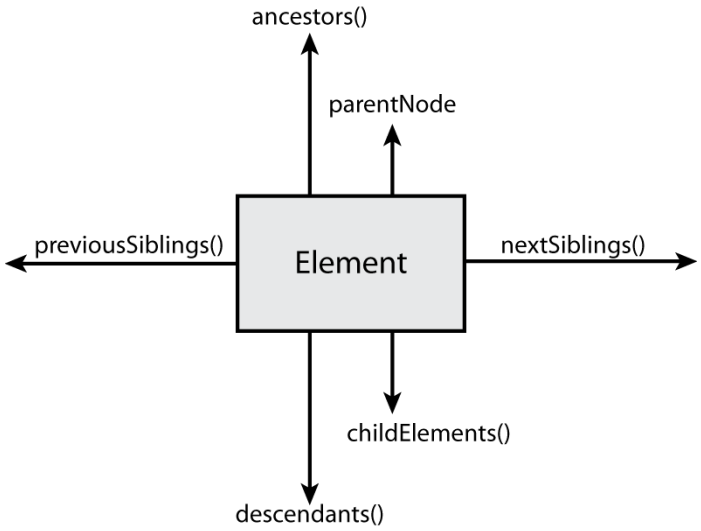

Figure 9.1 An element and its neighboring nodes

Method Name	Description
`ancestors()`	returns an array of the object's ancestors in the DOM tree: parent, grandparents, ...
`siblings()`	returns an array of all siblings of this element (elements only)
`previousSiblings()`, `nextSiblings()`	returns an array of all sibling elements before or after this element
`previous()` `previous(selector)` `next()` `next(selector)`	returns the sibling prior to or after this element (elements only), optionally matching the given CSS selector; if no previous or next sibling is found `undefined` is returned
`adjacent(selector)`	returns an array of all siblings of this element that match the given CSS selector(s)
`childElements()`	returns an array of this element's children (elements only)
`descendantOf(element)`	returns `true` if this element is a child, grandchild, etc. of **element** in the page's DOM tree
`descendants()`	returns an array of the object's children, grandchildren, etc.
`down(selector)` `down(selector, index)`	returns the element's first (or **index**th) descendant that matches the given optional CSS selector; if no selector is given all descendants are considered
`up(selector)` `up(selector, index)`	returns the element's first (or **index**th) ancestor that matches the given optional CSS selector; if no selector is given all descendants are considered

Table 9.8 Prototype's DOM node traversal methods

For example, if you have an element on your page with an `id` of `main` and you want to change the text of its neighbors to end with an exclamation point, you could write the code in Example 9.10.

```
var siblings = $("main").siblings();
for (var i = 0; i < siblings.length; i++) {
  siblings[i].innerHTML += "!";
}
```

Example 9.10 Processing siblings using Prototype's DOM node methods

Notice that `siblings`, like the rest of Prototype's DOM additions, is a method and must be called with parentheses, unlike the built-in DOM properties `firstChild`, `nextSibling`, etc.

Selecting Groups of Nodes

In the previous chapter we saw that you can call `document.getElementsByTagName` to get an array of DOM elements using a given HTML tag. This was useful for looping over many elements and processing them. We used this technique in our searchable fan site example in the DOM chapter.

But the ability to process all nodes with a given tag is not very versatile. Often we want to process a group of nodes that are related in some other way, such as all elements that have the `urgent` CSS class, or all `li` items that are inside an `ol` inside a particular `div` on the page. The existing DOM doesn't make it easy to do these kinds of manipulations. But Prototype comes to our rescue.

Prototype adds some methods to the `document` object (and to all other DOM objects) for accessing groups of nodes. The most useful of these methods is listed in Table 9.9. The `$$` function is especially useful for processing all elements that match a given CSS selector. You can set up your page so that the elements of interest use a particular class, and then target that class.

Method Name	Description
`select(selector)` (a.k.a. `$$` when called on `document`)	returns array of all DOM elements that match the given CSS selector string, such as `"div#sidebar > li"`

Table 9.9 Prototype's DOM selection methods

But even more useful is Prototype's ability to select a group of elements based on CSS selector strings. Recall that a selector is the part of a CSS rule that comes before the `{` brace, that determines which parts of the page the rule will affect. Selectors can be simple, such as `"p"` to select all paragraphs, or complex, such as `"div#topmenu > img.banner"` to select all `img` elements with a `class` of `banner` that reside directly inside the `div` element with an `id` of `topmenu`.

