Part I
Prototype

Prototype was one of the first JavaScript libraries to gain prominence during the Web 2.0 resurgence. When the term AJAX was first coined in 2005, making cross-browser XMLHttpRequests was a minefield of browser-specific code. Prototype assists you in your quest for cross-browser compatibility by smoothing out the rough edges of event handling by providing a common method for binding events to their respective handlers and providing a common interface for creating AJAX requests that work in all browsers. It also gives you a cross-browser way to manipulate the DOM, by handling the special cases in all browsers, and allowing you to focus on just writing code without cluttering up your code with browser-specific "if-else" statements.

Prototype extends the JavaScript language as well as the elements. The native JavaScript Object is extended to include methods for determining the type of data the object represents as well as helpful serialization methods. The Enumerable class allows you to easily traverse and manipulate your arrays of both JavaScript objects and DOM elements by providing useful methods such as `each()` and `map()` directly on your arrays. The native Function object is also extended with useful methods, such as `wrap()`, which let you write interceptors for your methods that provide useful features like logging.

Prototype eases inheritance with the Class object. You can easily extend your objects and create hierarchies without the headaches associated with normal inheritance in statically typed languages. All of these features make Prototype the best choice for writing logic in JavaScript, and it provides you with an excellent base for writing your own JavaScript library. Since Prototype does all of the heavy lifting for you, you can focus on the fun parts of library development — creating new widgets and data structures.

1

Extending and Enhancing DOM Elements

Prototype is an excellent framework to use either as your main JavaScript library or as the foundation of another library. Part of the magic of Prototype is the extension of DOM elements by the framework. By adding new methods to elements, Prototype makes it easier to write cross-browser code in a more eloquent manner. There are also several methods for taking care of the dirty details involved in positioning elements. It is easier to write unobtrusive JavaScript by taking advantage of helper methods such as `getElementsByClassName` and `getElementsBySelectors`, making it easy to apply styling or events to groups of elements with something in common.

In this chapter, you'll learn about:

- ❑ Extending a DOM element with Prototype
- ❑ Altering and manipulating content and size
- ❑ Using CSS to style an element

Extending a DOM element

Before Prototype came along, cross-browser code often looked a lot like a road map: a lot of branches and a lot of the same checks over and over again. By extending the elements you are working on, Prototype is able to centralize all of the cross-browser hacks that make JavaScript programming such a chore. Prototype keeps its extension methods for all elements in the `Element.Methods` and `Element.Methods.Simulated` object. If the element is an `input`, `select`, or `textarea` tag, the methods in `Form.Element.Methods` are also included. Form elements themselves are extended with the methods in `Form.Methods`. Most of these methods return the original element, so you can chain together methods like so: `$(myElement) .update("updated").show();`. It is important to note that not only is the element you choose extended, but all of the child elements of that element are also extended.

In browsers that support modification of the `HTMLElement.prototype`, Prototype adds the methods to `HTMLElement` for you. That means you don't have to call `Element.extends()` on any element you create by hand. You can start using Prototype methods immediately.

```
var newDiv = document.createElement("div");
newDiv.update("Insert some text");
newDiv.addClassName("highlight");
```

Internet Explorer doesn't support modifying `HTMLElement`, so you have to call `Element.extends()` or get a reference to the element using the `$()` or `$$()` methods.

$() — "The dollar function"

The easiest way to extend a DOM element is to use the `$()` function to get a reference to the element rather than using `document.getElementById` or some other method. When you obtain a reference this way, Prototype automatically adds all of the methods in `Element.Methods` to the element. If you pass a string to the method, it will get a reference to the element with the ID you specify for you. If you pass in a reference to the element, it will return the same reference but with the extension methods. This is the most common way to extend an element.

```
<body>
<div id="myId">Hello Prototype</div>
<script type="text/javascript">
      $("myId").hide();
</script>
</body>
```

$$()

This works in a similar manner to the `$()` function. It takes a CSS selector as an argument and returns an array of elements that match the selector. CSS selectors are a powerful tool for getting a specific element back from the DOM. The elements in the array will be in the same order as they appear in the DOM and each will be extended by Prototype.

```
$$('input');
// select all of the input elements

$$('#myId');
//select the element with the id "myId"

$$('input.validate');
//select all of the input elements with the class "validate"
```

Prototype does not use the browser's built-in CSS selector parsing, so it is free to implement selectors specified in newer versions of CSS than the browser supports. As a result, version 1.5.1 and higher of Prototype includes support for almost all of CSS3.

```
$$('#myId > input');
//select all of the input elements that are children of the element with the id "myId"

$$('table < tr:nth-child(even)');
//selects all of the even numbered rows of all table elements.
```

Element.extend()

This method accepts an element and extends the element using the methods found in `Element.Methods`. It is very similar to `$()` except it only accepts references to DOM objects and will not fetch a reference for you if you pass it an id.

Here is a simple example of using `Element.extend()`:

```
Var newDiv = document.createElement("div");
Element.extend(newDiv);
newDiv.hide();
```

