

How to do it...

Let's assume you have a single button element on a page. You can locate this button by using its tag in the following way:

```
WebElement loginButton = driver.findElement(By.tagName("button"));
loginButton.click();
```

Take another example where we want to count how many rows are displayed in `<table>`. We can do this in the following way:

```
WebElement table = driver.findElement(By.id("summaryTable"));
List<WebElement> rows = table.findElements(By.tagName("tr"));
assertEquals(10, rows.size());
```

How it works...

The `tagName()` locator method queries the DOM and returns a list of matching elements for the specified tag name. This method may not be reliable while locating individual elements and the page might have multiple instances of these elements.

See also

- The *Finding elements using `findElements` method* recipe

Finding elements using XPath

XPath (the XML path language) is a query language used to select nodes from an XML document. All the major browsers implement DOM Level 3 XPath (using <http://www.w3.org/TR/DOM-Level-3-XPath/>) specification, which provides access to a DOM tree.

The XPath language is based on a tree representation of the XML document and provides the ability to navigate around the tree and to select nodes using a variety of criteria.

Selenium WebDriver supports XPath to locate elements using XPath expressions, also known as XPath query.

One of the important differences between XPath and CSS is that, with XPath, we can search elements backwards or forwards in the DOM hierarchy, while CSS works only in a forward direction. This means that using XPath we can locate a parent element using a child element and vice versa.

In this recipe, we will explore some basic XPath queries to locate elements, and then examine some advanced XPath queries.

XML documents are treated as trees of nodes. The topmost element of the tree is called the root element. When an HTML document is loaded in DOM, it provides a similar tree of nodes. Here's an example of an HTML page:

```
<html>
  <head>
    <title>My Book List</title>
  </head>
  <body>
    <h1>My Book List</h1>
    <div>
      <table class="main-list">
        <tr>
          <td>Title</td>
          <td>Author</td>
          <td>Publication Year</td>
          <td>Price</td>
          <td>Book Page</td>
        </tr>
        <tr id="book_1">
          <td>XML Developer's Guide</td>
          <td>Gambardella, Matthew</td>
          <td>Publication Year</td>
          <td class="price">44.95</td>
          <td><div class="desc">An in-depth look at creating applications
            with XML.</div></td>
          <td><a href="/book_1.html">
            
            </a></td>
        </tr>
      </table>
    </div>
  </body>
</html>
```

Let's understand some basic XPath terminology before we move on to using XPath, with the following listed terms. We will use the previous HTML document as an example:

Term	Description
Nodes	<p>DOM represents an HTML document as trees of nodes. Here are examples of nodes from the previous HTML document:</p> <ul style="list-style-type: none"> ▶ <code>html</code>: This is the root element node ▶ <code>title</code>: This is the element node ▶ <code>id="book_1"</code>: This represents the attributes and values <p>The topmost element of the tree is called the root node or element.</p>
Atomic Values	<p>Atomic values are nodes with no children or parents. For example:</p> <p>Gambardella, Matthew</p> <p>XML Developer's Guide</p> <p>44.95</p>
Parents	Each element and attribute has one parent. For example, the <code>body</code> element is the parent of <code>div</code> . Similarly, <code>div</code> is the parent of the <code>table</code> element.
Children	Element nodes may have zero, one, or more children. For example, there are two <code>tr</code> elements, which are children of the <code>table</code> element.
Siblings	Nodes that have the same parent. For example, <code>h1</code> and <code>div</code> are all siblings and their parent is the <code>body</code> element.
Ancestors	A node's parent, parent's parent, and so on. For example, ancestors of the <code>table</code> element are <code>div</code> , <code>body</code> and <code>html</code> .
Descendants	A node's children, children's children, and so on. For example, the descendants of the <code>table</code> element are <code>tr</code> , <code>td</code> and <code>div</code> .

