

```
    var style = OpenLayers.Util.extend({}, OpenLayers.Feature.Vector.
style["default"]);
```

If you look at the source code you will find that `OpenLayers.Feature.Vector.style["default"])` is defined as:

```
{
    fillColor: "#ee9900",
    fillOpacity: 0.4,
    hoverFillColor: "white",
    hoverFillOpacity: 0.8,
    strokeColor: "#ee9900",
    strokeOpacity: 1,
    strokeWidth: 1,
    strokeLinecap: "round",
    strokeDashstyle: "solid",
    hoverStrokeColor: "red",
    hoverStrokeOpacity: 1,
    hoverStrokeWidth: 0.2,
    pointRadius: 6,
    hoverPointRadius: 1,
    hoverPointUnit: "%",
    pointerEvents: "visiblePainted",
    cursor: "inherit"
}
```

Once we have a fresh copy of the symbolizer, we change the `fillColor` and `pointRadius` properties. What is the challenge here? Well, we do not want fixed values for these properties, we want these properties to take their values from the feature's attributes they are styling.

Fortunately, OpenLayers helps us with the **attribute replacement** syntax. In the same way, we can write a literal value as follows:

```
pointRadius: 15
```

We can specify that the radius value must come from the feature's `featureRadius` attribute:

```
pointRadius: '${featureRadius}'
```

So in our sample, our features have the attribute `radius` defined as a random value that can be used here:

```
style.pointRadius = "${radius}";
```

In the same way as we can use an attribute to be replaced as the property value, we can also set a function which must return the value to be used as the property value. This is the case for the `fillColor` property:

```
style.fillColor = '${colorFunction}';
```

As we will see next, the function `colorFunction` returns a value depending on the feature's `temp` attribute, that is, we do not want to use the `temp` attribute directly but a value computed from it.

Unfortunately, attribute replacement cannot be used directly in a symbolizer hash applied to a feature or layer, it only works through an `OpenLayers.Style` instance. In addition, thanks to the `OpenLayers.Style` instance, we can define the functions, such as `colorFunction` to be used to compute a style property value.

Attribute replacement can only be used through an `OpenLayers.Style` instance.

In the recipe, once we define the symbolizer hash, we can create an instance as follows:

```
var defaultStyle = new OpenLayers.Style(style, {
    context: {
        colorFunction: function(feature) {
            return colors[feature.attributes.temp];
        }
    }
});
```

The first parameter is the symbolizer hash that has been previously defined, which makes use of the attribute's replacement feature. The second parameter, the `context`, is an object passed in the process of rendering features. Here we define the required functions, such as the `colorFunction` that will be available in the rendering process, and will define the value for the `fillColor` depending on the `temp` attribute of each feature.

At this point, we are almost done. The only remaining thing is to create a vector layer that uses the defined `OpenLayers.Style` instance to style the features.

The `OpenLayers.Layer.Vector` class has a `styleMap` property used to specify the styles to apply to the features. The `OpenLayers.StyleMap` class can be instantiated with passing a different argument, but here we are creating it using the previously defined `OpenLayers.Style` style:

```
var vectorLayer = new OpenLayers.Layer.Vector("Features", {
    styleMap: new OpenLayers.StyleMap(defaultStyle)
});
map.addLayer(vectorLayer);
```

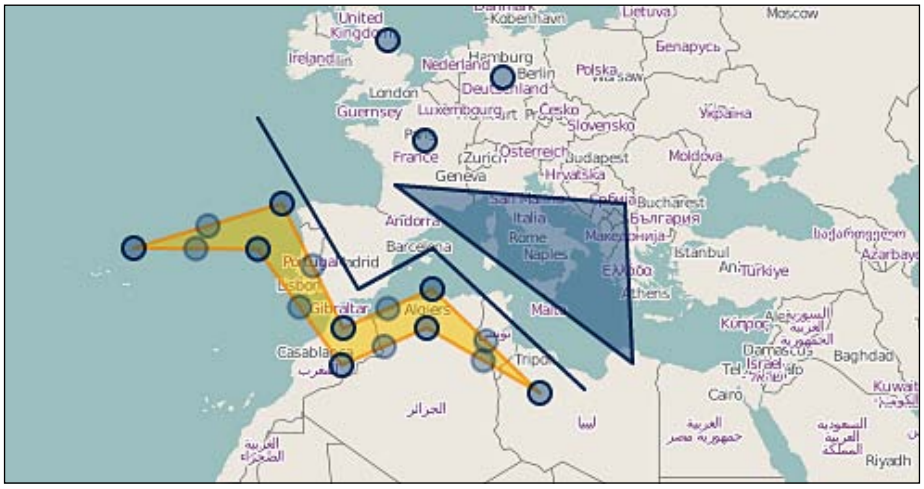
Now, our recipe is complete. As we can see, there is no need to create a symbolizer and apply it to each feature. The only thing we need to do is to define a style and assign it to the vector layer.

See also

- The *Playing with StyleMap and render intents* recipe
- The *Styling features using symbolizers* recipe
- The *Creating features programmatically* recipe in *Chapter 3*, *Vector Layers*

Playing with StyleMap and the render intents

There are some controls, such as `SelectFeature`, `ModifyFeature`, or `EditingToolbar`, which change the style of the feature depending on its current state, that is, if it is selected or is currently being edited. How does OpenLayers manage this? The answer is, through the **render intents**:



This recipe shows how we can modify the styles used for each render intent to change the look of our applications.

This way, features will be drawn on the map using blue instead of orange. Temporary features, those that are going to be created, will be drawn using green. Finally, those features that are selected, or are in the middle of the modification process, will be drawn using orange.

How to do it...

1. Create a new HTML file and add the OpenLayers dependencies. The first step is to add the `div` element to hold the map instance:

```
<div id="ch07_rendering_intents" style="width: 100%; height:
95%;"></div>
```

2. In the JavaScript section, initialize the map instance, add a base layer, and center the viewport:

```
<script type="text/javascript">
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch07_rendering_intents");

    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);

    map.setCenter(new OpenLayers.LonLat(0,0), 2)
```

3. Now we are going to create three different styles:

```
    var defaultStyle = new OpenLayers.Style({
        fillColor: "#336699",
        fillOpacity: 0.4,
        hoverFillColor: "white",
        hoverFillOpacity: 0.8,
        strokeColor: "#003366",
        strokeOpacity: 0.8,
        strokeWidth: 2,
        strokeLinecap: "round",
        strokeDashstyle: "solid",
        hoverStrokeColor: "red",
        hoverStrokeOpacity: 1,
        hoverStrokeWidth: 0.2,
        pointRadius: 6,
        hoverPointRadius: 1,
        hoverPointUnit: "%",
        pointerEvents: "visiblePainted",
```

```
    cursor: "inherit"
});
var selectStyle = new OpenLayers.Style({
    fillColor: "#ffcc00",
    fillOpacity: 0.4,
    hoverFillColor: "white",
    hoverFillOpacity: 0.6,
    strokeColor: "#ff9900",
    strokeOpacity: 0.6,
    strokeWidth: 2,
    strokeLinecap: "round",
    strokeDashstyle: "solid",
    hoverStrokeColor: "red",
    hoverStrokeOpacity: 1,
    hoverStrokeWidth: 0.2,
    pointRadius: 6,
    hoverPointRadius: 1,
    hoverPointUnit: "%",
    pointerEvents: "visiblePainted",
    cursor: "pointer"
});
var temporaryStyle = new OpenLayers.Style({
    fillColor: "#587058",
    fillOpacity: 0.4,
    hoverFillColor: "white",
    hoverFillOpacity: 0.8,
    strokeColor: "#587498",
    strokeOpacity: 0.8,
    strokeLinecap: "round",
    strokeWidth: 2,
    strokeDashstyle: "solid",
    hoverStrokeColor: "red",
    hoverStrokeOpacity: 1,
    hoverStrokeWidth: 0.2,
    pointRadius: 6,
    hoverPointRadius: 1,
    hoverPointUnit: "%",
    pointerEvents: "visiblePainted",
    cursor: "inherit"
});
```

4. After this, create a `StyleMap` instance that holds the three styles created as three different render intents:

```
var styleMap = new OpenLayers.StyleMap({
    'default': defaultStyle,
    'select': selectStyle,
    'temporary': temporaryStyle
});
```

5. Now we can create a vector layer using the previous `StyleMap` instance:

```
var vectorLayer = new OpenLayers.Layer.Vector("Features", {
    styleMap: styleMap
});
map.addLayer(vectorLayer);
```

6. Finally, we are going to add some controls to the map to allow the addition of new features and modification of the existing ones:

```
    var editingControl = new OpenLayers.Control.
EditingToolbar(vectorLayer);
    var modifyControl = new OpenLayers.Control.
ModifyFeature(vectorLayer, {
        toggle: true
    });
    editingControl.addControls([modifyControl]);
    map.addControl(editingControl);
</script>
```

How it works...

Every vector layer can have an `OpenLayers.StyleMap` instance associated with it. On its own, a `StyleMap` instance stores one or more references to the `OpenLayers.Style` instances, each one of which acts as a render intent:

```
var styleMap = new OpenLayers.StyleMap({
    'default': defaultStyle,
    'select': selectStyle,
    'temporary': temporaryStyle
});
```

Every `Style` instance stores information about a style, and usually they are created from a symbolizer hash, as in this recipe:

```
var defaultStyle = new OpenLayers.Style({
    fillColor: "#336699",
    fillOpacity: 0.4,
    hoverFillColor: "white",
```

```
            hoverFillOpacity: 0.8,
            strokeColor: "#003366",
            strokeOpacity: 0.8,
            strokeWidth: 2,
            strokeLinecap: "round",
            strokeDashstyle: "solid",
            hoverStrokeColor: "red",
            hoverStrokeOpacity: 1,
            hoverStrokeWidth: 0.2,
            pointRadius: 6,
            hoverPointRadius: 1,
            hoverPointUnit: "%",
            pointerEvents: "visiblePainted",
            cursor: "inherit"
        });
```

Here we have defined a new style for the three render intents: `default`, `select`, and `temporary`, which are well known render intents used by most of the controls.

A `StyleMap` can store as many render intents as we desire, we are not limited to these three commonly used render intents. For example, we can define render intents such as `red` or `hidden`, and associate a `Style` for them that renders features in red or not display them at all.

By setting the property `display` to `"none"` on the style's symbolizer hash, we can hide features. This is usually used in the `delete` render intent.

The render intents such as `default`, `select`, and `temporary`, are used extensively by many components within OpenLayers. This way, when a feature is rendered, the `default` style is used. When a feature is selected using the `OpenLayers.Control.SelectFeature` control, the `select` render intent is used to render the features. And when we are creating a new feature with `OpenLayers.Control.EditingToolbar` (which internally uses `OpenLayers.Control.DrawFeature`), the control renders the feature using the style defined on the `temporary` render intent.

So, creating new render intents is no problem. In addition, we can create our custom controls and let them decide which render intent the layer must use to render the features.

Finally, let's briefly describe the code used to create the panel with the controls.

First, we have created an `OpenLayers.Control.EditingToolbar` instance:

```
    var editingControl = new OpenLayers.Control.
EditingToolbar(vectorLayer);
```

This is an `OpenLayers.Control.Panel` control containing buttons that activates/deactivates some `OpenLayers.Control.DrawFeature` controls. Next, we have created an `OpenLayers.ControlModifyFeature` instance, which is a single control and we have added it to the EditingToolbar control so that it becomes visible as a new button:

```
    var modifyControl = new OpenLayers.Control.
ModifyFeature(vectorLayer, {
        toggle: true
    });
    editingControl.addControls([modifyControl]);
```

In the screenshot, the `ModifyFeature` control is represented by the cross icon.

There's more...

The process to style and render a feature is complex. The following lines summarize the main steps involved in the feature styling process.

For each feature, a vector layer must render the following:

- The method `OpenLayers.Layer.Vector.drawFeature(feature, style)` is called. It accepts two parameters: the `feature` to be drawn and the `style` to be used. It can be a symbolizer or a render intent string.
- If the feature has a `style` property, it is used to render the feature.
- Otherwise, if the vector layer has a `style` property, it is used to render the feature.
- Otherwise, if the `style` argument is provided and it is a style symbolizer, then it is used to render the feature.
- If the `style` is a render intent string, then a symbolizer is created from the `Style` property associated to the render intent using the `createSymbolizer` method. This is where feature attributes are merged within the symbolizer.