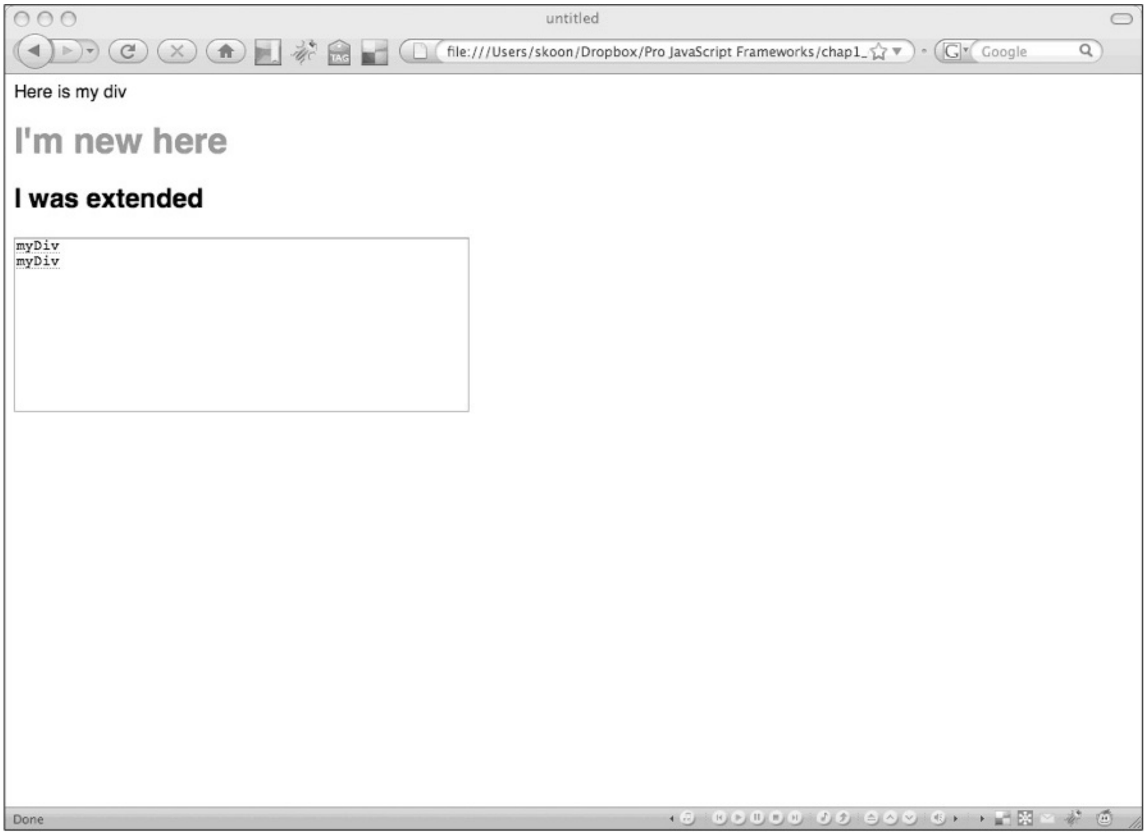
Element as a Constructor

You can also use the `Element` object as a way to construct new DOM elements rather than using the built-in DOM methods. New elements created in this way are automatically extended by Prototype and can be used immediately.

```
<head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
      <title>untitled</title>
      <style>
          .redText { color: red;}
      </style>
</head>
<body>
    <div id="myDiv" class="main">Here is my div</div>

    <textarea id="results" cols="50" rows="10"></textarea>
    <script type="text/javascript" src="prototype-1.6.0.2.js"></script>
<script type="text/javascript">
        Event.observe(window,"load", function(e) {
            $("results").value = "";
            $("results").value += $("myDiv").id + "\n";
            $("results").value += $$(".main")[0].id + "\n";
            var newEl = new Element("h1",{"class": "redText"});
            $("myDiv").insert(newEl);
            newEl.update("I'm new here");

            var manuallyCreated = document.createElement("h2");
            $("myDiv").insert(manuallyCreated);
            Element.extend(manuallyCreated);
            manuallyCreated.update("I was extended");
        });
    </script></body>
```

Since `$$()` returns a DOM-ordered array of elements, you have to refer to the first element in the array by the ordinal 0.

Here you see some of the ways you can extend an element. First, you use the `$()` method to grab the element by ID and extend the element. Next, you use the `$$()` method and pass in a CSS selector to get the element by class name. Now you will use the `Element` object as a constructor and create a new H1 element with a class of `redText`, inserting it into the myDiv element and setting the text of the newly created element. Finally, you create an element the old-fashioned way and use `Element.extend()` to extend the element, as shown in Figure 1-1.


untitled
file:///Users/skoon/Dropbox/Pro JavaScript Frameworks/chap1_
Google
Done

Here is my div

I'm new here

I was extended

```
myDiv
myDiv
```

Figure 1-1

Navigating the DOM

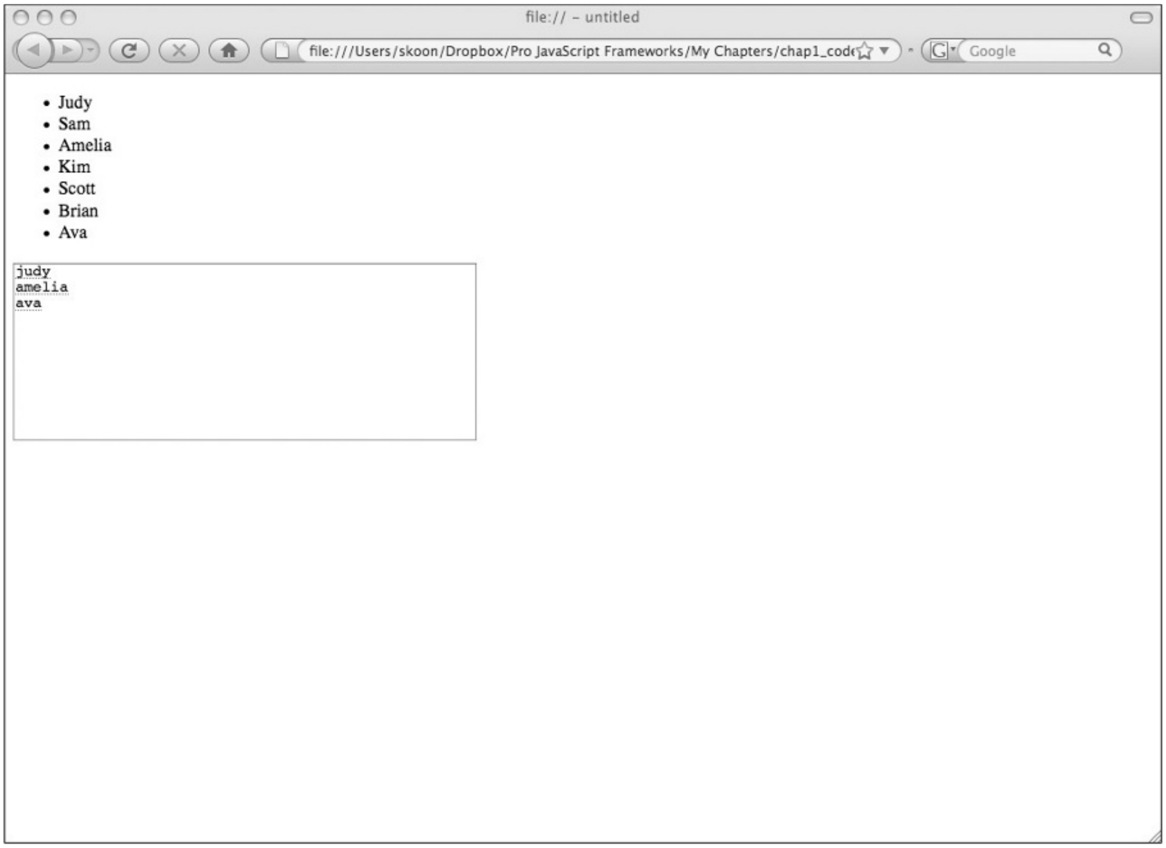
Trying to figure out where the element you are interested in is located in the DOM and what elements are surrounding that element is no easy task. Prototype's `Element` object provides a multitude of ways to traverse the DOM. Several methods allow you to specify CSS rules to narrow your search. All of Prototype's DOM navigation methods ignore white space and only return element nodes.