Selecting nodes

XPath uses path expressions to select nodes from the tree. The node is selected by following a path or steps. The most useful path expressions are listed as follows:

Expression	Description
<code>nodename</code>	This will select all nodes with the name "nodename". For example, <code>table</code> will select all the table elements.
<code>/</code> (slash)	<p>This will select element(s) relative to the root element. For example:</p> <ul style="list-style-type: none">▶ <code>/html</code>: This will select the root HTML element. A slash (/) is used in the beginning and it defines an absolute path.▶ <code>html/body/table</code>: will select all table elements that are children of HTML. <p>The slash (/) is used at the start of a code element, and it defines an absolute path. It defines ancestor and descendant relationships if used in the middle; for example, <code>//div/table</code> returns the <code>div</code> containing a <code>table</code> object.</p>
<code>//</code> (double slash)	<p>This will select node(s) in the document from the current node that match the selection irrespective of its position. For example:</p> <ul style="list-style-type: none">▶ <code>//table</code> will select all the table elements no matter where they are in the document▶ <code>//tr//td</code> will select all the <code>td</code> elements▶ <code>//a//img</code> will select all the <code>img</code> elements that are children of the "a" (anchor) element <p>Double slash (//) defines a descendant relationship if used in the middle; for example, <code>/html//title</code> returns the <code>title</code> element that is descendant of the <code>html</code> element.</p>
<code>.</code> (dot)	This represents the current node.
<code>..</code> (double dot)	This will select the parent of the current node. For example, <code>//table/..</code> will return the <code>div</code> element.
<code>@</code>	<p>This represents an attribute. For example:</p> <ul style="list-style-type: none">▶ <code>//@id</code>: This will select all the elements where the <code>id</code> attribute are defined no matter where they are in the document▶ <code>//img/@alt</code>: This will select all the <code>img</code> elements where the <code>@alt</code> attribute is defined

How to do it...

Let's explore some basic XPath expressions that can be used in Selenium WebDriver. Selenium WebDriver provides the `xpath()` method to locate elements using XPaths.

Finding elements with an absolute path

XPath absolute paths refer to the very specific location of the element, considering its complete hierarchy in the DOM. Here is an example where the **Username Input** field is located using the absolute path. When providing an absolute path, a space is given between the elements:

```
WebElement userName =  
    driver.findElement(By.xpath("/html/body/div/div/form/input"));
```

However, this strategy has limitations as it depends on the structure or hierarchy of the elements on a page. If this changes, the locator will fail to get the element.

Finding elements with a relative path

With a relative path, we can locate an element directly irrespective of its location in the DOM. For example, we can locate the **Username Input** field in the following way, assuming it is the first `<input>` element in the DOM:

```
WebElement userName = driver.findElement(By.xpath("//input"));
```

Finding elements using predicates

A predicate is embedded in square brackets and is used to find out specific node(s) or a node that contains a specific value.

In the previous example, the XPath query will return the first `<input>` element that it finds in the DOM. There could be multiple elements matching the specified XPath query. If the element is not the first element, we can also locate the element by using its index in the DOM. For example, in our login form, we can locate the **Password** field, which is the second `<input>` element on the page, in the following way:

```
WebElement userName = driver.findElement(By.xpath("//input[2]"));
```

Finding elements using attributes values with XPath

We can find elements using their attribute values in XPath. In the following example, the **Username** field is identified using the ID attribute:

```
WebElement userName =  
    driver.findElement(By.xpath("//input[@id='username']"));
```

Here is another example where the image is located using the `alt` attribute:

```
WebElement previousButton =  
    driver.findElement(By.xpath("//img[@alt='Previous']"));
```

You might come across situations where one attribute may not be sufficient to locate an element and you need combined additional attributes for a precise match. In the following example, multiple attributes are used to locate the `<input>` element for the **Login** button:

```
WebElement previousButton =  
    driver.findElement(By.xpath  
        ("//input[@type='submit'][@value='Login']"));
```

The same result can be achieved by using XPath and operator:

```
WebElement previousButton = driver.findElement  
    (By.xpath("//input[@type='submit' and @value='Login']"));
```

In the following example, either of the attributes is used to locate the elements using XPath or operator:

```
WebElement previousButton = driver.findElement  
    (By.xpath("//input[@type='submit' or @value='Login']"));
```

Finding elements using attributes with XPath

This strategy is a bit different from the earlier strategy where we want to find elements based only on the specific attribute defined for them but not attribute values. For example, we want to lookup all the `` elements that have the `alt` attribute specified:

```
List<WebElement> imagesWithAlt = driver.findElements  
    (By.xpath ("//img[@alt]"));
```

Here's another example where all the `` elements will be searched and where the `alt` attribute is not defined. We will use the `not` function to check the negative condition:

```
List<WebElement> imagesWithAlt = driver.findElements  
    (By.xpath ("//img[not (@alt)]"));
```

Performing partial match on attribute values XPath also provides a way to find elements matching partial attribute values using XPath functions. This is very useful to test applications where attribute values are dynamically assigned and change every time a page is requested. For example, ASP.NET applications exhibit this kind of behavior where IDs are generated dynamically.