See also

- The *Styling features using symbolizers* recipe
- The *Improving style using StyleMap and the replacement of feature's attributes* recipe

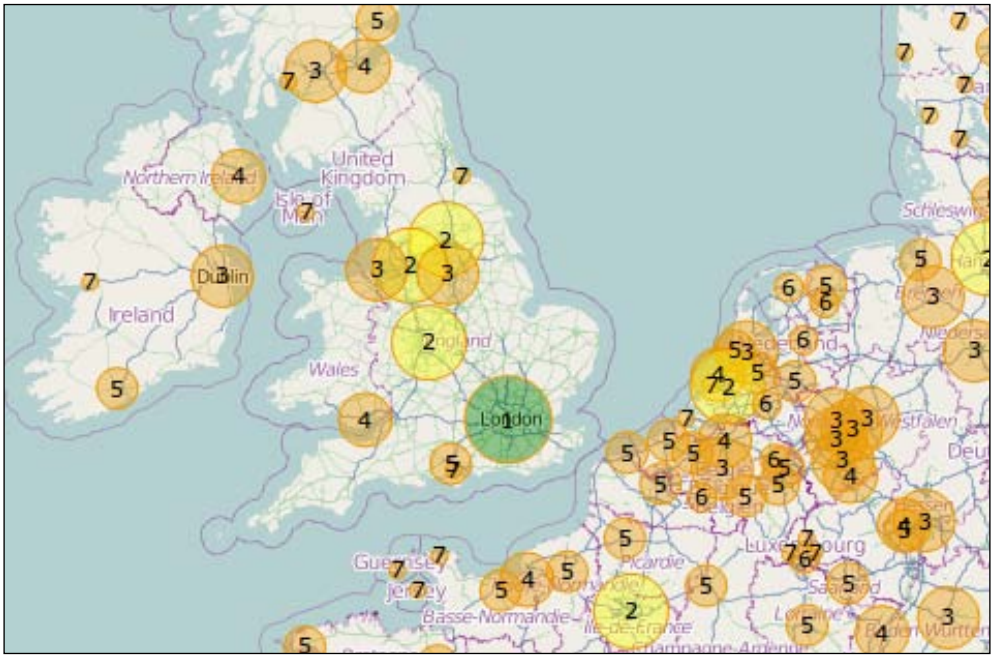
Working with unique value rules

Usually, we do not only style features by what they represent, for example a city or a village, but we style depending on their attributes, such as the number of citizens, year of foundation, and number of squares.

To help on these cases, OpenLayers offers us the possibility to define rules to decide how to style features. For example, we can define a rule that for all features of a city with a population greater than 100,000 a point with radius 20 and color brown can be rendered, while for cities with a population less than 100,000, a point with radius 10, color orange, and semi transparent can be rendered.

Beginning in the world of the rules, the concept of *unique value rules* are the simplest case we can find. The idea is simple, apply one style or another depending on the value of a feature's attribute.

In this recipe, we are going to load a GeoJSON file, with some cities of the world, and apply a rule that will set the radius of the points depending on the popularity rank attribute (the `POP_RANK` attribute), as shown in the following screenshot:



How to do it...

1. Start creating a new HTML file and add the OpenLayers dependencies. Add a `div` element to hold the map instance:

```
<div id="ch07_unique_value_rules" style="width: 100%; height:
95%;"></div>
```

2. Within the `script` element, add the required code to initialize the map and add a base layer:

```
<script type="text/javascript">
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch07_unique_value_rules");

    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);

    map.setCenter(new OpenLayers.LonLat(0,0), 4)
```

3. Now define different styles to be used on the `POP_RANK` feature's attribute:

```
    var styles = {
      7: { pointRadius: 4, label: "${POP_RANK}" },
      6: { pointRadius: 7, label: "${POP_RANK}" },
      5: { pointRadius: 10, label: "${POP_RANK}" },
      4: { pointRadius: 13, label: "${POP_RANK}" },
      3: { pointRadius: 15, label: "${POP_RANK}" },
      2: { pointRadius: 18, label: "${POP_RANK}", fillColor:
"yellow" },
      1: { pointRadius: 21, label: "${POP_RANK}", fillColor:
"green" }
    };
```

4. Create a `StyleMap` instance and define a unique value rule:

```
    var styleMap = new OpenLayers.StyleMap();
    styleMap.addUniqueValueRules("default", "POP_RANK", styles);
```

5. Finally, add a vector layer with some cities of the world and make use of the previous `StyleMap` instance:

```
    map.addLayer(new OpenLayers.Layer.Vector("World Cities
(GeoJSON)", {
        protocol: new OpenLayers.Protocol.HTTP({
            url: "http://localhost:8080/openlayers-cookbook/
recipes/data/world_cities.json",
            format: new OpenLayers.Format.GeoJSON()
        }),
```

```
            styleMap: styleMap,
            strategies: [new OpenLayers.Strategy.Fixed()]
        }));
    </script>
```

How it works...

Almost all the magic of this recipe resides in the `OpenLayers.StyleMap.addUniqueValueRules()` method. So, the sentence:

```
styleMap.addUniqueValueRules("default", "POP_RANK", styles);
```

means, apply to the `default` render intent the specified `style` property, depending on the value of the `POP_RANK` attribute.

It makes more sense once we look at the hash style. Depending on the value of the `POP_RANK`, the radius of the points representing cities will vary between `4` and `21`:

```
var styles = {
  7: { pointRadius: 4, label: "${POP_RANK}" },
  6: { pointRadius: 7, label: "${POP_RANK}" },
  5: { pointRadius: 10, label: "${POP_RANK}" },
  4: { pointRadius: 13, label: "${POP_RANK}" },
  3: { pointRadius: 15, label: "${POP_RANK}" },
  2: { pointRadius: 18, label: "${POP_RANK}", fillColor: "yellow"
},
  1: { pointRadius: 21, label: "${POP_RANK}", fillColor: "green" }
};
```

Finally, we have added to the map a vector layer that uses the previously created `StyleMap` instance where the unique value rules are defined:

```
map.addLayer(new OpenLayers.Layer.Vector("World Cities (GeoJSON)",
{
    protocol: new OpenLayers.Protocol.HTTP({
        url: "http://localhost:8080/openlayers-cookbook/recipes/
data/world_cities.json",
        format: new OpenLayers.Format.GeoJSON()
    }),
    styleMap: styleMap,
    strategies: [new OpenLayers.Strategy.Fixed()]
}));
```

In addition, the vector layer uses an `OpenLayers.Protocol.HTTP` instance to load the GeoJSON file and an `OpenLayers.Strategy.Fixed` instance is used to just load the source data once.

There's more...

The use of unique value rules through the `addUniqueValueRules()` method is easy, but as we can understand, it works only for a discrete value range.

Also, the flexibility is poor because it is equivalent to an *is equal* rule, where we have no way to map a range of values to the same style.

See also

- The *Playing with StyleMap and the render intents* recipe
- The *Defining custom rules to style features* recipe
- The *Using point features as markers* recipe in *Chapter 3*, *Vector Layers*
- The *Working with popups* recipe in *Chapter 3*, *Vector Layers*

Defining custom rules to style features

We will see a brief explanation before continuing with this recipe. The goal, as in the other recipes in the chapter, is to style the features of a vector layer depending on their attributes' values or their kind of feature.

So, an `OpenLayers.Layer.Vector` layer class can have an `OpenLayers.StyleMap` instance associated with it, which determines the default style of the layers if it has only one `OpenLayers.Style`, or the set of styles that can be applied for each render intent if it contains more than one `OpenLayers.Style`. In its own way, each `OpenLayers.Style` instance can be used in two forms:

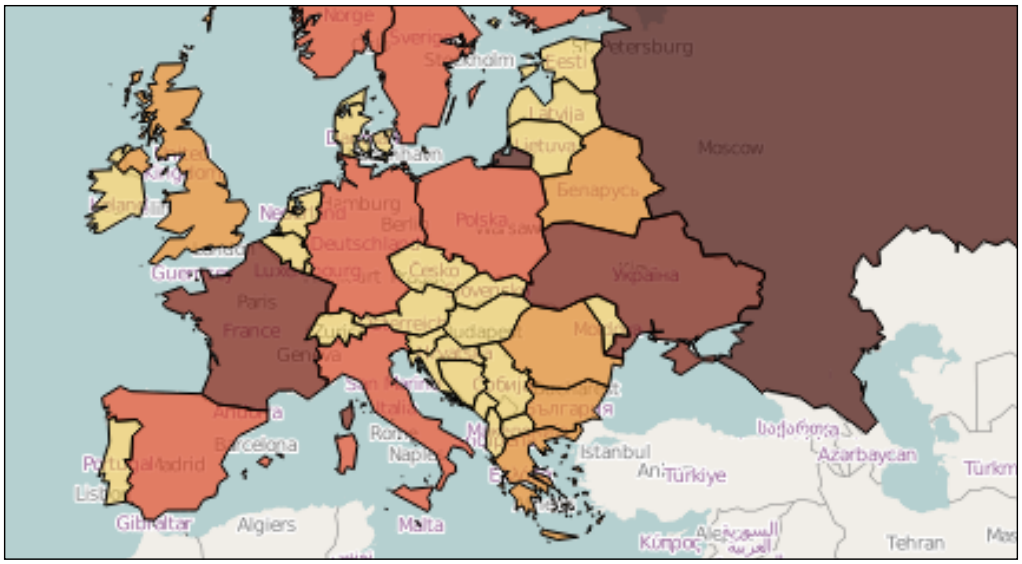
- Having a symbolizer hash acting as the default style to apply to the features
- Having some `OpenLayers.Rule` instances associated with it

Here we arrive to the main concept of this recipe, the **rules**.

A rule is nothing more than a join between a filter (concretely an `OpenLayers.Filter`) and a symbolizer, if the filter matches the feature then the symbolizer is applied.

This simple thing gives us lot of flexibilities and power to style our features. In addition to the possibility to use symbolizers with attribute replacement, we can also use the set of filters OpenLayers offers us: comparison filters, spatial filters, or logical filters.

The goal of this recipe is to load a GML file with European countries and style them depending on their `AREA` attribute, as shown in the following screenshot:



How to do it...

1. Once created an HTML file with the OpenLayers dependencies, add the `div` element to hold the map:

```
<div id="ch07_custom_rules" style="width: 100%; height: 95%;"></
div>
```

2. In the JavaScript code section, initialize the map, add OpenStreetMap as the base layer, and center the map at the desired place:

```
<script type="text/javascript">
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch07_custom_rules");

    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);

    map.setCenter(new OpenLayers.LonLat(40,50).transform(new
OpenLayers.Projection("EPSG:4326"),
    new OpenLayers.Projection("EPSG:900913")), 3);
```

3. Now, define five different rules to style elements based on the `AREA` attribute of the features. The following code has the rule to check if the value is less than 10,000:

```
    var aRule = new OpenLayers.Rule({
        filter: new OpenLayers.Filter.Comparison({
            type: OpenLayers.Filter.Comparison.LESS_THAN,
            property: "AREA",
            value: 10000
```

```
        }),
        symbolizer: {
            fillColor: "#EBC137",
            fillOpacity: 0.5,
            strokeColor: "black"
        }
    });
```

4. The following code has the rule to check if the value is between 10,000 and 25,000:

```
    var bRule = new OpenLayers.Rule({
        filter: new OpenLayers.Filter.Logical({
            type: OpenLayers.Filter.Logical.AND,
            filters: [
                new OpenLayers.Filter.Comparison({
                    type: OpenLayers.Filter.Comparison.GREATER_
THAN,
                    property: "AREA",
                    value: 10000
                }),
                new OpenLayers.Filter.Comparison({
                    type: OpenLayers.Filter.Comparison.LESS_THAN_
OR_EQUAL_TO,
                    property: "AREA",
                    value: 25000
                })
            ]
        }),
        symbolizer: {
            fillColor: "#E38C2D",
            fillOpacity: 0.7,
            strokeColor: "black"
        }
    });
```

5. The rule to check if the value is between 25,000 and 50,000:

```
    var cRule = new OpenLayers.Rule({
        filter: new OpenLayers.Filter.Logical({
            type: OpenLayers.Filter.Logical.AND,
            filters: [
                new OpenLayers.Filter.Comparison({
                    type: OpenLayers.Filter.Comparison.GREATER_
THAN,
                    property: "AREA",
                    value: 25000
                }),
```

```
                new OpenLayers.Filter.Comparison({
                    type: OpenLayers.Filter.Comparison.LESS_THAN_
OR_EQUAL_TO,
                    property: "AREA",
                    value: 50000
                })
            ]
        }),
        symbolizer: {
            fillColor: "#DB4C2C",
            fillOpacity: 0.7,
            strokeColor: "black"
        }
    });
```

6. The rule to check if the value is between 50,000 and 100,000:

```
    var dRule = new OpenLayers.Rule({
        filter: new OpenLayers.Filter.Logical({
            type: OpenLayers.Filter.Logical.AND,
            filters: [
                new OpenLayers.Filter.Comparison({
                    type: OpenLayers.Filter.Comparison.GREATER_
THAN,
                    property: "AREA",
                    value: 50000
                }),
                new OpenLayers.Filter.Comparison({
                    type: OpenLayers.Filter.Comparison.LESS_THAN_
OR_EQUAL_TO,
                    property: "AREA",
                    value: 100000
                })
            ]
        }),
        symbolizer: {
            fillColor: "#771E10",
            fillOpacity: 0.7,
            strokeColor: "black"
        }
    });
```

7. And finally, the rule to check for values greater than 100,000:

```
    var eRule = new OpenLayers.Rule({
        filter: new OpenLayers.Filter.Comparison({
```

```
            type: OpenLayers.Filter.Comparison.GREATER_THAN_OR_
EQUAL_TO,
            property: "AREA",
            value: 100000
        }),
        symbolizer: {
            fillColor: "#48110C",
            fillOpacity: 0.7,
            strokeColor: "black"
        }
    });
```

8. Create the style with the customary rules defined previously:

```
    var style = new OpenLayers.Style();
    style.addRules([aRule, bRule, cRule, dRule, eRule]);
```

9. Finally, create a vector layer that loads the GML file and uses the previous style:

```
    map.addLayer(new OpenLayers.Layer.Vector("World Cities
(GeoJSON)", {
        protocol: new OpenLayers.Protocol.HTTP({
            url: "http://localhost:8080/openlayers-cookbook/
recipes/data/europe.gml",
            format: new OpenLayers.Format.GML()
        }),
        styleMap: new OpenLayers.StyleMap(style),
        strategies: [new OpenLayers.Strategy.Fixed()]
    }));
</script>
```

How it works...