adjacent

This method finds all of an element's siblings that match the selector you specify. This method is useful for dealing with lists or table columns.

```
<body>
    <ul id="PeopleList">
        <li class="female" id="judy">Judy</li>
        <li class="male" id="sam">Sam</li>
        <li class="female" id="amelia">Amelia</li>
        <li class="female" id="kim">Kim</li>
        <li class="male" id="scott">Scott</li>
        <li class="male" id="brian">Brian</li>
        <li class="female" id="ava">Ava</li>
    </ul>
    <textarea id="results" cols="50" rows="10"></textarea>
    <script type="text/javascript" src="prototype-1.6.0.2.js"></script>
    <script type="text/javascript">
        Event.observe(window,"load", function(e) {
            var els =$("kim").adjacent("li.female");
            $("results").value = "";
            for(var i = 0;i < els.length; i++) {
                $("results").value += els[i].id + "\n";
            }
        });
    </script>
</body>
```

Here you start with the list element with the ID "kim" and gather the `li` elements adjacent with the class name `female`, as shown in Figure 1-2.


file:// – untitled
file:///Users/skoon/Dropbox/Pro JavaScript Frameworks/My Chapters/chap1_code
Google

- Judy
- Sam
- Amelia
- Kim
- Scott
- Brian
- Ava

```
judy
amelia
ava
```

Figure 1-2

ancestors

This collects all of the element's ancestors in the order of their ancestry. The last ancestor of any given element will always be the HTML element. Calling this method on the HTML element will just return an empty array. Given the following HTML snippet:

```
<html>
      <body>
            <div id="myDiv">
                  <p id="myParagraph">Hello pops</p>
            </div>
      </body>
</html>
```

The array would be returned with the elements in the following order:

```
DIV @--> BODY @--> HTML
```

You can use the following code to verify this behavior:

```
<html>
      <body>
            <div id="myDiv">
                  <p id="myParagraph">Hello pops</p>
            </div>
            <textarea id="results" cols="50" rows="10"></textarea>
            <script type="text/javascript" src="prototype-1.6.0.2.js"></script>
            <script type="text/javascript">
                Event.observe(window,"load", function(e) {
                    var a = $('myParagraph').ancestors();
                    $('results').value = "";
                    for(var i = 0;i < a.length;i++) {
                        $('results').value += a[i].tagName + "\n";
                    }
                });
            </script>

      </body>
</html>
```

up/down/next/previous

These four methods comprise Prototype's core DOM traversal functionality. They allow you to define a starting element, and then walk around the DOM at your leisure. All of the methods are chainable, allowing you to call each in succession on whatever element was returned by the preceding function. If no element can be found that matches the criteria you define, `undefined` is returned. Each method accepts two arguments: a CSS selector or a numeric index. If no argument is passed, the first element matching the criteria is returned. If an index is passed, the element at that position in the element's corresponding array is returned. For example, the resulting array used for the `down()` method will match the element's descendants array. If a CSS selector is passed in, the first element that matches that rule is returned. If both an index and a CSS rule are passed in, the CSS rule is processed first and then the index is used to select the element from the array defined by the CSS rule.

up

Returns the first ancestor matching the specified index and/or CSS rule. If no ancestor matches the criteria, `undefined` is returned. If no argument is specified, the element's first ancestor is returned. This is the same as calling `element.parentNode` and passing the parent through `Element.extend`.

down

Returns the first descendant matching the specified index and/or CSS rule. If no descendant matches the criteria, `undefined` is returned. If no argument is specified, the element's first descendant is returned.

next

Returns the element's siblings that come after the element matching the specified index and/or CSS rule. If no siblings match the CSS rule, all the following siblings are considered. If no siblings are found after the element, `undefined` is returned.