The following table explains the use of these XPath functions:

Syntax	Example	Description
<code>starts-with()</code>	<code>input[starts-with(@id, 'ctrl')]</code>	Starting with: For example, if the ID of an element is <code>ctrl_12</code> , this will find and return elements with <code>ctrl</code> at the beginning of the ID.
<code>ends-with()</code>	<code>input[ends-with(@id, '_userName')]</code>	Ending with: For example, if the ID of an element is <code>a_1_userName</code> , this will find and return elements with <code>_userName</code> at the end of the ID.
<code>contains()</code>	<code>Input[contains(@id, 'userName')]</code>	Containing: For example, if the ID for an element is <code>panel_login_userName_textfield</code> , this will use the <code>userName</code> part in the middle to match and locate the element.

Matching any attribute using a value

XPath matches the attribute for all the elements for a specified value and returns the element. For example, in the following XPath query, `'userName'` is specified. XPath will check all the elements and their attributes to see if they have this value and return the matching element.

```
WebElement userName =
    driver.findElement(By.xpath("//input[@*='username']"));
```

Here are more examples of using XPath predicates to find elements using their position and contents:

Expression	Description
<code>/table/tr[1]</code>	This will select the first <code>tr</code> (row) element that is the child of the <code>table</code> element.
<code>/table/tr[last()]</code>	This will select the last <code>tr</code> (row) element that is the child of the <code>table</code> element.
<code>/table/tr[last()-1]</code>	This will select the second last <code>tr</code> (row) element that is the child of the <code>table</code> element.
<code>/table/tr[position()>4]</code>	This will select the three <code>tr</code> (rows) elements that are child of the <code>table</code> element.
<code>//tr[td>40]</code>	This will select all the <code>tr</code> (rows) elements that have one of their children <code>td</code> with value greater than 40.

Selecting unknown nodes

Apart from selecting the specific nodes, XPath also provides wildcards to select a group of elements:

Wildcard	Description	Example
*	Matches any element node.	<ul style="list-style-type: none">▶ <code>/table/*</code>: This will select all child elements of a table element▶ <code>//*</code>: This will select all elements in the document▶ <code>//*[@class='price']</code>: This will select any element in the document which has an attribute named <code>class</code> with a specified value, that is <code>price</code>
@	Matches any attribute node.	<ul style="list-style-type: none">▶ <code>//td[@*]</code>: This will select all the <code>td</code> elements that have any attribute
<code>node()</code>	Matches any node of any kind.	<ul style="list-style-type: none">▶ <code>//table/node()</code>: This will select all the child elements of <code>table</code>

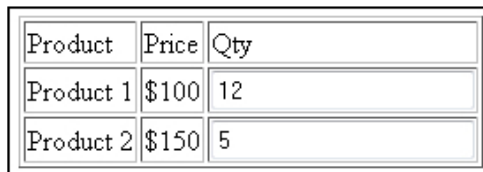
Selecting several paths

Using the union `|` operator in XPath expressions, we can select several paths together, as shown in the following table:

Path Expression	Action
<code>//div /p //div/span</code>	This will select all the <code>p</code> (paragraph) and <code>span</code> elements of the <code>div</code> element.
<code>//p //span</code>	This will select all the <code>p</code> (paragraph) and <code>span</code> elements in the document.