Prototype's mighty `select` method, which has been added to the `document` object as well as to every other DOM object, accepts a string parameter representing a CSS selector, and returns an array of all DOM objects for elements that match that selector. The selector you pass is a string that follows the same syntax as a CSS selector does. For example, if you want to hide all paragraphs with the class of `announcement` that are contained within the `div` with `id` of `news`, you could write:

```
var paragraphs = document.select("div#news p.announcement");
for (var i = 0; i < paragraphs.length; i++) {
  paragraphs[i].hide();
}
```

Since you often want to call `select` on the overall `document` object, and to reduce typing, Prototype introduces a `$$` function that is a shorthand for `document.select`. You can select a group of elements using `$$` even if your CSS file has no style rule for that group. For example, it is fine to use `$$` to select all paragraphs even if your CSS file does have a rule for the `p` tag. (We jokingly refer to `$$` as the "mo' money" function with our students.) For example, Example 9.11 accomplishes the same thing as the previous code, hiding the same group of paragraphs.

```
var paragraphs = $$("div#news p.announcement");
for (var i = 0; i < paragraphs.length; i++) {
  paragraphs[i].hide();
}
```

Example 9.11 Selecting elements with $$

$$ can also be used in a window.onload handler to attach event handlers to a group of elements. For example, the code in Example 9.12 listens to clicks on all buttons with a class of control directly inside of the section with id of game.

```
window.onload = function() {
  var gameButtons = $$("#game > button.control");
  for (var i = 0; i < gameButtons.length; i++) {
    gameButtons[i].onclick = gameButtonClick;
  }
};

function gameButtonClick() {
  ...
}
```

Example 9.12 Attaching event handlers with $$

$$ effectively replaces all of the other multi-element selection techniques and is simpler and more pleasant to use. Therefore it's our recommended way of accessing groups of nodes, and it's what we'll use in our examples in the rest of the textbook.

Common Error

Misusing $$

Many students make simple mistakes when first using Prototype's $$ function. For example, it's easy to forget to write . or # in front of a desired class or id that you want to select. Example 9.13 shows two attempts to use $$ to fetch all elements with the class of control. But the first call will return an empty array, because there is no element with a tag of control. The second line shows a correct call.

```
var gameButtons = $$("control");    // incorrect
var gameButtons = $$(".control");   // correct
```

Example 9.13 Common error: Forgetting . or # when using $$

Another important thing to remember is that $$ does not return the same type of value as $. While $ returns a single DOM element object, $$ returns an array of the elements it matched. If you want to do something to those DOM objects, you must loop over the results and process each one. The first line in Example 9.14 incorrectly attempts to set buttons with a class of control to have red text. The code afterward correctly loops over each element and applies the style to it.

```
$$(".control").style.color = "red";     // incorrect

var gameButtons = $$(".control");
for (var i = 0; i < gameButtons.length; i++) {
  gameButtons[i].style.color = "red";   // correct
}
```

Example 9.14 Common error: Treating $$ return value as a single element

Even if the selector you pass to `$$` matches only a single element, the result is still returned as an array (an array containing just one element). You must either loop over the elements returned (if there will be many of them) or access elements at particular indexes. If you're sure the array will contain just one element, you can just directly refer to its element `[0]`, as shown in Example 9.15.

```
$$(".control")[0].style.color = "red";   // correct
```

Example 9.15 Modifying a single element using $$

9.1.6 Prototype and Forms

Prototype includes several useful features for dealing with HTML forms. Since a common JavaScript usage pattern is to get a DOM object for a form control and then examine or set its value, Prototype also includes a variant of `$` called `$F`. The `$F` function accepts a form control's `id` string as its parameter and returns the `value` of that control. For example, for an input text control, `$F` returns the text typed into that text box. Example 9.16 shows the syntax for the `$F` function.

```
var name = $F("id");
```

Example 9.16 Syntax template for accessing form control values

`$F` does not provide any functionality that cannot be achieved through other means such as the `$` function and the `value` property, but it is a concise way to get a form control's value.

Prototype also adds a set of additional methods to every form control that are useful for manipulating the control and its value. Several of the useful methods are shown in Table 9.10. For example, if a form has a text box with an `id` of `tip` and you want to clear the box's text if it represents an integer less than 10, you could write code such as the following:

```
if ($F("tip") < 10) {
  $("tip").clear();    // erase text if number typed is too small
}
```

Method Name	Description
`activate()`	gives the control focus and selects its text, if any
`clear()`	removes any text from the control
`disable()`, `enable()`	disables or enables the form control's value from being changed
`focus()`	gives the control focus
`getValue()`	returns the current value of the control (usually a string, but an array of strings for multiple-select list boxes); a longhand for `$F`
`present()`	returns `true` if there is any text typed into the control
`select()`	selects/highlights the text in the control

Table 9.10 Prototype form control element methods

Self-Check

1. What are the contents of the following array after the following code runs?

```
var nums = [2.7, 5.1, 18.6, 5.1, 2.7, 16, 27];
nums = nums.uniq().without(16);
for (var i = 0; i < nums.length; i++) {
  a[i] = a[i].round();
  if (a[i] % 2 == 0) {
    a[i] = a[i].toColorPart().times(3);
  }
}
```

2. Suppose we have an element with `id` of `box`. How would we use Prototype to make this element invisible on the page? How would we delete the element from the page?
3. Suppose we have an element with `id` of `box`. How would we use Prototype to check whether the element's font is bold, and if so, to also make the font become italic?
4. In Prototype's terminology, what is the difference between a parent node and an ancestor node? Between a child and a descendant?
5. How would you use Prototype to retrieve the DOM objects for all paragraph (`p`) elements on the page that have the CSS class of `story` that are inside the div with `id` of `container`?