As we described at the beginning of the recipe, an `OpenLayers.Style` instance admits a set of `OpenLayers.Rule` instances to style the features.

Given a rule, all the features that match the specified `OpenLayers.Filter` are styled with the specified symbolizer hash, and thanks to the filters, we have enough flexibility to create the comparison or logical filters.

In the code, we have created five filters. Let's describe two of them.

The `aRule` rule is formed by a comparison filter that matches all the features with an `AREA` attribute having a value less than 10,000:

```
var aRule = new OpenLayers.Rule({
    filter: new OpenLayers.Filter.Comparison({
```

```
            type: OpenLayers.Filter.Comparison.LESS_THAN,
            property: "AREA",
            value: 10000
        }),
        symbolizer: {
            fillColor: "#EBC137",
            fillOpacity: 0.5,
            strokeColor: "black"
        }
    });
```

The `bRule` uses a more complex rule. In this case, it is a logical `AND` filter composed of two comparison filters. It matches all the features to check whether their `AREA` attribute is greater than 10,000 and less than or equal to 25,000:

```
    var bRule = new OpenLayers.Rule({
        filter: new OpenLayers.Filter.Logical({
            type: OpenLayers.Filter.Logical.AND,
            filters: [
                new OpenLayers.Filter.Comparison({
                    type: OpenLayers.Filter.Comparison.GREATER_THAN,
                    property: "AREA",
                    value: 10000
                }),
                new OpenLayers.Filter.Comparison({
                    type: OpenLayers.Filter.Comparison.LESS_THAN_OR_
EQUAL_TO,
                    property: "AREA",
                    value: 25000
                })
            ]
        }),
        symbolizer: {
            fillColor: "#E38C2D",
            fillOpacity: 0.7,
            strokeColor: "black"
        }
    });
```

Once we have created all the desired rules, we can create an `OpenLayers.Style` instance:

```
    var style = new OpenLayers.Style();
    style.addRules([aRule, bRule, cRule, dRule, eRule]);
```

Then apply it to the vector layer:

```
    map.addLayer(new OpenLayers.Layer.Vector("World Cities (GeoJSON)",
{
        protocol: new OpenLayers.Protocol.HTTP({
            url: "http://localhost:8080/openlayers-cookbook/recipes/
data/europe.gml",
            format: new OpenLayers.Format.GML()
        }),
        styleMap: new OpenLayers.StyleMap(style),
        strategies: [new OpenLayers.Strategy.Fixed()]
    }));
```

We have created an `OpenLayers.StyleMap` instance passing only one style and not a style for each desired render intent. This means there will be no render intents in the layer, or expressed in other words, all the render intents will be rendered with the same style.

Because the vector layer must read data from a GML file in our server, we have made use of an `OpenLayers.Protocol.HTTP` instance that loads files from the specified URL and uses an instance in the `OpenLayers.Format.GML` format to read it.

Finally, to center the map's viewport, we needed to transform the coordinates.

Because the base layer of the map is OpenStreetMap, this makes the map's projection to become EPSG:900913, while we are specifying the center location as latitude/longitude using the EPSG:4326. Because of this we need to make a transformation:

```
  map.setCenter(new OpenLayers.LonLat(40,50).transform(new OpenLayers.
Projection("EPSG:4326"),
    new OpenLayers.Projection("EPSG:900913")), 3);
```

There's more...

In our code, we have created the style with the sentences:

```
var style = new OpenLayers.Style();
style.addRules([aRule, bRule, cRule, dRule, eRule]);
```

But the `OpenLayers.Style` constructor can accept two parameters: a symbolizer hash, to be used as the default style, and a set of options where we need to specify instance properties. With this in mind we can also instantiate the style as:

```
var style = new OpenLayers.Style({
    our_default_style
}, {
    rules: [aRule, bRule, cRule, dRule, eRule]
});
```

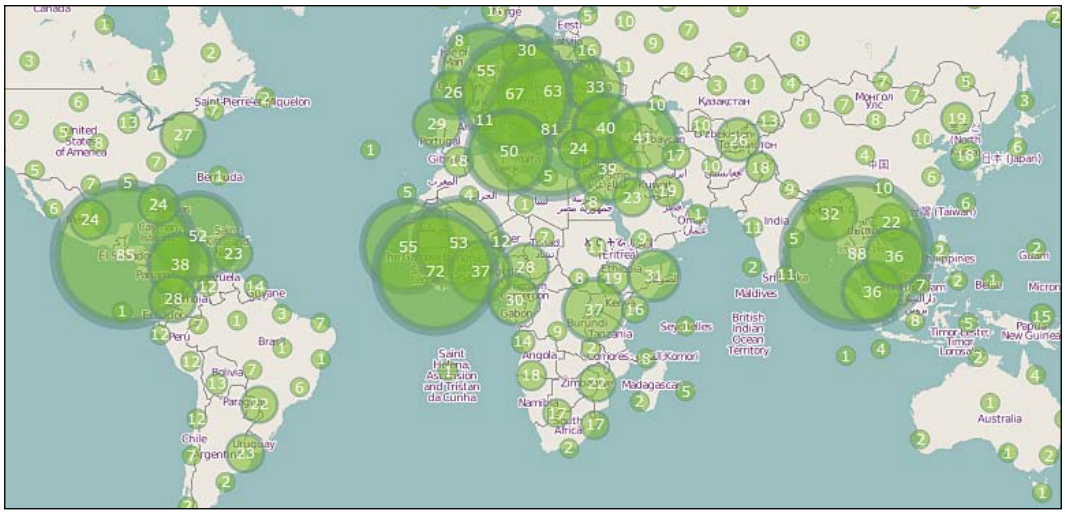
See also

- The *Working with unique value rules* recipe
- The *Styling features using symbolizers* recipe
- The *Improving style using StyleMap and the replacement of feature's attributes* recipe

Styling clustered features

When working with lots of feature points, it is common to use the cluster strategy to avoid overlapping of points and improve the rendering performance.

In this recipe we are going to show how easy it is to style a vector layer using a cluster strategy:



Our layer vector will read a GeoJSON file with some cities of the world. The style will have the following characteristics:

- For each cluster we will show the number of contained features
- The point radius and border will depend on the number of contained features, the more features within it, the greater the radius will be

How to do it...

1. Start adding the `div` element for the map:

```
<div id="ch07_cluster_number_style" style="width: 100%; height:
95%;"></div>
```

2. Instantiate an `OpenLayers.Map` instance:

```
<script type="text/javascript">
    // Create the map using the specified DOM element
    var map = new OpenLayers.Map("ch07_cluster_number_style");
```

3. Add OpenStreetMap as the base layer and center the viewport:

```
    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);

    map.setCenter(new OpenLayers.LonLat(0,20).transform
        (new OpenLayers.Projection("EPSG:4326"),
    new OpenLayers.Projection("EPSG:900913")), 2);
```

4. Load the data from a GeoJSON file and apply the desired style:

```
    var cities = new OpenLayers.Layer.Vector
        ("World Cities (GeoJSON)", {
        protocol: new OpenLayers.Protocol.HTTP({
            url: "http://localhost:8080/openlayers-cookbook/
              recipes/data/world_cities.json",
            format: new OpenLayers.Format.GeoJSON()
        }),
        strategies: [new OpenLayers.Strategy.Fixed(),
          new OpenLayers.Strategy.Cluster({distance: 25})],
        styleMap: new OpenLayers.StyleMap({
            'default': new OpenLayers.Style({
                strokeWidth: '${strokeFunction}',
                strokeOpacity: 0.5,
                strokeColor: "#88aaaa",
                fillColor: "#99CC55",
                fillOpacity: 0.5,
                pointRadius: '${radiusfunction}',
                label: "${count}",
                fontColor: "#ffffff"
            }, {
                context: {
                    strokeFunction: function(feature) {
                        var count = feature.attributes.count;
                        var stk = Math.max(0.1 * count, 1);
                        return stk;
                    },
                    radiusFunction: function(feature) {
                        var count = feature.attributes.count;
                        var radius = Math.max(0.60 * count, 7);
                        return radius;
```

```
                        }
                    }
                })
            })
        });
        map.addLayer(cities);
    </script>
```

How it works...

After creating the map instance and adding the base layer, we have centered the viewport. Note how we have translated the coordinates from EPSG:4326 (latitude/longitude) to EPSG:900913 used by the map (implicitly used by OpenStreetMap layer):

```
    map.setCenter(new OpenLayers.LonLat(0,20).transform(new
OpenLayers.Projection("EPSG:4326"),
    new OpenLayers.Projection("EPSG:900913")), 2);
```

Next, we have added the vector layer:

```
    var cities = new OpenLayers.Layer.Vector("World Cities (GeoJSON)",
{
        protocol: new OpenLayers.Protocol.HTTP({
            url: "http://localhost:8080/openlayers-cookbook/recipes/
data/world_cities.json",
            format: new OpenLayers.Format.GeoJSON()
        }),
        strategies: [new OpenLayers.Strategy.Fixed(), new OpenLayers.
Strategy.Cluster({distance: 25})],
        styleMap: new OpenLayers.StyleMap({
            'default': …
        })
    });
```

To load the GeoJSON file from our server, we have used an `OpenLayers.Protocol.HTTP` instance with the `OpenLayers.Format.GeoJSON` format to read it.

For layer strategies, we have specified the `OpenLayers.Strategy.Fixed` to load the content once and the `OpenLayers.Strategy.Cluster({distance: 25})` sentence to group features. The property `distance` sets the pixel distance that defines where two features must go into the same cluster.

At this point and, before continuing with the recipe, we need to describe how the clustering process works.

When the layer is going to be rendered, the clustering algorithm checks for each feature if they are too close to other ones. For each set of features that are too close, a new point (cluster) is created and rendered. This way the number of points to draw on the map can be reduced drastically. In addition, each cluster point feature will contain references to the set of features it represents and also a `count` attribute with the number of features it contains.

Returning to our code, let's see the style applied to the layer, which is the most important thing in the recipe.

First, we have set the style for the `default` rendering intent:

```
styleMap: new OpenLayers.StyleMap({
    'default': …
})
```

This means if we use some control that changes the render intent of the layer to something different from the default, the style will probably be different.

If we create the `OpenLayers.StyleMap` instance without passing directly the style instance, that is, without specifying a render intent, then the style will be the same for any render intent: `new OpenLayers.StyleMap(our_style_here)`.

Now, let's look at the `OpenLayers.Style` instance defined for the layer:

```
new OpenLayers.Style({
    strokeWidth: '${strokeFunction}',
    strokeOpacity: 0.5,
    strokeColor: "#88aaaa",
    fillColor: "#99CC55",
    fillOpacity: 0.5,
    pointRadius: '${radiusfunction}',
    label: "${count}",
    fontColor: "#ffffff"
}, {
    context: {
        strokeFunction: function(feature) {
            var count = feature.attributes.count;
            var stk = Math.max(0.1 * count, 1);
            return stk;
        },
        radiusFunction: function(feature) {
            var count = feature.attributes.count;
            var radius = Math.max(0.60 * count, 7);
            return radius;
        }
    }
})
```

The constructor receives two parameters: a symbolizer hash, which defines the style properties, and a set of options.

In the symbolizer hash, we have used the attribute replacement feature:

```
strokeWidth: '${strokeFunction}',
...
pointRadius: '${radiusfunction}',
label: "${count}",
….
```

The `count` attribute is taken from the cluster point feature attributes, as we explained previously.