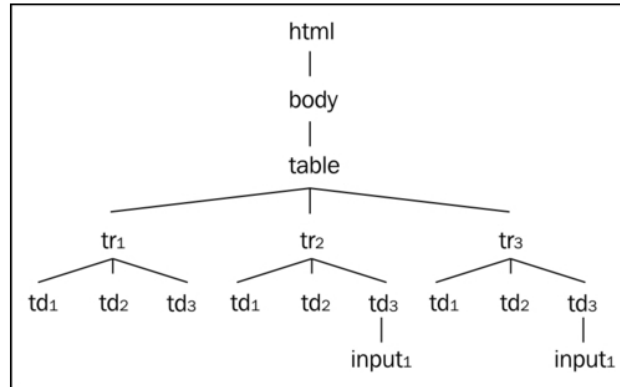
Locating elements with XPath axes

XPath axes help to find elements based on the element's relationship with other elements in a document. The following screenshot shows some examples for some common XPath axes used to find elements from a `<table>` element. This can be applied to any other element structure from your application.



Product	Price	Qty
Product 1	\$100	12
Product 2	\$150	5

The following image shows a graphical representation of the HTML elements:



Axis	Description	Example	Result
ancestor	Selects all ancestors (parent, grandparent, and so on) of the current node.	<code>//td[text()='Product 1']/ancestor::table</code>	This will get the table element.
descendant	Selects all descendants (children, grandchildren, and so on) of the current node.	<code>/table/descendant::td/input</code>	This will get the input element from the third column of the second row from the table.
following	Selects everything in the document after the closing tag of the current node.	<code>//td[text()='Product 1']/following::tr</code>	This will get the second row from the table.
following-sibling	Selects all siblings after the current node.	<code>//td[text()='Product 1']/following-sibling::td</code>	This will get the second column from the second row immediately after the column that has Product 1 as the text value.

preceding	Selects all nodes that appear before the current node in the document, except ancestors, attribute nodes, and namespace nodes.	//td[text()=' \$150']/preceding::tr	This will get the header row.
preceding-sibling	Selects all siblings before the current node.	//td[text()=' \$150']/preceding-sibling::td	This will get the first column of third row from the table.

You can find more about XPath axes at http://www.w3schools.com/xpath/xpath_axes.asp.

How it works...

XPath is a powerful language to query and process DOM trees in browsers. XPath is used to navigate through elements and attributes in a DOM tree. XPath provides various rules, functions, operators, and syntax to find the elements.

The majority of browsers support XPath, and Selenium WebDriver provides the ability to find elements using the XPath language.

Using the `xpath()` method of the `By` class, we can locate elements using XPath syntax.

Finding elements using CSS selectors

The Cascading Style Sheets (CSS) is a style sheet language used to describe the presentation semantics (the looks and formatting) of a document written in a markup language such as HTML or XML.

Major browsers implement CSS parsing engines to format or style the pages using CSS syntax. CSS was introduced to keep the presentation information separate from the markup or content. For more information on CSS and CSS selectors, visit http://en.wikipedia.org/wiki/Cascading_Style_Sheets.

In CSS, the pattern-matching rules determine which style should be applied to elements in the DOM. These patterns, called **selectors**, may range from simple element names to rich contextual patterns. If all conditions in the pattern are true for a certain element, the selector matches the element, and the browser applies the defined style in CSS syntax.

In this recipe, we will explore some basic CSS selectors and then, later on, we will dive into advanced CSS selectors.

How to do it...

Let's explore some basic CSS selectors that can be used in Selenium WebDriver. Selenium WebDriver's `By` class provides the `cssSelector()` method to find elements using CSS selectors.

Finding elements with an absolute path

CSS absolute paths refer to the very specific location of the element considering its complete hierarchy in the DOM. Here is an example where the **Username Input** field is located using the absolute path. When providing an absolute path, a space is given between the elements, as shown in the following code example:

```
WebElement userName = driver.findElement(By.cssSelector("html body  
div div form input"));
```

You can also use the previous selector in the following way by describing the direct parent-to-child relationships with the `>` separator:

```
WebElement userName = driver.findElement(By.cssSelector("html >  
body > div > div > form > input"));
```

However, this strategy has limitations as it depends on the structure or hierarchy of the elements on a page. If this changes, the locator will fail to find the element.

Finding elements with a relative path

With a relative path, we can find an element directly, irrespective of its location in the DOM. For example, we can find the **Username Input** field in the following way, assuming it is the first `<input>` element in the DOM:

```
WebElement userName = driver.findElement(By.cssSelector("input"));
```

The following CSS selectors use Class and ID attributes to find elements using relative paths. This is the same as the `className()` and `id()` locator methods. However, there is another strategy where we can use any other attribute of the element that is not covered in the `By` class.