previous

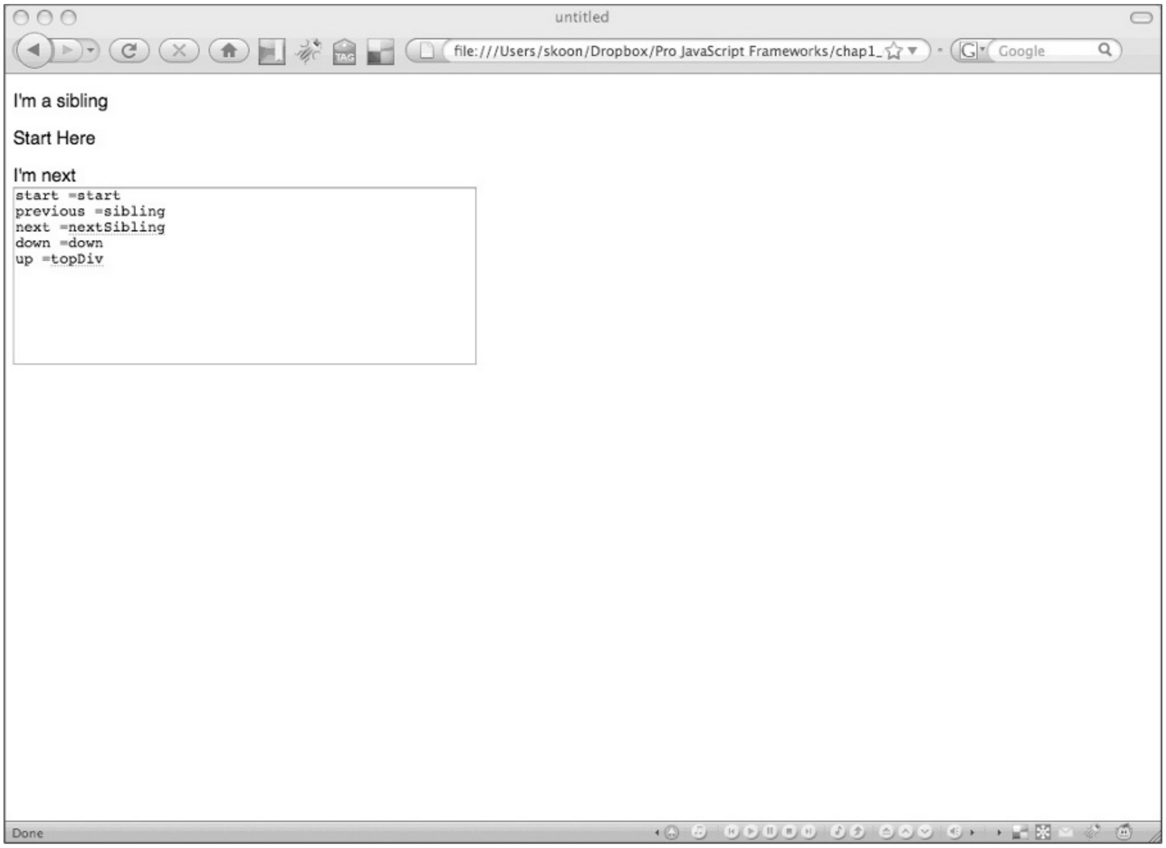
Returns the element's siblings that come before the element matching the specified index and/or CSS rule. If no siblings match the CSS rule, all the previous siblings are considered. If no siblings are found before the element, `undefined` is returned.

Take a fragment of HTML that looks like the following example. Here you are defining four elements that relate to each other like this:

```
<div id="up">
        <p id="prevSibling">I'm a sibling</p><div id="start"><p id="down">Start
Here</p></div> <span id="nextSibling">I'm next</span>
    </div>
```

This code starts at the start DIV and looks at the previous, next, up, and down elements. You start at the element with the ID start. The paragraph element containing the text "Start Here" is the first child of the starting element and is returned by calling the `down` method. The `up` method returns the `topDiv` div. The `previous` method returns the `sibling` paragraph element and `next` returns the `nextSibling` `span`, as shown in Figure 1-3.

```
<body>
    <div id="up">
        <p id="prevSibling">I'm a sibling</p><div id="start"><p id="down">Start
Here</p></div> <span id="nextSibling">I'm next</span>
    </div>
    <textarea id="results" cols="50" rows="10"></textarea>
    <script type="text/javascript" src="prototype-1.6.0.2.js"></script>
    <script type="text/javascript">
        Event.observe(window,"load", function(e) {
            var startEl = $('start');
            var previousEl = startEl.previous();
            var upEl = startEl.up();
            var downEl = startEl.down();
            var nextEl = startEl.next();

            var resultTextArea = $("results");
            resultTextArea.value = "";
            resultTextArea.value += "start =" + startEl.id + "\n";
            resultTextArea.value += "previous =" + previousEl.id + "\n";
            resultTextArea.value += "next =" + nextEl.id + "\n";
            resultTextArea.value += "down =" + downEl.id + "\n";
            resultTextArea.value += "up =" + upEl.id + "\n";
        });
    </script>
</body>
```



untitled
file:///Users/skoon/Dropbox/Pro JavaScript Frameworks/chap1_
Google
Done

I'm a sibling

Start Here

I'm next

```
start =start
previous =sibling
next =nextSibling
down =down
up =topDiv
```

Figure 1-3

descendants/descendantOf/firstDescendant/ immediateDescendants

All of these methods allow you to work with the children of a given element. The methods `descendants` and `immediateDescendants` return arrays of child elements.

- **descendants** — This method returns an array containing the children of the element. If the element has no children, an empty array is returned.
- **descendantOf** — This method returns a Boolean telling you whether or not the given element is a descendant of the given ancestor.
- **firstDescendant** — This method returns the first child of a given element that is itself an element.
- **immediateDescendants** — (deprecated) This method returns an array of the elements one level down and no further.

getElementsBySelector/getElementsByClassName

These methods allow you to select groups of elements based on their attributes or position and manipulate the elements however you choose. Both of these methods have been deprecated and you should use the `$$()` method in place of them.

childElements

This useful function gathers up all the children of an element and returns them as an array of extended elements. The elements are returned in the same order as they are in the DOM. So, the element at index 0 is the closest child to the parent element and so forth.

Altering Page Content

Prototype provides four methods for changing content on a page: insert, remove, replace, and update. These methods can be called using the `Element` object and are added to any element that is extended. They all take an optional argument, which is the element to be altered. The insert and replace methods call `eval()` on any script tags contained in the content passed to them. Any of these methods that take a content argument will accept plain text, an HTML fragment, or a JavaScript object that supports `toString()`.

insert(element, content), insert(element, {position:content)

Insert takes the content you provide and inserts it into an element. If you do not specify a position (such as top, bottom, before, or after), your content will be appended to the element. This method is useful for dynamically inserting content retrieved from a web service or for loading elements into a page one piece at a time for performance reasons.