On the other hand, the `strokeFunction` and `radiusFunction` are not attributes, but functions which are defined in the `context` property of the `OpenLayers.Style` options. All the symbolizer properties are evaluated against the `context` object. So, each of the functions receives a feature reference every time the layer is going to be rendered.

In the case of `radiusFunction`, it computes the radius for the point depending on the `count` attribute, returning the maximum value between the range of 60 percent of `count` or `7`:

```
radiusFunction: function(feature) {
    var count = feature.attributes.count;
    var radius = Math.max(0.60 * count, 7);
    return radius;
}
```

As we can see, the use of context is powerful enough to allow us to set style properties dynamically and dependence on other feature attributes.

See also

- The *Improving style using StyleMap and the replacement of feature's attributes* recipe
- The *Defining custom rules to style features* recipe
- The *Using the cluster strategy* recipe in *Chapter 3*, *Vector Layers*

8
Beyond the Basics

In this chapter we will cover:

- Working with projections
- Requesting remote data with `OpenLayers.Request`
- Creating a custom control
- Creating a custom renderer
- Selecting features intersecting with a line
- Making an animation with image layers

Introduction

OpenLayers is a big and complex framework. There is no other option available for a framework that allows working with many GIS standards, reading from many different data sources, rendering on different browser technologies, and so on. This power comes with a price.

The implementation of OpenLayers tries to have as less dependencies on external libraries as possible. This means, OpenLayers requires implementing many features that we can find in other projects: DOM elements' manipulation, AJAX requests, and so on.

This chapter shows some of these features, in addition to other possible common needs we can require in our day-to-day work that are not explained in other chapters, such as creation of layer animations or the implementation of custom controls. Because of this, the chapter is more suited for more experienced JavaScript programmers.

Working with projections

In contrast to other JavaScript mapping libraries, OpenLayers allows working with a great number of projections.

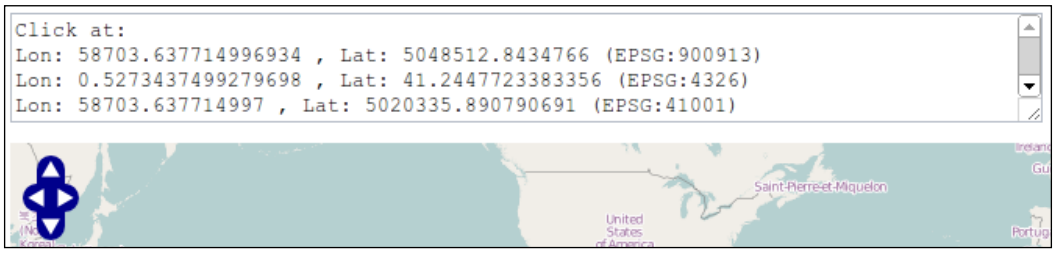
Usually, we specify the desired projection for the map. Later when adding a vector layer to the map, we need to specify to the layer projection so that OpenLayers transforms features from the layer's projection to the map's projection.

But, by default, OpenLayers has a great limitation on projections: we can only use **EPSG:4326** and **EPSG:900913**. Why? Because transforming between projections is not a simple task and there are other great projects that can make it.

So, when we want to work with projections other than EPSG:4326 and EPSG:900913, OpenLayers uses **Proj4js Library** (`http://trac.osgeo.org/proj4js`).

Teaching about projections is out of the scope of this book. The EPSG codes are simply a standardized way to classify and identify the great amount of available projections. EPSG:4326 corresponds to the WGS84 (World Geodetic System) and EPSG:900913 is the Spherical Mercator projection popularized by their use in Google Maps.

Let's go to see how we can integrate Proj4js with OpenLayers and how easy it is to make use of it. The idea is to create an application that shows a map and a text area that will show the coordinates of the clicked location:



Getting ready

We must place some of the available Proj4js files at our web application directory. To do so, perform the following steps:

- Go to the Proj4js project's web page and download the distribution ZIP file (for this recipe we have used `http://download.osgeo.org/proj4js/proj4js-1.1.0.zip`)

- Uncompress the downloaded file and copy the `proj4js-compressed.js` file and `defs` folder within your web application folder

How to do it...

1. Create an HTML file and add the OpenLayers dependencies. As a dependency, also include the `Proj4js` library:

```
<script type="text/javascript" src="./js/proj4js-1.1.0/proj4js-
compressed.js"></script>
```

2. Now add the code for the text area and the map:

```
<textarea id="textarea" name="textarea" data-dojo-type="dijit.
form.SimpleTextarea" rows="4" cols="80"></textarea>
<br/><br/>
<div id="ch08_projections" style="width: 100%; height: 85%;"></
div>
```

3. In the JavaScript section, create a new control to manage the click event:

```
<script type="text/javascript">
    // Create the click control
    OpenLayers.Control.Click = OpenLayers.Class(OpenLayers.
Control, {

        defaultHandlerOptions: {
            'single': true,
            'double': false,
            'pixelTolerance': 0,
            'stopSingle': false,
            'stopDouble': false
        },

        initialize: function(options) {
            this.handlerOptions = OpenLayers.Util.extend({}, this.
defaultHandlerOptions);
            OpenLayers.Control.prototype.initialize.apply(this,
arguments);
            this.handler = new OpenLayers.Handler.Click(
            this, {
                'click': this.trigger
            },
            this.handlerOptions);
        },
```

4. On the `trigger` function, add the following code to transform and show the coordinates in the `textarea` object:

```
        trigger: function(e) {
            var lonlatS = map.getLonLatFromViewPortPx(e.xy);
            var lonlatT1 = lonlatS.clone().transform( map.
getProjectionObject(), new OpenLayers.Projection("EPSG:41001") );
            var lonlatT2 = lonlatS.clone().transform( map.
getProjectionObject(), new OpenLayers.Projection("EPSG:4326") );

            var message = "Click at: \n"+
                "Lon: " + lonlatS.lon + " , Lat: "+lonlatS.lat + "
("+map.getProjection()+")\n" +
                "Lon: " + lonlatT2.lon + " , Lat: "+lonlatT2.lat +
" (EPSG:4326) \n" +
                "Lon: " + lonlatT1.lon + " , Lat: "+lonlatT1.lat +
" (EPSG:41001) \n";

            dijit.byId("textarea").set('value', message);
        },

        CLASS_NAME: "OpenLayers.Control.Click"
    });
```

5. Create the map instance, add a base layer, and center the viewport:

```
    var map = new OpenLayers.Map("ch08_projections");
    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);
    map.setCenter(new OpenLayers.LonLat(0,0), 2);
```

6. Finally, create a new click control instance and add it to the map:

```
    var click = new OpenLayers.Control.Click();
    map.addControl(click);
    click.activate();
</script>
```

How it works...

OpenLayers makes use of the Proj4js code internally when available. So as OpenLayers developers we do not need to use the Proj4js API directly, the only requirement is to add the Proj4js dependency in our application:

```
<script type="text/javascript" src="./js/proj4js-1.1.0/proj4js-
compressed.js"></script>
```

When the user clicks at some place on the map, the click control (that we will see later) executes the `trigger` function. The `e` variable contains all the click event's information that includes the pixel's xy position.

```
        trigger: function(e) {
            var lonlatS = map.getLonLatFromViewPortPx(e.xy);
            var lonlatT1 = lonlatS.clone().transform( map.
getProjectionObject(), new OpenLayers.Projection("EPSG:41001") );
            var lonlatT2 = lonlatS.clone().transform( map.
getProjectionObject(), new OpenLayers.Projection("EPSG:4326") );
            …
            ...
```

Ensure that the projection definition you are using, is defined within the `defs` folder. Otherwise you will need to create a new file with the transformation expressed in the proj4 notation.

Given an `OpenLayers.LonLat` instance, we can translate among projections using the `tranform()` method.

We always can make use of the `transform()` method but without including the Proj4js dependencies, they will only translate between EPSG:4326 and EPSG:900913.

Thanks to the `OpenLayers.Map.getLonLatFromViewPortPx()` method we can go from `OpenLayers.Pixel` to the `OpenLayers.LonLat` instance.

Because the transform method modifies the current instance, we create a new one using the `clone()` method to avoid modifying the source variable.

At this point, the `trigger` method can construct a message string and place it within the text area.

Finally, let's briefly describe the click control used in the recipe.

The first step is to define the new control as a subclass of the `OpenLayers.Control` class:

```
OpenLayers.Control.Click = OpenLayers.Class(OpenLayers.Control, {
```

The control will use an `OpenLayers.Handler`, so here we will define some options:

```
        defaultHandlerOptions: {
            'single': true,
            'double': false,
            'pixelTolerance': 0,
            'stopSingle': false,
            'stopDouble': false
        },
```

The `initialize` method is responsible to initialize the control instance. First, we create a set of options as a combination (using `OpenLayers.Util.extend()` method) of the previously defined object and options passed by the user as arguments:

```
        initialize: function(options) {
            this.handlerOptions = OpenLayers.Util.extend({}, this.
defaultHandlerOptions);
            OpenLayers.Control.prototype.initialize.apply(this,
arguments);
            this.handler = new OpenLayers.Handler.Click(
            this, {
                'click': this.trigger
            },
            this.handlerOptions);
        },
```

We have initialized an `OpenLayers.Handler.Click` instance to execute the `trigger` listener function every time it detects that the user has pressed the mouse button.

Finally, as a good practice we set the `CLASS_NAME` attribute with a string identifying our new control class:

```
        CLASS_NAME: "OpenLayers.Control.Click"
    });
```

See also

- The *Playing with the map's options* recipe in *Chapter 1*, *Web Mapping Basics*
- The *Creating features programmatically* recipe in *Chapter 3*, *Working with Vector Layers*

Retrieving remote data with OpenLayers.Request

Data is the basis for a web mapping application. We can add raster or vector layers to the map, which will load images or vector information.

In the case of vector layers, thanks to the `OpenLayers.Protocol` and `OpenLayers.Format` subclasses, we can configure the layer to load data from different sources and with different formats.

Anyway, there can be circumstances where we need to request data by ourselves, read the specific format, and add features. We are talking about making asynchronous JavaScript calls.

This recipe shows how we can use the helper class `OpenLayers.Request` to asynchronously request data from the remote servers.

Here, we are going to request a URL that returns random x and y values that we will process as point features on the map.

OpenLayers is a framework for GIS web developers, so it is designed to be independent from other projects, such as jQuery and Dojo that offer facilities to request remote data and implement its own.

How to do it...

1. Once created the HTML file with OpenLayers library dependencies, add a `div` element to hold the map:

```
<div id="ch08_requesting" style="width: 100%; height: 95%;"></div>
```

2. In the JavaScript section, create the map instance, add a base layer, and center the viewport:

```
var map = new OpenLayers.Map("ch08_requesting");
var osm = new OpenLayers.Layer.OSM();
map.addLayer(osm);

// Center viewport
map.setCenter(new OpenLayers.LonLat(0,0), 2);
```

3. Create a vector layer and add to the map:

```
var vectorLayer = new OpenLayers.Layer.Vector("Points");
map.addLayer(vectorLayer);
```

4. Finally, make a request to the `points.php` utility code, which returns a set of random x and y values:

```
    OpenLayers.Request.GET({
        url: "utils/points.php",
        params: {
            num: 100
        },
        success: function(response) {
            var format = new OpenLayers.Format.JSON();

            var points = format.read(response.responseText);
            for(var i=0; i< points.length; i++) {
                var p = new OpenLayers.Geometry.Point(points[i].x,
points[i].y);
                p.transform(new OpenLayers.Projection("EPSG:4326"),
new OpenLayers.Projection("EPSG:900913"));

                var f = new OpenLayers.Feature.Vector(p);
                vectorLayer.addFeatures([f]);
            }
        },
        failure: function(response) {
            alert("Sorry, there was an error requesting data
!!!");
        }
    });
```

How it works...

In JavaScript, the `XMLHttpRequest` object allows us to communicate with the server side.

More information about working with `XMLHttpRequest` can be found on `http://acuriousanimal.com/blog/2011/01/27/working-with-the-javascript-xmlhttprequest-object` and `https://developer.mozilla.org/en/AJAX/Getting_Started`.

Due to compatibility problems among browsers, OpenLayers uses a cross-browser W3C compliant version of the `XMLHttpRequest` object and wrapping it has implemented the `OpenLayers.Request` class.

`OpenLayers.Request` class implements the HTTP methods: `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, and `OPTIONS`, and is used by other OpenLayers classes to get/send data from/to remote servers (such as `OpenLayers.Protocol.HTTP`).

An HTTP introduction can be found at: `https://developer.mozilla.org/en/HTTP`

In this recipe we have used the `OpenLayers.Request.GET` method with the following parameters:

- `url`: It is the URL we are going to request
- `params`: It is a set of options parameters we can send in the `GET` request
- `success`: It is a callback function to be executed if the URL is successfully requested
- `failure`: It is a callback function to be executed if any problem occurs

In our code, we are requesting the `utils/points.php` passing a `num` parameter, this is the same as requesting the URL `utils/points.php?num=100`:

```
OpenLayers.Request.GET({
    url: "utils/points.php",
    params: {
        num: 100
    },
    success: function(response) {
        …
    },
    failure: function(response) {
        …
    }
});
```

If for some reason the request fails, the method `failure` is executed and will show an alert message. On the other hand if the request succeeds we read the returned response and add point features to the map.

The `points.php` script returns a random number of x and y values, depending on the `num` parameter, encoded as JSON array. The response of the call is nothing more than a text that must be interpreted and we can find it in the `responseText` of the response's property.

To convert the JSON string into a JavaScript object we can use the class `OpenLayers.Format.JSON`:

```
var format = new OpenLayers.Format.JSON();
var points = format.read(response.responseText);
```

Finally, for each object we read the x and y values and create a point feature:

```
            for(var i=0; i< points.length; i++) {
                var p = new OpenLayers.Geometry.Point(points[i].x,
points[i].y);
                p.transform(new OpenLayers.Projection("EPSG:4326"),
new OpenLayers.Projection("EPSG:900913"));

                var f = new OpenLayers.Feature.Vector(p);
                vectorLayer.addFeatures([f]);
            }
```

The x and y values returned by the PHP scripts goes from -180 to180 for x and -80 to 80 for y. Because of this, we translate the coordinates from EPSG:4326 to EPSG:900913, which is the map's base layer projection.

There's more...

`OpenLayers.Request` is a powerful class allowing working with almost any HTTP method. For example, in addition to the `GET` method we can also use the `POST` method to send data to servers.

If you are going to work extensively with AJAX in your application, be sure to understand the limitations of **Cross-Domain Requests** (**XDR**) and the **same origin policy** (`http://en.wikipedia.org/wiki/Same_origin_policy`).

Finally, take a close look at `OpenLayers.Request` class options. You can find options, such as `async` to specify if the request must be made synchronously or asynchronously, `user/password` to make requests against servers with basic authentication, or `headers` to set the HTTP headers of the request.

See also

- The *Reading features using Protocols directly* recipe in *Chapter 3*, *Working with Vector Layers*
- The *Reading and creating features from a WKT* recipe in *Chapter 3*, *Working with Vector Layers*

Creating a custom control

OpenLayers has plenty of controls that address a broad range of needs. Unfortunately, the requirements we could have for building a new web application can imply the creation of a new one, or the extension of a previous one:

In this recipe, we are going to create a new control named **Cross**. The control will show a crosshair symbol, as shown in the previous screenshot, similar to the target selectors in the ancient war planes, which will show the location it is pointing to. In addition, the control will allow registering the click events that will return the current location too.

How to do it...

1. Create an HTML file and add the OpenLayers dependencies, then include the code of our new control:

```
<script type="text/javascript" src="./recipes/ch08/crossControl.
js"></script>
```

2. Next, add the two CSS classes required for the control:

```
<style>
    .olControlCross {
        width: 48px;
        height: 48px;
        background: url('./recipes/data/target.png') no-repeat;
    }
    .olControlCrossText {
        position: relative;
```

```
        top: -10px;
        width: 200px;
        color: black;
    }
</style>
```

3. Now, add a `div` element to hold the map:

```
<div id="ch08_drawing_cross" style="width: 100%; height: 95%;"></
div>
```

4. Within the JavaScript code, create the map instance and add a base layer:

```
    var map = new OpenLayers.Map("ch08_drawing_cross");
    var layer = new OpenLayers.Layer.WMS( "OpenLayers WMS",
    "http://vmap0.tiles.osgeo.org/wms/vmap0", {layers: 'basic'} );
    map.addLayer(layer);

    // Center viewport
    map.setCenter(new OpenLayers.LonLat(0,0), 2);
```

5. Create the cross control, add it to the map, and activate it:

```
    var crossControl = new OpenLayers.Control.Cross({
        eventListeners: {
            "crossClick": function(event) {
                var lonlat = event.lonlat;
                var message = "Clicked on: " + lonlat.lon + " / "
+ lonlat.lat;
                alert(message);
            }
        }
    });
    map.addControl(crossControl);
    crossControl.activate();
```

6. Now, we are going to describe step by step the source code of the new control we have created. First, create a new `crossControl.js` file and start applying the *best* practice of writing a description about the control:

```
/**
 * Class: OpenLayers.Control.Cross
 * The Cross control renders a cross in the middle of the map.
 *
 * Inherits from:
 *  - <OpenLayers.Control>
 */
```

7. Next, create the new `OpenLayers.Control.Cross` class as a subclass of `OpenLayers.Control`:

```
OpenLayers.Control.Cross = OpenLayers.Class(OpenLayers.Control, {
```

8. Define the set of attributes and methods of the new control. The first step is to initialize an array with the set of events our control can emit:

```
    /**
     * crossClick event is triggered when the cross is clicked by
the mouse.
     */
    EVENT_TYPES: ["crossClick"],
```

9. Next, there is a `size` property, that is used to know the image control size and required to compute the exact control location:

```
    /**
     * Parameter: size
     * {OpenLayers.Size} with the desired dimension for the image
     */
    size: null,
```

10. The last attribute is used to store a reference to the DOM element used as a label to show the current control target's location:

```
    /**
     * Parameter: element
     * {DOMElement} for the label shown by the control
     */
    element: null,
```

11. Once we have defined all the required properties used in the class, we need to initialize the control. Again, it is a good practice to comment in the source code following the OpenLayers conventions:

```
    /**
     * Constructor: OpenLayers.Control.Cross
     * Draw a cross in the middle of the map.
     *
     * Parameters:
     * options - {Object} An optional object whose properties will
be used
     *     to extend the control.
     */
    initialize: function(options) {
        // Concatenate events specific to measure with those from
the base
        this.EVENT_TYPES =
        OpenLayers.Control.Cross.prototype.EVENT_TYPES.concat(
```

```
                OpenLayers.Control.prototype.EVENT_TYPES);

            if(!options) {
                options = {};
            }
            if(!options.size) {
                options.size = new OpenLayers.Size(48, 48);
            }
            OpenLayers.Control.prototype.initialize.apply(this,
[options]);
        },
```

12. Next, we need to implement the `draw` method, which is called when the control is ready to be displayed on the page and is responsible to set the required DOM elements to render the control. The first step involves computing the right position for the control, which is the middle of the map:

```
        /**
         * Method: draw
         *
         * Returns:
         * {DOMElement}
         */
        draw: function() {

            // Compute center position
            var position = new OpenLayers.Pixel(
                (this.map.div.offsetWidth - this.size.w) / 2,
                (this.map.div.offsetHeight - this.size.h) / 2
                );
```

13. Then, we can call the `draw` method of the superclass to draw the control. This will initialize the `this.div` property (inherited from `OpenLayers.Control`) with the DOM element that will hold the control. By default a CSS class `olControlCross` is added to the `div` element, so we can style it easily:

```
            OpenLayers.Control.prototype.draw.apply(this, [position]);
```

14. After this, we can create a new `div` element for the label that will show the current target's location. This is done by using the method `OpenLayers.Util.createDiv`. In addition, thanks to the `OpenLayers.Element.addClass` method, we set the CSS class `olControlCrossText` to the label so the user can style the label:

```
            // Create location label element
            this.element = OpenLayers.Util.createDiv(null);
            OpenLayers.Element.addClass(this.element,
"olControlCrossText");
```

15. Compute the current `OpenLayers.LonLat` position and set the label text. We omit the code for the `computeLonLat` function that can be found within the control class:

```
        var lonlat = this.computeLonLat();();
        this.element.innerHTML = lonlat.lon + " / " + lonlat.lat;
```

16. Add the label element to the main control element:

```
        this.div.appendChild(this.element);
```

17. As a final step, we register two listeners. First, a listener for the `this.div` element to detect when the mouse clicks the control:

```
        // Listen for event in the control's div
        OpenLayers.Event.observe(this.div, 'click', OpenLayers.
Function.bind(this.onClick, this));
```

18. Second, a listener for the map's `move` event, so we can update the control's location label:

```
        // Register event for map's move event.
        this.map.events.register("move", this, this.onMove);
```

19. And finally, return a reference to the `this.div` element:

```
        return this.div;
    },
```

20. Next is the code for two listeners. The `onMove` method updates the label's text (target's location) each time the map is moved:

```
    /**
     * Updates the location text.
     */
    onMove: function (event) {
        var lonlat = this.computeLonLat();
        this.element.innerHTML = lonlat.lon + " / " + lonlat.lat;
    },
```

21. The `onClick` function is executed when there is a mouse click on the control. Its responsibility is to trigger the `crossClick` event so that any outside listener can by notified:

```
    /**
     * Fires a crossClick event.
     */
    onClick: function (event) {
        var lonlat = this.computeLonLat();
        this.events.triggerEvent("crossClick", {
            lonlat: lonlat
        });
    },
```

22. This is the code for the helper function that computes the `OpenLayers.LonLat` from the current control's pixel position:

```
    /**
     * Computes the control location.
     *
     * Returns:
     * {<OpenLayers.LonLat>}
     */
    computeLonLat: function() {
        var pixel = this.position.clone();
        pixel.x += this.size.w/2;
        pixel.y += this.size.h/2;
        return this.map.getLonLatFromPixel(pixel);
    },
```

23. Last, but not the least, we have to set the property `CLASS_NAME` with a string identifying the control name. By convention, it is the whole namespace of the control:

```
    CLASS_NAME: "OpenLayers.Control.Cross"
});
```

How it works...

The program does not have much mystery, in addition to the base layer we have created a cross control:

```
var crossControl = new OpenLayers.Control.Cross({
    eventListeners: {
        "crossClick": function(event) {
            var lonlat = event.lonlat;
            var message = "Clicked on: " + lonlat.lon + " / " +
lonlat.lat;
            alert(message);
        }
    }
});
```

We have initialized the cross control by registering a listener function on the `crossClick` event, which is triggered each time there is a mouse click on the cross image.

There's more...

Note how we can create the new control class:

```
OpenLayers.Control.Cross = OpenLayers.Class(OpenLayers.Control, {
    …
});
```

New classes or subclasses are easily created with `OpenLayers.Class`. It requires two parameters:

- The **source** class
- An **object** with the class definition that will extend the source class

In our code, we are extending the `OpenLayers.Control` class with the properties and functions defined in the second parameter defined in the object literal notation.

In addition, any class must be initialized using the `initialize` method. The usual order of actions to be done is:

1. Merge the array of `EVENT_TYPES` controls with those defined in the base `OpenLayers.Control` class. This means we are extending the base event types with the set defined in the new cross control:

```
    initialize: function(options) {
        // Concatenate events specific to measure with those from
the base
        this.EVENT_TYPES =
        OpenLayers.Control.Cross.prototype.EVENT_TYPES.concat(
            OpenLayers.Control.prototype.EVENT_TYPES);
```

2. Set a default value for the instance properties if they are not defined in the constructor:

```
        if(!options) {
            options = {};
        }
        if(!options.size) {
            options.size = new OpenLayers.Size(48, 48);
        }
```

3. Call the superclass constructor. Once we have the subclass initialized, we need to initialize the superclass. This is a bottom-top initialization:

```
        OpenLayers.Control.prototype.initialize.apply(this,
[options]);
    },
```

See also

- The *Adding and removing controls* recipe in *Chapter 5*, *Adding Controls*
- The *Listening for non OpenLayers events* recipe in *Chapter 4*, *Working with Events*
- The *Adding the WMS layer* recipe in *Chapter 2*, *Adding Raster Layers*

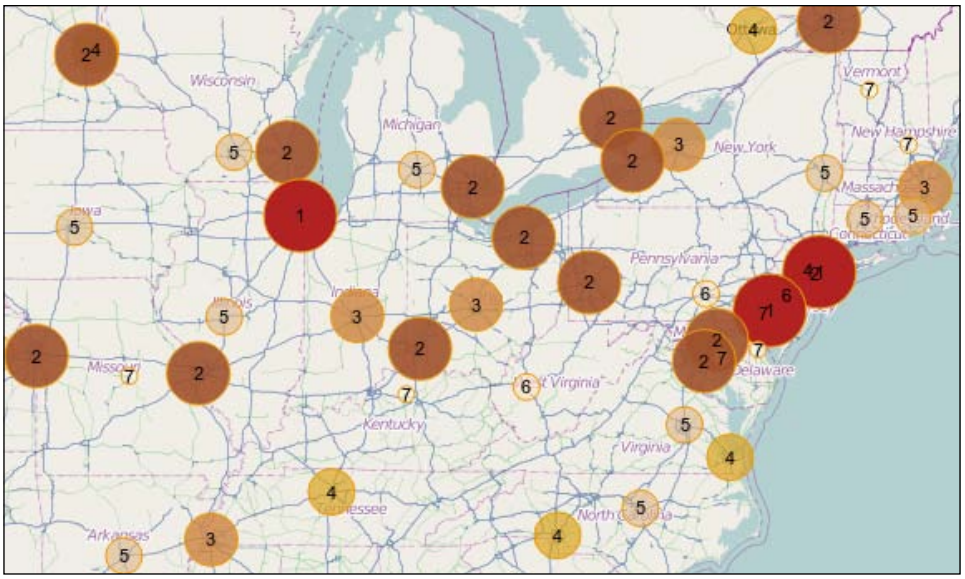
Creating a custom renderer

When working with vector layers, styling is a great feature which offers us a lot of possibilities: fill color and opacity, stroke color, labels and text colors, and so on. But, what if we need more?