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

        function insertSample() {
            $("MainDiv").insert("New Content added at the end by default");
            $("MainDiv").insert({top:"Added at the top"});
            $("MainDiv").insert({before:"Added before the element"})
            $("MainDiv").insert({after:"Added after the element"});
            $("MainDiv").insert({bottom:"Added at the bottom"});
        };
        insertSample();
    </script>
```

remove

Calling remove on an extended element removes it completely from the DOM. The function returns the removed element. This method is most often used to remove an element after a user has chosen to delete whatever item the element represents in the UI.

```
<body>
    <table id="myTable">
        <tr id="firstRow"><td>First Row</td></tr>
        <tr id="secondRow"><td>Second Row</td></tr>
        <tr id="thirdRow"><td>Third Row</td></tr>
    </table>
    <script type="text/javascript" src="prototype-1.6.0.2.js"></script>
    <script type="text/javascript">

        function removeRow() {
            $("secondRow").remove();
        };
        removeRow();
    </script>
</body>
```

replace

Replace takes away the element specified and replaces it with the content provided. This removes the element and its children from the DOM.

```
<body>
    <div id="MainDiv">
        <div id="tempDiv">Place holder</div>
    </div>
    <script type="text/javascript" src="prototype-1.6.0.2.js"></script>
    <script type="text/javascript">
        //simulate loading content from a web service
        setTimeout(function () {
            $("MainDiv").replace("<h1>Replaced Content</h2>");
        }, 1000);
    </script>
</body>
```

update

Update replaces the content of an element with the specified content. It does not remove the element from the DOM, although it does remove any children of the element.

```
<body>
    <div id="MainDiv">Here is some content to be updated</div>
    <script type="text/javascript" src="prototype-1.6.0.2.js"></script>
    <script type="text/javascript">

    //simulate loading content from a web service
    setTimeout(function() {
        $("MainDiv").update("updated the content");
    }, 1000);
    </script>
</body>
```

Manipulating Element Size, Position, and Visibility

One of the hardest things about working with the DOM in different browsers is getting the dimensions of the elements contained in the DOM. Each browser has quirks relating to how it sizes elements in the DOM and how its size affects the flow of the surrounding elements.

Positioning an Element

Setting an element's position is one of the cornerstones of modern web page design. Often when designing dynamic web pages, you need to be able to move elements around and place them on the page exactly where you want them. To place an element precisely on the page, you should first set its position CSS style. Setting the position style rule to absolute means that the element's top and left coordinates are calculated from the top-left corner of the document. Setting the position to relative allows you to position the element using numbers calculated to the containing block's top-left corner. Prototype provides a few methods for easily setting an element's position style.

makePositioned, undoPositioned

These methods allow you to easily make CSS-positioned blocks out of elements in your DOM. Calling `makePositioned` on an element sets its `position` to `relative` if its current position is static or `undefined`. The `undoPositioned` method sets the element's `position` back to what it was before `makePositioned` was called.

```
$("myElement").makePositioned();
$("myElement").undoPositioned();
```

absolutize, relativize

These methods change the positioning setting of the given element by setting the position style to either absolute or relative, respectively.

```
$("myElement").absolutize();
$("myElement").relativize();
```

clonePosition

This method creates a new element with the same position and dimensions as the current element. You specify what settings are applied to the new element by using an optional parameter containing the following options:

Setting	Description
setLeft	Applies the source's CSS left property. Defaults to true.
setTop	Applies the source's CSS top property. Defaults to true.
setWidth	Applies the source's CSS width property. Defaults to true.
setHeight	Applies the source's CSS height property. Defaults to true.
offsetLeft	Lets you offset the clone's left CSS property by *n* value. Defaults to 0.
offsetTop	Lets you offset the clone's top CSS property by *n* value. Defaults to 0.

Dealing with Offsets

Prototype has a couple of different methods on its `Element` object that make finding the offset of an element easier.

cumulativeOffset, positionedOffset, viewportOffset

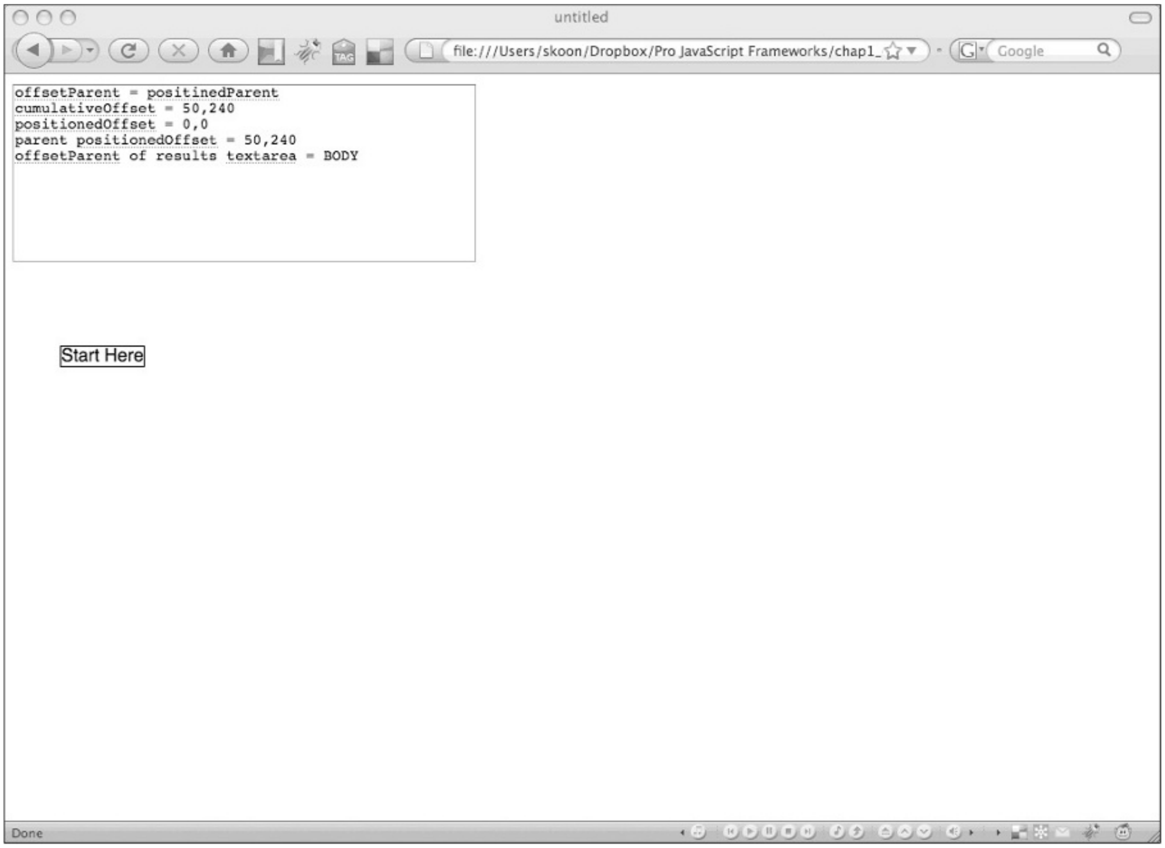
Each of these methods returns two numbers, the top and left values of the given element in the form `{ left: number, top: number}`. The `cumulativeOffset` method returns the total offset of an element from the top left of the document. The positionedOffset method returns the total offset of an element's closest positioned (one whose position is set to `'static'`) ancestor. The viewportOffset method returns the offset of the element relative to the viewport.

getOffsetParent

This method returns the nearest positioned ancestor of the element, and returns the body element if no other ancestor is found.

The following code illustrates how the different offsets are calculated. In it, you have two elements: a parent `DIV` with one child. The parent has its `position` set to `absolute` and is positioned 240 pixels from the top and 50 pixels from the left side of the document. When you call the `getOffsetParent` method of the element with the ID of `start`, the positioned element `positionedParent` is returned. The `results` textarea has no positioned ancestors. If you call `getOffsetParent` on it, the `BODY` element is returned. Since the `start` element itself is not positioned, calling `positionedOffset` returns 0,0, as shown in Figure 1-4.