Every `OpenLayers.Layer.Vector` instance contains a renderer that is responsible to render the layer's features (such as points, paths, and polygons) on the map using the best technologies available in the browser. These can be the HTML5 Canvas element (`http://en.wikipedia.org/wiki/Canvas_element`) available in many modern browsers (such as Firefox or Chrome), SVG (`http://en.wikipedia.org/wiki/Scalable_Vector_Graphics`), or VML (`http://en.wikipedia.org/wiki/Vector_Markup_Language`).

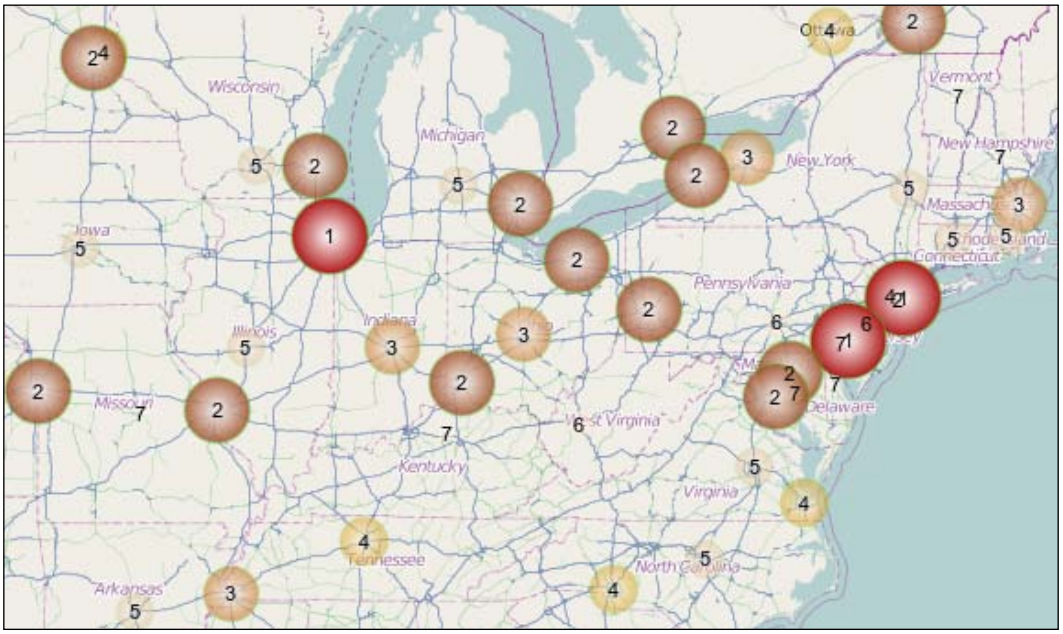
When `OpenLayers.Layer.Vector` is initialized, OpenLayers looks for the best available rendering engine and creates an instance of `OpenLayers.Renderer` that will render the features on the map.

The goal of this recipe is to show how we can create a new renderer to improve the visualization of the features.



The previous screenshot shows some styled point geometry features and was rendered using the default OpenLayers renderer.

The following screenshot shows the same features rendered with our new renderer implementation:



We are going to extend `OpenLayers.Renderer.Canvas`, to improve visualizations. This renderer works using the HTML5 Canvas element. This means the new renderer only will work on HTML5 compliant browsers. You can check if the canvas element is supported by your browser at: `http://html5test.com`

How to do it...

1. Once an HTML file is created with OpenLayers dependencies, the first step is to include the file with the new renderer class' implementation:

```
<script type="text/javascript" src="./recipes/ch08/
gradientRenderer.js"></script>
```

2. Now, as usual you can add the `div` element that will hold the map:

```
<div id="ch08_renderer" style="width: 100%; height: 95%;"></div>
```

3. In the JavaScript section, add the following code to initialize the map and add a base layer:

```
    var map = new OpenLayers.Map("ch08_renderer");
    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);
```

4. Center the map's viewport:

```
    var center = new OpenLayers.LonLat(-80,40);
    center.transform(new OpenLayers.Projection("EPSG:4326"), new
OpenLayers.Projection("EPSG:900913"));
    map.setCenter(center, 5);
```

5. Create a `StyleMap` instance with a **unique value rule** based on the `POP_RANK` attribute of the features:

```
    var styles = {
        7: { pointRadius: 5, label: "${POP_RANK}", fillColor:
"#FFF8DC", fillOpacity: 0.6},
        6: { pointRadius: 8, label: "${POP_RANK}", fillColor:
"#FFE4C4", fillOpacity: 0.6},
        5: { pointRadius: 11, label: "${POP_RANK}", fillColor:
"#DEB887", fillOpacity: 0.6},
        4: { pointRadius: 14, label: "${POP_RANK}", fillColor:
"#DAA520", fillOpacity: 0.7},
        3: { pointRadius: 16, label: "${POP_RANK}", fillColor:
"#CD853F", fillOpacity: 0.8},
        2: { pointRadius: 19, label: "${POP_RANK}", fillColor:
"#A0522D", fillOpacity: 0.9},
        1: { pointRadius: 22, label: "${POP_RANK}", fillColor:
"#B22222", fillOpacity: 1.0}
    };

    var styleMap = new OpenLayers.StyleMap();
    styleMap.addUniqueValueRules("default", "POP_RANK", styles);
```

6. Create the vector layer and add it to the map:

```
    var vectorLayer = new OpenLayers.Layer.Vector("Cities", {
        styleMap: styleMap,
        renderers: ["Gradient"],
        protocol: new OpenLayers.Protocol.HTTP({
            url: "http://localhost:8080/openlayers-cookbook/
recipes/data/world_cities.json",
            format: new OpenLayers.Format.GeoJSON()
        }),
        strategies: [new OpenLayers.Strategy.Fixed()]
    });
    map.addLayer(vectorLayer);
```

7. Let's go to see the `OpenLayers.Renderer.Gradient` implementation that beautifies our point's features that are rendered with a nice gradient style. Start creating a JavaScript file named `gradientRenderer.js`, which we have included previously in the main program. Following good practices, we start the commenting in the file:

```
/**
 * Class: OpenLayers.Renderer.Gradient
 * Improved canvas based rendered to draw points using gradient.
 *
 * Inherits:
 *  - <OpenLayers.Renderer.Canvas>
 */
```

8. Now, create an `OpenLayers.Renderer.Canvas` subclass named `OpenLayers.Renderer.Gradient`:

```
OpenLayers.Renderer.Gradient = OpenLayers.Class(OpenLayers.
Renderer.Canvas, {
```

9. The first method to implement in a new OpenLayers class must be the `initialize()` method:

```
    /**
     * Constructor: OpenLayers.Renderer.Gradient
     *
     * Parameters:
     * containerID - {<String>}
     * options - {Object} Optional properties to be set on the
renderer.
     */
    initialize: function(containerID, options) {
        OpenLayers.Renderer.Canvas.prototype.initialize.
apply(this, arguments);
    },
```

10. Next, we implement the `drawPoint()` method, inherited from the `OpenLayers.Renderer.Canvas` class, which is responsible to render the point geometry features of the layer. Now, the method receives three parameters: the geometry object, the style to apply, and the feature identifier attribute:

```
    /**
     * Method: drawPoint
     * This method is only called by the renderer itself.
     *
     * Parameters:
     * geometry - {<OpenLayers.Geometry>}
     * style    - {Object}
     * featureId - {String}
     */
    drawPoint: function(geometry, style, featureId) {
```

11. From the `geometry` parameter, compute the exact pixel position of the point. This can be done with the `getLocalXY()` method inherited from the `OpenLayers.Renderer.Canvas` class:

```
        var pt = this.getLocalXY(geometry);
        var p0 = pt[0];
        var p1 = pt[1];

        if(!isNaN(p0) && !isNaN(p1)) {
            if(style.fill !== false) {
                this.setCanvasStyle("fill", style);
```

12. Using the `fillColor` and `fillOpacity` properties create a string for the color to be applied for the gradient:

```
                // Create color from fillColor and fillOpacity
properties.
                var color = style.fillColor;
                color += "ff";
                color = color.replace("#", "0x");

                var colorRGBA = 'rgba(' +
                ((color >> 24) & 0xFF) + ',' +
                ((color >> 16) & 0xFF) + ',' +
                ((color >>  8) & 0xFF) + ',' +
                style.fillOpacity + ')';
```

13. Then create a canvas gradient, centered in the feature's location and with the radius value specified in the feature's style:

```
                var gradient = this.canvas.
createRadialGradient(p0, p1, 0, p0, p1, style.pointRadius);
```

14. Define the necessary steps so that the gradient goes from white to the previously created RGB color:

```
                gradient.addColorStop(0, '#FFFFFF');
                gradient.addColorStop(0.9, colorRGBA);
                gradient.addColorStop(1, 'rgba(1,255,0,0)');

                this.canvas.fillStyle = gradient;
                this.canvas.fillRect(0, 0, this.root.width, this.
root.height);

                this.canvas.fill();
            }
        }
    },
```

15. Finally, identify the new class by setting the `CLASS_NAME` property:

```
        CLASS_NAME: "OpenLayers.Renderer.Gradient "
    });
```

How it works...

The important point of this recipe resides in one of the properties we have specified for the vector layer, the `renderers`.

The `renderers` property allows us to specify the set of `OpenLayers.Renderer` that the layer can make use of. Usually this property is never used and by default, its value is: `renderers: ['SVG', 'VML', 'Canvas']`. This means the supported renderer instances the layer can use are `OpenLayers.Renderer.SVG`, `OpenLayers.Renderer.VML`, and `OpenLayers.Renderer.Canvas`.

For this recipe, we have created the class `OpenLayers.Renderer.Gradient`, which we will describe later. The setting `renderers: ["Gradient"]` means we only want to allow the layer to work with an `OpenLayers.Renderer.Gradient` instance.

Let's describe in more detail how to initialize the vector layer:

```
    var vectorLayer = new OpenLayers.Layer.Vector("Cities", {
          styleMap: styleMap,
          renderers: ["Gradient"],
          protocol: new OpenLayers.Protocol.HTTP({
              url: "http://localhost:8080/openlayers-cookbook/recipes/
data/world_cities.json",
              format: new OpenLayers.Format.GeoJSON()
          }),
          strategies: [new OpenLayers.Strategy.Fixed()]
      });
```

In addition to the renderers, we have used an `OpenLayers.Protocol.HTTP` instance with an `OpenLayers.Format.GeoJSON` instance, to load a GeoJSON file with some cities around the world. The features within the file have, among others, the `POP_RANK` attribute.

Thanks to the `OpenLayers.Strategy.Fixed` strategy instance, the layer loads the data source, through the previous protocol, only once. We have no need to load the file each time the map is zoomed.

Last, but not the least, we have set the `styleMap` property to a previously created `OpenLayers.StyleMap` instance:

```
      var styleMap = new OpenLayers.StyleMap();
      styleMap.addUniqueValueRules("default", "POP_RANK", styles);
```

This style map is defined making use of the unique value rule feature based on the `POP_RANK` attribute. This property takes values from `1` to `7`, so we define a symbolizer hash style for each possible value, playing with the radius and fill color properties:

```
    var styles = {
        7: { pointRadius: 5, label: "${POP_RANK}", fillColor:
"#FFF8DC", fillOpacity: 0.6},
        6: { pointRadius: 8, label: "${POP_RANK}", fillColor:
"#FFE4C4", fillOpacity: 0.6},
        5: { pointRadius: 11, label: "${POP_RANK}", fillColor:
"#DEB887", fillOpacity: 0.6},
        4: { pointRadius: 14, label: "${POP_RANK}", fillColor:
"#DAA520", fillOpacity: 0.7},
        3: { pointRadius: 16, label: "${POP_RANK}", fillColor:
"#CD853F", fillOpacity: 0.8},
        2: { pointRadius: 19, label: "${POP_RANK}", fillColor:
"#A0522D", fillOpacity: 0.9},
        1: { pointRadius: 22, label: "${POP_RANK}", fillColor:
"#B22222", fillOpacity: 1.0}
    };
```

For the main program, there is nothing more to comment, except for the way we have centered the map's viewport.