```
<body>
    <div id="positionedParent" style="position:absolute;border:1px solid black;
top:240px;left:50px;">
        <div id="start">Start Here</div>
    </div>

    <textarea id="results" cols="50" rows="10"></textarea>
    <script type="text/javascript" src="prototype-1.6.0.2.js"></script>
    <script type="text/javascript">
        Event.observe(window,"load", function(e) {
            $("results").value = "";
            $("results").value += "offsetParent = " + $("start").getOffsetParent()
.id + "\n";
            $("results").value += "cumulativeOffset = " + $("start")
.cumulativeOffset() + "\n";
            $("results").value += "positionedOffset = " + $("start")
.positionedOffset() + "\n";
            $("results").value += "parent positionedOffset = " + $("start")
.parentNode.positionedOffset() + "\n";
        });
    </script>
</body>
```



```
offsetParent = positinedParent
cumulativeOffset = 50,240
positionedOffset = 0,0
parent positionedOffset = 50,240
offsetParent of results textarea = BODY
```

untitled
file:///Users/skoon/Dropbox/Pro JavaScript Frameworks/chap1_
Google
Start Here
Done

Figure 1-4

Showing/Hiding Elements

Showing and hiding an element has been part of your web developer's toolkit since you typed in your first script tag.

show/hide

These methods allow you to quickly change an element's visibility. They do this by setting the elements `display` CSS style to `none`. Setting the display to none removes the element from the flow of the document and causes the browser to render the other elements in the page as if the element were not present.

```
$("myElement").show();
$("myElement").hide();
```

setOpacity

This method sets the opacity of a given element, while relieving you of the burden of dealing with various browser inconsistencies. It takes a floating-point number, with 0 being totally transparent and 1 being completely opaque. Using this method is the equivalent of setting the opacity via a CSS class or using the `setStyle` method, passing in a value for opacity.

```
$("myElement").setOpacity(0.5);
```

Sizing an Element

Every browser has some kind of quirk associated with the way it represents elements on the screen and how it calculates the element's dimensions. Different browsers calculate an element's computed style differently. Prototype equalizes the differences and returns the correct computed style for the browser.

getDimensions, getHeight, getWidth

Using these methods, you can get the computed dimensions of an element at run time. The `getDimensions` method returns an object containing the computed height and width of the element. When you call `getDimensions`, it's best to save the returned value in a local variable and refer to that rather than making multiple calls. If you just want the width or height, it's best to just call the appropriate method.

```
Var dimensions = $('myDiv').getDimensions();
Var currentWidth = dimensions.width;
Var currentHeight = dimensions.height;
```

makeClipping, undoClipping

The CSS `clip` property allows you to define whether or not the element's content should be shown if the content is wider or taller than the element's width and height will allow. Since the `clip` property is poorly supported amongst the browsers, Prototype provides this method that will set an element's `overflow` property to `hidden` for you. You can use `undoClipping` to allow the element to resize normally.

Working with CSS and Styles

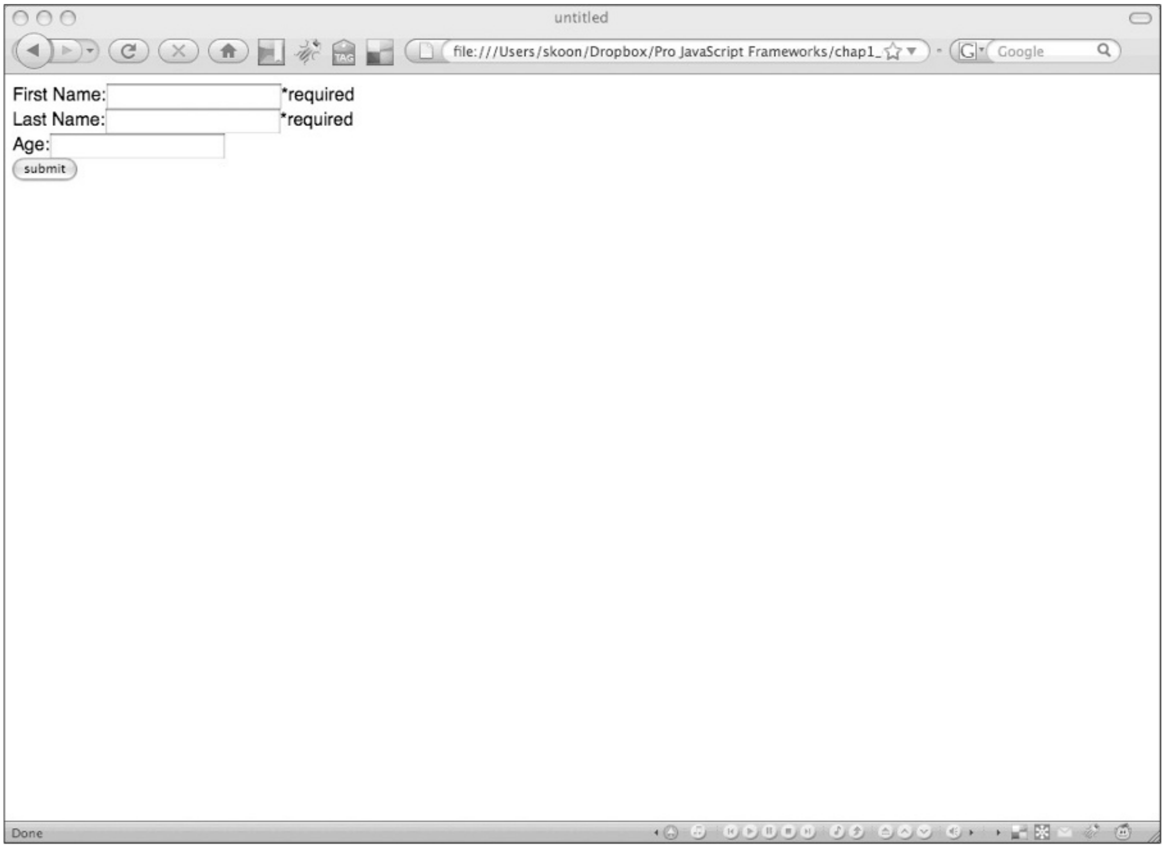
CSS classes are useful for marking elements in response to some event. Say you are creating an online survey form and you want to mark several fields as required, but you don't want to get each element that is required by ID and check them one by one to make sure the user has entered a proper value. You can create a CSS class called "required" and apply it to each of the elements you need the user to enter a value into. Sometimes you need to change an element's style or class at run time in response to a user- or data-driven event, say if you are changing a table row from read-only to editable. Classes are an invaluable tool in any web developer's toolkit. Prototype makes it easier for you to apply and remove CSS classes from elements in your DOM.

addClassName, removeClassName, toggleClassNames

These three methods all alter the className property of a given element. All of them check to make sure the element has the given class name. These methods are useful when you need to set classes on an element or need to turn a CSS style on or off. Their names are self-explanatory.

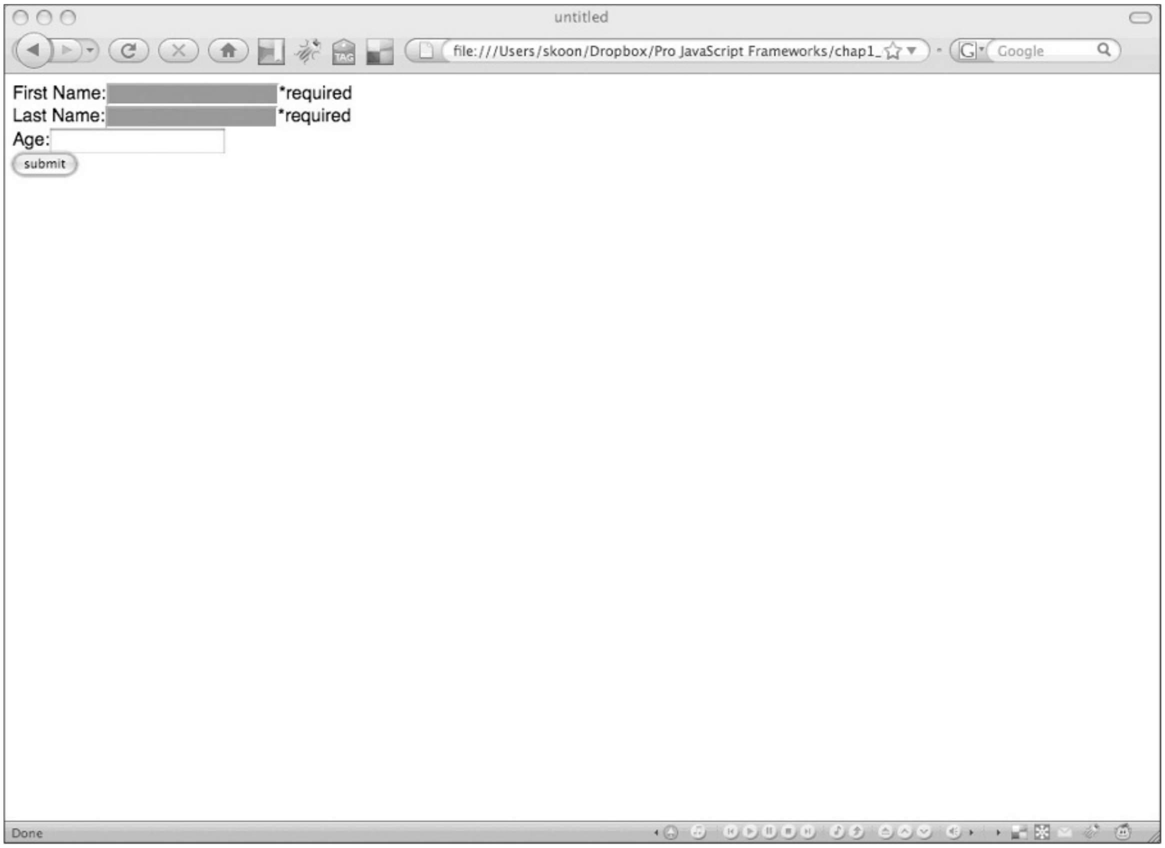
```
<head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
      <title>untitled</title>
    <style>
        .invalid { background:red;}
    </style>
</head>
<body>
    <form id="myForm" method="post">
        First Name:<input type="text" id="firstName" class="required"><br/>
        Last Name:<input type="text" id="lastName" class="required"><br/>
        Age:<input type="text" id="age"><br/>
        <input type="button" id="submitButton" value="submit">
    </form>
    <script type="text/javascript" src="prototype-1.6.0.2.js"></script>
    <script type="text/javascript">