Because the base layer is `OpenStreetMap`, which by default uses an EPSG:900913 projection, and we have specified the center in the EPSG:4326 projection, we need to transform the coordinates to the appropriate map's projection.

There's more...

Usually, this `initialize` method starts calling the `initialize` method of its parent class and then sets the concrete properties of the instance.

In this case, our class has no specific property to initialize, so it is not strictly necessary to implement this method, but as an example, (and as a good practice) we have written the call to the parent class:

```
    initialize: function(containerID, options) {
        OpenLayers.Renderer.Canvas.prototype.initialize.apply(this,
arguments);
    },
```

The process followed within the `OpenLayers.Renderer.Canvas` class to render the features of a layer is a bit complex, but can be summarized as:

- For each feature the class checks its geometry
- Depending on the geometry type the class invokes a different method specially designed to render points, lines, or polygons

Because our renderer is implemented to beautify points we only have rewritten the `drawPoint` method, which is responsible to render point geometries.

The renderer we have defined here uses the HTML5 canvas element, because of this, the main part of the recipe is related to this technology.

Lots of information about the HTML5 canvas element can be found on the Internet. We want to point to this tutorial from the Mozilla project: `https://developer.mozilla.org/en/Canvas_tutorial`.

A great exercise would be to create an SVG version of this renderer. This way, the possibility to render gradient points would be available in more browsers.

See also

- The *Creating a custom control* recipe
- The *Styling features using symbolizers* recipe in *Chapter 7*, *Styling Features*
- The *Defining custom rules to style features* recipe in *Chapter 7*, *Styling Features*

Selecting features intersecting with a line

One common action when working with features within a vector layer is its selection and, of course, OpenLayers has some feature selection controls.

The `OpenLayers.Control.SelectFeature` control is specially useful as a selection control because the selection is made on the client side, that is, the selection is made through the features loaded in the browser. There is no request to a WFS server.

The `OpenLayers.Control.SelectFeature` control can work in different ways. We can select a feature just by clicking on it or we can draw a box to select all the contained features.

In contrast, it does not allow the possibility to select features that intersect with a path.

In this recipe, we are going to see how we can extend the `OpenLayers.Control.SelectFeature` control to allow select features for drawing a path.

How to do it...

1. Create an HTML5 file and add the OpenLayers library dependencies. Now, include the code for the new `OpenLayers.Control.SelectFeature` control:

```
<script type="text/javascript" src="./recipes/ch08/
selectFeaturePath.js"></script>
```

2. Within the `body` section, add a `div` element to hold the map instance:

```
<div id="ch08_selecting" style="width: 100%; height: 95%;"></div>
```

3. Next, place the JavaScript code within a `script` element at the document's `head` element:

```
<script type="text/javascript">
```

4. Create the map instance, add a base layer, and center the viewport:

```
    var map = new OpenLayers.Map("ch08_selecting");
    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);

    // Center viewport
    var center = new OpenLayers.LonLat(25,50);
    center.transform(new OpenLayers.Projection("EPSG:4326"), new
OpenLayers.Projection("EPSG:900913"));
    map.setCenter(center, 4);
```

5. Now, create a vector layer that loads a GML file with the European countries:

```
    var vectorLayer = new OpenLayers.Layer.Vector("Europe", {
        protocol: new OpenLayers.Protocol.HTTP({
            url: "http://localhost:8080/openlayers-cookbook/
recipes/data/europe.gml",
            format: new OpenLayers.Format.GML()
        }),
        strategies: [new OpenLayers.Strategy.Fixed()]
    });
    map.addLayer(vectorLayer);
```

6. Then, create an instance of our new `OpenLayers.Control.SelectFeaturePath` control, add it to the map and activate it:

```
    var sp = new OpenLayers.Control.
SelectFeaturePath(vectorLayer);
    map.addControl(sp);
    sp.activate();
</script>
```

7. Now, we are going to see the new `SelectFeaturePath` control's implementation. Create a `selectFeaturePath.js` file and add some control description:

```
/**
 * Class: OpenLayers.Control.SelectFeaturePath
 * The SelectFeaturePath control selects vector features from a
given layer
 * that intersects with a path.
 *
```

```
 * Inherits from:
 *  - <OpenLayers.Control.SelectFeature>
 */
```

8. Create the new class:

```
OpenLayers.Control.SelectFeaturePath = OpenLayers.
Class(OpenLayers.Control.SelectFeature, {
```

9. Implement the `initialize` method responsible for initializing the control. Note how we call the super class's `initialize` method also, to initialize the parent class:

```
    /**
     * Constructor: OpenLayers.Control.SelectFeaturePath
     * Create a new control for selecting features using
     * an OpenLayers.Handler.Path handler.
     *
     * Parameters:
     * layers - {<OpenLayers.Layer.Vector>}, or an array of vector
layers. The
     *     layer(s) this control will select features from.
     * options - {Object}
     */
    initialize: function(layers, options) {
        OpenLayers.Control.SelectFeature.prototype.initialize.
apply(this, arguments);

        this.box = true;
        this.handlers.box = new OpenLayers.Handler.Path(this, {
            done: this.selectPath
        });
    },
```

10. Implement the `selectPath` method. This method selects those features which intersect with the created line:

```
    /**
     * Method: selectPath
     * Callback from the handlers.box set up when <path> selection
is done.
     * Select those features that intersect with the path.
     *
     * Parameters:
     * path - {<OpenLayers.Geometry.LineString>}
     */
    selectPath: function(path) {
        // If multiple is false, first deselect currently selected
features
```

```
            if (!this.multipleSelect()) {
                this.unselectAll();
            }

            // Consider we want multiple selection
            var prevMultiple = this.multiple;
            this.multiple = true;
            var layers = this.layers || [this.layer];
            var layer;
            for(var l=0; l<layers.length; ++l) {
                layer = layers[l];
                for(var i=0, len = layer.features.length; i<len; ++i)
{
                    var feature = layer.features[i];
                    // Check if the feature is displayed
                    if (!feature.getVisibility()) {
                        continue;
                    }

                    if (this.geometryTypes == null || OpenLayers.Util.
indexOf(
                        this.geometryTypes, feature.geometry.CLASS_
NAME) > -1) {
                        if (path.intersects(feature.geometry)) {
                            if (OpenLayers.Util.indexOf(layer.
selectedFeatures, feature) == -1) {
                                this.select(feature);
                            }
                        }
                    }
                }
            }
            this.multiple = prevMultiple;
        },
```

11. Finally, create a unique identifier for the class:

```
        CLASS_NAME: "OpenLayers.Control.SelectFeaturePath"
});
```

How it works...

The main program does not have much mystery. We have created a map, added a vector layer, and then added our custom `SelectFeaturePath` control:

```
var sp = new OpenLayers.Control.SelectFeaturePath(vectorLayer);
map.addControl(sp);
sp.activate();
```

Note how we have transformed the coordinates to center the map's viewport. This is because we are specifying the location in EPSG:4326 and the map is using a base layer with EPSG:900913:

```
var center = new OpenLayers.LonLat(25,50);
center.transform(new OpenLayers.Projection("EPSG:4326"),
  new OpenLayers.Projection("EPSG:900913"));
map.setCenter(center, 4);
```

Because our control is an extension of the `OpenLayers.Control.SelectFeature` control, it is important to describe it briefly before going into the code's description.

As we commented at the beginning of the recipe, this control works on the client side, that is, no request is made to the data source. The control iterates over all features in a layer (or layers) and selects those features that match the criteria.

By default, without specifying any options on its instantiation, the control allows us to select features by clicking on it. We can also set properties, such as the `hover` property, which allows us to highlight the feature pointed by the mouse or, the important one here, the `box` property, which allows the feature selection to draw a box.

This is done because the control internally uses an instance of an `OpenLayers.Handler.Box` handler, which is responsible to draw a box and returns a bounding box's coordinates so the control can check which features are inside the box.

The idea to extend our control and allow selecting features just by drawing a path is simple, instead of using an `OpenLayers.Handler.Box` handler we are going to use an `OpenLayers.Handler.Path` handler.

All right, we have background knowledge about what we need. Let's go to see how we have done.

We have created the new `OpenLayers.Control.SelectFeaturePath` class using the `OpenLayers.Class` method. This class allows merging the properties and methods of two objects:

```
OpenLayers.Control.SelectFeaturePath = OpenLayers.Class(OpenLayers.
Control.SelectFeature, {
    …
    ...
});
```

In the previous line, we are merging the properties and methods of the `OpenLayers.Control.SelectFeature` class with those defined in the second argument, which is an object in the literal notation.

Within the `initialize()` method, it is important we set the `box` and `handlers.box` properties inherited from the superclass. Setting the `handlers.box` to a new instance of `OpenLayers.Handler.Path` makes the control use a path handler instead of a box handler to select the features.

The `box` property set to `true` indicates we want to select the features using a handler instead of simply clicking on them:

```
    initialize: function(layers, options) {
        OpenLayers.Control.SelectFeature.prototype.initialize.
apply(this, arguments);

        this.box = true;
        this.handlers.box = new OpenLayers.Handler.Path(this, {
            done: this.selectPath
        });
    },
```

Using a property called `handlers.box` to specify a path handler is not the clearest way to do it, but in contrast, is the easiest one. The `OpenLayers.Control.SelectFeature` class uses this property to get the appropriate handler to make the selection.

In the previous code, when the path handler is initialized, we have set a listener function for the `done` event, which is triggered when the path handler finishes drawing the line.

The `selectPath` method, is responsible to find out which features of the layer intersect the path and change its renderer style to highlight them.

Note, the listener function receives an `OpenLayers.Geometry.LineString` instance with the created path:

```
        selectPath: function(path) {
            // If multiple is false, first deselect currently selected
    features
            if (!this.multipleSelect()) {
                this.unselectAll();
            }
```

Because the control can work with more than one layer, we need to iterate over all the layers and all the features on each layer:

```
            // Consider we want multiple selection
            var prevMultiple = this.multiple;
            this.multiple = true;
            var layers = this.layers || [this.layer];
            var layer;
            for(var l=0; l<layers.length; ++l) {
                layer = layers[l];
                for(var i=0, len = layer.features.length; i<len; ++i) {
                    var feature = layer.features[i];
                    // Check if the feature is displayed
                    if (!feature.getVisibility()) {
                        continue;
                    }
```

After checking if the feature is visible we can check whether the feature intersects with the path, using the `intersects` method:

```
                    if (this.geometryTypes == null || OpenLayers.Util.
    indexOf(
                        this.geometryTypes, feature.geometry.CLASS_NAME) >
    -1) {
                        if (path.intersects(feature.geometry)) {
                            if (OpenLayers.Util.indexOf(layer.
    selectedFeatures, feature) == -1) {
                                this.select(feature);
                            }
                        }
                    }
                }
            }
            this.multiple = prevMultiple;
        },
```

The `OpenLayers.Util.indexOf(array, object)` function returns the index at which an object is found within an array.

To change the rendering style of the feature, we simply call the method `select` inherited from the `OpenLayers.Control.SelectFeature` class.

A vector layer can render the features using different styles, called render intents: default, select, or temporary are the default render intents we can use.

See also

- The *Creating a custom renderer* recipe
- The *Playing with StyleMap and the render intents* recipe in *Chapter 7*, *Styling Features*

Making an animation with image layers

When working with geographic information, its geometrical representation within the space is not the only important thing. Day by day time is becoming a new and important dimension to take into account.

This way, visualizations must show how data changes over time: city population, country frontiers, roads built, and so on.

There are many solutions to animate the data evolution through time but, as always, we work with web technologies, there are two groups: the solutions based on the server side and those based on the client side.

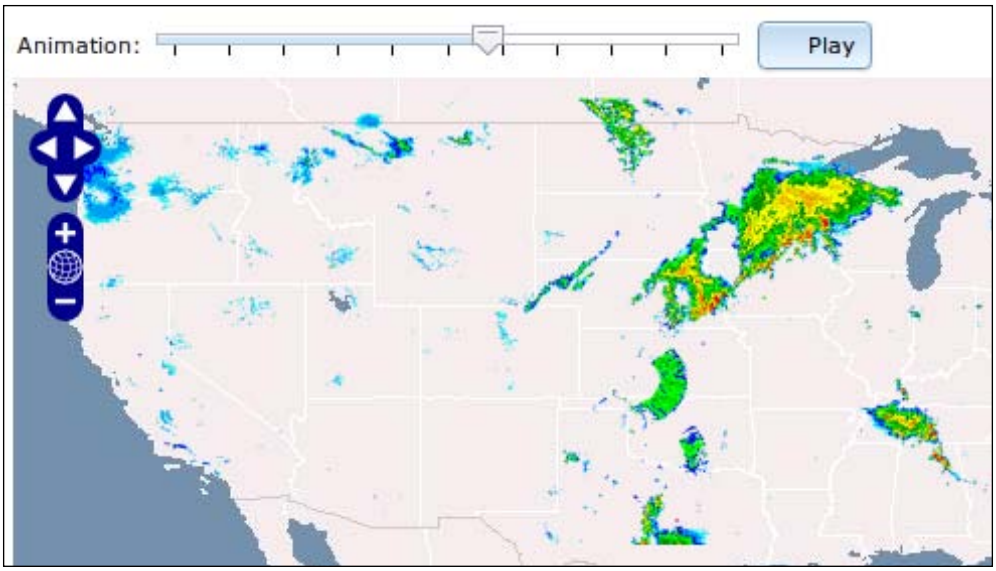
For server-side solutions, we can find the `TIME` parameter in the WMS and WFS standards. It allows us to request for raster or vector data in a specific time or within a range.