        function init() {
            var requiredInputs = $$(".required");
            for(i = 0;i < requiredInputs.length;i++) {
                $(requiredInputs[i]).insert({"after":"*required"});
                Event.observe(requiredInputs[i], "change", function(e) {
                    if(this.hasClassName("invalid")) { this.removeClassName
("invalid")};
                });
            };
            Event.observe("submitButton", "click", validateUserInput);

        };

function validateUserInput() {
            var requiredInputs = $$(".required");
              for(var i = 0; i < requiredInputs.length; i++) {
                if(requiredInputs[i].value == "") {
                    requiredInputs[i].addClassName("invalid");
                } else {
                    if (requiredInputs[i].hasClassName("invalid")) {
                        requiredInputs[i].removeClassName(invalid);
                    };
                };
            };
        };       Event.observe(window, 'load', init);
    </script>
</body>
```

One common task for JavaScript is form validation. Here, you've set up a simple form and defined a simple rule; users have to enter some text into elements that have the "required" class. You can enforce that rule by collecting all of the elements who have the required class using the `$$()` method and passing in a CSS selector. Once you have an array containing those elements, you iterate over the array and check that the value property of each element does not equal an empty string. If it does, you use the `addClass` method to add the `invalid` class to the element. You then also check to see if the class already has the `invalid` class and the user has entered text. If an element contains text and has the `invalid` class, you remove the class since it passes the validation rules, as shown in Figure 1-5 and Figure 1-6, respectively.


untitled
file:///Users/skoon/Dropbox/Pro JavaScript Frameworks/chap1_
Google

First Name: *required
Last Name: *required
Age:
submit

Figure 1-5


untitled
file:///Users/skoon/Dropbox/Pro JavaScript Frameworks/chap1_
Google
submit
Done

First Name: *required
Last Name: *required
Age:

Figure 1-6

hasClassName, classNames

These methods tell you what classes have been applied to the element in question. The `hasClassName` method allows you to determine if a given element has the class name in its `className` property. The `classNames` method has been deprecated; it returns an array containing the classes that have been applied to the element.

setStyle, getStyle

These methods allow you to quickly set styles on your elements and get values for specific styles. You may only query for styles defined by the Document Object Model (DOM) Level 2 Style Specification. To set a style on your element, you pass in an object hash of key-value pairs of the styles you wish to set.

```
El.setStyle( { "font-family": "Arial", "color" : "#F3C" });
```

To get the value for a specific style, pass in the style's name as an argument.

```
El.getStyle("font-size");
```

Internet Explorer returns the literal value while all other browsers return the computed value for styles. For example, if you specify the font-size as 1em, that is what IE will return. Other browsers may return a pixel value for the font-size.

Extending an Element with Your Own Methods

Prototype makes it easy to add your own methods to the `Element` object using the `addMethods` method. The `addMethods` method takes a hash of the methods you want to add. Suppose you want to add a method to any element that will allow you to strip all the whitespace out of the element's text. Here's what that function might look like:

```
function removeWhiteSpace(element) {
    if(element.innerText) {
        return element.innerText.replace(" ", "", "gi");
    } else if(element.textContent){
        return element.textContent.replace(" ", "", "gi");
    }
};
```

First, you need to rewrite the method a little to match what Prototype expects. Then you can call `Element.addMethods`.

```
<body>
    <div id="myDiv">Remove the whitespace</div>
    <script type="text/javascript" src="prototype-1.6.0.2.js"></script>
    <script type="text/javascript">
        var myFunc = {
            removeWhitespace : function (element) {
                  if(element.innerText) {
                    return element.innerText.replace(" ", "", "gi");
                } else if(element.textContent){
                    return element.textContent.replace(" ", "", "gi");
                }
            }
        };

    Element.addMethods(myFunc);

    alert($("myDiv").removeWhitespace());
    </script>
</body>
```

What you did here was wrap your function with an intrinsic object so that addMethods can work its magic. You can take this one step further and return the element itself to allow for chaining.

```
var myFunc = {
    removeWhitespace : function (element) {
        if(element.innerText) {
            element.innertText = element.innerText.replace(" ", "", "gi");
        } else if(element.textContent){
            element.textContent = element.textContent.replace(" ", "", "gi");
        }
        return element;
    }
};
```

So now your method is ready to be used by Prototype.

Summary

In this chapter, you looked at how Prototype makes it easy to obtain a reference to DOM elements by ID, CSS class, and their position relative to other elements. Prototype automatically adds helper methods to your elements when possible, and adds the methods when you use the `Element.extends()`, `$()`, or `$$()` methods to get a reference to the element. Prototype also smoothes out some of the bumps associated with positioning elements and finding out the dimensions of a given element.