Server solutions means the client must request the server every time we want to show the data for a different interval.

For the client side, a simple solution is to have in the memory all the data, and only show those that correspond to the interval we are interested in.

In this recipe we are going to show how easily we can create an animation on the client side.

We are going to load some images from **NEXTRAD** (`http://en.wikipedia.org/wiki/NEXRAD`), showing the rain evolution at different time instants (as shown in the following screenshot), and we will create an animation by simply showing or hiding images:



How to do it...

1. Create a new HTML file and add the OpenLayers dependencies. In the `body` section start adding the elements necessary for the play button and the slider:

```
<table>
    <tr>
        <td>
            Animation:
        </td>
        <td>
            <div id="animSlider" dojoType="dijit.form.
HorizontalSlider" value="0" minimum="0" maximum="100"
intermediateChanges="true"
                showButtons="false" style="width:300px;"
onChange="animation">
               <div dojoType="dijit.form.HorizontalRule"
container="bottomDecoration" count=11 style="height:5px;"></div>
            </div>
        </td>
        <td>
            <div dojoType="dijit.form.ToggleButton"
iconClass="dijitCheckBoxIcon" onChange="animateAction">Play</div>
        </td>
    </tr>
</table>
```

In addition to OpenLayers, we are using the Dojo Toolkit framework (`http://dojotoolkit.org`) to create more attractive user interfaces. Learning Dojo is out of the scope of this book and also, it is not necessary to know it to understand this recipe.

2. Next, add the `div` element to hold the map:

```
<div id="ch08_animating_raster" style="width: 100%; height:
100%;"></div>
```

3. Now, in the header section add the next JavaScript code. Start initializing the map instance, add a base layer, and center the viewport:

```
<script type="text/javascript">
    var map = new OpenLayers.Map("ch08_animating_raster");
    var wms = new OpenLayers.Layer.WMS("OpenLayers WMS Basic",
"http://labs.metacarta.com/wms/vmap0",
    {
        layers: 'basic'
    });
    map.addLayer(wms);

    // Center the view
    map.setCenter(new OpenLayers.LonLat(-85, 40), 4);
```

4. Now, we are going to create some raster image layers, add them to the map, and also store them on an array to control their visibility:

```
    var img_extent = new OpenLayers.Bounds(-131.0888671875,
30.5419921875, -78.3544921875, 53.7451171875);
    var img_size = new OpenLayers.Size(780, 480);

    var img_ulr = image = null;
    var imgArray = [];
    for(var i=1; i<=32; i++) {
        index = (i<10) ? "0"+i : i;
        img_url = "http://localhost:8080/openlayers-cookbook/
recipes/data/radar/nexrad"+index+".png";
        image = new OpenLayers.Layer.Image("Image Layer", img_url,
img_extent, img_size, {
            isBaseLayer: false,
            alwaysInRange: true, // Necessary to always draw the
image
            visibility: false
        });
        imgArray.push(image);
```

```
        map.addLayer(image);
    }
    imgArray[0].setVisibility(true);
```

5. The following code controls the changes in the slider widget:

```
    var currentIndex = 0;
    function animation(value){
        imgArray[currentIndex].setVisibility(false);
        currentIndex = Math.floor(value * 31 / 100);
        imgArray[currentIndex].setVisibility(true);
    }
```

6. Finally, the following code controls the automatic animation when the **Play** button is clicked:

```
    var interval = null;
    function animateAction(checked) {
        if(checked) {
            interval = setInterval(function() {
                var v = dijit.byId('animSlider').get('value');
                v = (v>=100) ? 0 : (v+1);
                dijit.byId('animSlider').set('value', v);
                animation(v);
            },50);
        } else {
            clearInterval(interval);
        }
    }
</script>
```

How it works...

As mentioned at the beginning of this recipe, the idea is to animate some weather radar on the client side. For this reason, we load a set of images from the server, creating a layer for each one.

The `OpenLayers.Layer.Image` class is used here to hold each image as a single layer. Its constructor requires five parameters:

- `name`: The name of the layer
- `url`: The URL where the image must be taken
- `extent`: The bounds of the image within the map
- `size`: The size in pixels
- `options`: A set of options to be passed to the layer

The `extent` and the `size` arguments are used to compute the resolution of the image.

In our case, all the image layers will have the same `extent` and `size`:

```
    var img_extent = new OpenLayers.Bounds(-131.0888671875,
30.5419921875, -78.3544921875, 53.7451171875);
    var img_size = new OpenLayers.Size(780, 480);
```

Later, within the loop to create all the image layers, we are setting dynamically the value of the `img_url` variable and passing some options to the `OpenLayers.Layer.Image` constructor:

```
        image = new OpenLayers.Layer.Image("Image Layer", img_url,
img_extent, img_size, {
            isBaseLayer: false,
            alwaysInRange: true, // Necessary to always draw the image
            visibility: false
        });
        imgArray.push(image);
```

By setting the `isBaseLayer` property to `false` we are specifying that our layer is not a base layer, it will act as an overlay. In addition, we set the `visibility` property to `false` to initially hide the layer. Later, we will set the first image layer as the visible one:

```
    imgArray[0].setVisibility(true);
```

The `alwaysInRange` property is inherited from the superclass `OpenLayers.Layer` and specially useful in this case. We want our image layer to be visible at any zoom level. We do not want OpenLayers to compute the right resolution; given the image layer extent and size, the layer must be shown. So, setting `alwaysInRange` to `true` makes the layer always visible.

Next, we are going to see how to automatically automate the animation. The `animateAction` is executed when the play button is pressed. It is a toggle button, so depending on its state, the boolean `checked` parameter will be true if checked or false if not:

```
    function animateAction(checked) {
        if(checked) {
```

If the play button is checked, then we create an interval to execute the given anonymous function every 50 milliseconds which, in fact, increases the value of the slider and calls the `animation` function:

```
            interval = setInterval(function() {
                var v = dijit.byId('animSlider').get('value');
                v = (v>=100) ? 0 : (v+1);
                dijit.byId('animSlider').set('value', v);
                animation(v);
            },50);
        } else {
```

If the button is unchecked, then we remove the interval reference to stop the execution:

```
        clearInterval(interval);
    }
```

Intervals and **timeouts** are used in JavaScript to create animations and delays.

A good explanation of intervals in JavaScript can be found at `http://www.w3schools.com/jsref/met_win_setinterval.asp`.

Finally, let's take a look at the `animation` function, which is responsible to change the layer visibilities and create the animation effect.

All the layers are added to the map and also stored in the `imgArray`. The global `currentIndex` variable is used to hold the current visible layer. The animation function does three things:

- Hides the current visible layer
- Given the numeric `value`, that varies from 0 to 100, computes the layer array index, that goes from 0 to 31
- Shows the new current layer

```
var currentIndex = 0;
function animation(value){
    imgArray[currentIndex].setVisibility(false);
    currentIndex = Math.floor(value * 31 / 100);
    imgArray[currentIndex].setVisibility(true);
}
```

That's all! Once the set of layers is loaded in the client side, using any modern browser, the performance of the animation is good enough.

This is only a sample, so there are tons of things to improve. For example, think on how to implement a situation where we have a remote server with hundreds of images to load sequentially. In this case we cannot load all images at once because that can cause an out-of-memory problem in the browser.

Supposing we have thousands of images to animate, we could implement some buffer strategy. Given a buffer of ten images, we can load the first ten images from the server, then animate them and when the animation arrives to the last loaded image, load the next ten images from the server.

As we can see, these situations are out of the scope of this book and not only related to OpenLayers but with software architecture and design.

See also

- The *Creating an Image layer* recipe in *Chapter 2*, *Adding Raster Layers*
- The *Changing layer opacity* recipe in *Chapter 2*, *Adding Raster Layers*

Index

Symbols

A

B

C

D

E

F

G

H

N

O

P

Q

R

S

T

U

V

W

X

Z

Thank you for buying
OpenLayers Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

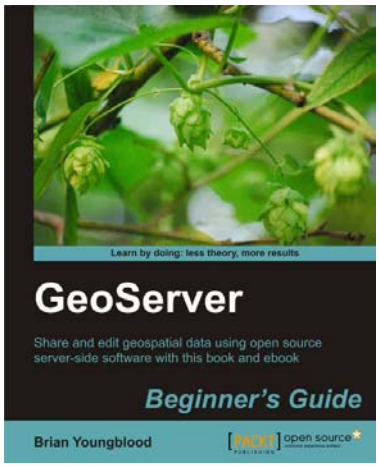
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



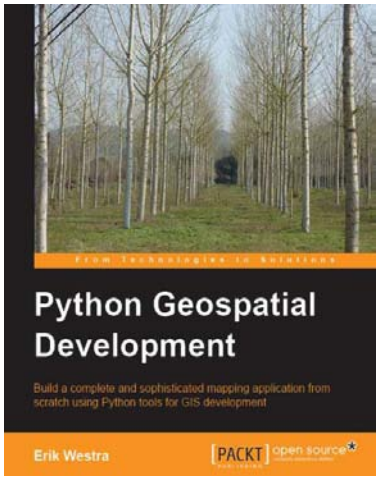
GeoServer Beginner's Guide

ISBN: 978-1-849516-68-6 Paperback: 350 pages

Share and edit geospatial data using open source server-side software with this book and ebook

1. Learn free and open source geospatial mapping without prior GIS experience
2. Share real-time maps quickly
3. Learn step-by-step with ample amounts of illustrations and usable code/list

---



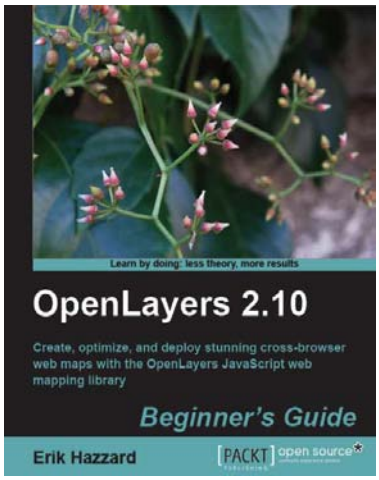
Python Geospatial Development

ISBN: 978-1-849511-54-4 Paperback: 508 pages

Build a complete and sophisticated mapping application from scratch using Python tools for GIS development

1. Build applications for GIS development using Python
2. Analyze and visualize Geo-Spatial data
3. Comprehensive coverage of key GIS concepts
4. Recommended best practices for storing spatial data in a database
5. Draw maps, place data points onto a map, and interact with maps

[PACKT] open source
community experience distilled
PUBLISHING

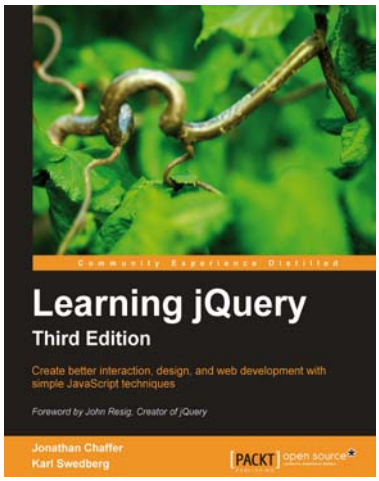


OpenLayers 2.10 Beginner's Guide

ISBN: 978-1-84951-412-5 Paperback: 372 pages

Create, optimize, and deploy stunning cross-browser web maps with the OpenLayers JavaScript web mapping library

1. Learn how to use OpenLayers through explanation and example
2. Create dynamic web map mashups using Google Maps and other third-party APIs
3. Customize your map's functionality and appearance
4. Deploy your maps and improve page loading times



Learning jQuery, Third Edition

ISBN: 978-1-849516-54-9 Paperback: 428 pages

Create better interaction, design, and web development with simple JavaScript techniques

1. An introduction to jQuery that requires minimal programming experience
2. Detailed solutions to specific client-side problems
3. Revised and updated version of this popular jQuery book