Finding elements using the Class selector

While finding elements using the CSS selector, we can use the `Class` attribute to locate an element. This can be done by specifying the type of HTML tag, then adding a dot followed by the value of the `class` attribute in the following way:

```
WebElement loginButton =  
driver.findElement(By.cssSelector("input.login"));
```

This will find the **Login** button's `<input>` tag whose `Class` attribute is `login`.

Sometimes, multiple CSS classes are given for an element. For example:

```
<input type="text"
      class="username textfield" />
```

In this case, we can use multiple class names, as shown in the following example:

```
WebElement loginButton =
    driver.findElement(By.cssSelector("input.login.textfield"));
```

There is also a shortcut where you can put a "." (period) and class attribute value and ignore the HTML tag. However, this will return all the elements with the class as `login` and the test may not return the correct element, as shown in the following code example:

```
WebElement loginButton =
    driver.findElement(By.cssSelector(".login"));
```

This method is similar to the `className()` locator method.

Finding elements using the ID selector

We can find an element using the ID attribute. This can be done by specifying the type of HTML tag, then entering a # (hash) followed by the value of the `Class` attribute, as shown in the following code:

```
WebElement userName =
    driver.findElement(By.cssSelector("input#username"));
```

This will return the username `<input>` element using its `id` attribute.

There is also a shortcut where you can enter # and a class attribute value and ignore the HTML tag. However, this will return all the elements with the `id` set as `username` and the test may not return the correct element. This has to be used very carefully:

```
WebElement userName =
    driver.findElement(By.cssSelector("#username"));
```

This method is similar to the `id` locator strategy.

Finding elements using the attributes selector

Apart from the `class` and `id` attributes, CSS selectors also enable the finding of elements using other attributes of the element. In the following example, the `Name` attribute is used to locate an `<input>` element:

```
WebElement userName =
    driver.findElement(By.cssSelector("input[name=username]"));
```

Using the `name` attribute to locate an element is similar to the `name()` locator method of the `By` class.

Let's use some other attributes to find an element. In the following example, the `` element is located by using its `alt` attribute:

```
WebElement previousButton =
    driver.findElement(By.cssSelector("img[alt='Previous']"));
```

You might come across situations where one attribute may not be sufficient to find an element and you need to combine additional attributes for a precise match. In the following example, multiple attributes are used to locate the **Login** button's `<input>` element:

```
WebElement previousButton =
    driver.findElement(By.cssSelector("input[type='submit']
    [value='Login']"));
```

Finding elements using the attributes name selector

This strategy is a bit different from the earlier strategy where we want to find elements based on only the specific attribute defined for them but not attribute values. For example, we want to look up all the `` elements that have the `alt` attribute specified:

```
List<WebElement> imagesWithAlt =
    driver.findElements(By.cssSelector("img[alt]"));
```

A Boolean `not()` pseudo-class can also be used to find elements not matching the specified criteria. For example, to find all the `` elements that do not have the `alt` attribute, the following method can be used:

```
List<WebElement> imagesWithoutAlt =
    driver.findElements(By.cssSelector("img:not([alt])"));
```

Selecting several paths

Using the `or` selector `,` in CSS selectors, we can select a single or several elements matching the given criteria, as shown in the following code:

```
List<WebElement> elements =
    driver.findElements(By.cssSelector("div, p"));
```

This will select all the `<div>` and all the `<p>` elements:

```
List<WebElement> elements =
    driver.findElements(By.cssSelector("div.first, div.last"));
```

This will select `<div>` with class `first` and `last`.

Performing a partial match on attribute values

CSS selector provides a way to find elements matching partial attribute values. This is very useful for testing applications where attribute values are dynamically assigned and change every time a page is requested. For example, ASP.NET applications exhibit this kind of behavior, where IDs are generated dynamically. The following table explains the use of CSS partial match syntax:

Syntax	Example	Description
<code>^=</code>	<code>input[id^='ctrl']</code>	Starting with: For example, if the ID of an element is <code>ctrl_12</code> , this will find and return elements with <code>ctrl</code> at the beginning of the ID.
<code>\$=</code>	<code>input[id\$='_userName']</code>	Ending with: For example, if the ID for an element is <code>a_1_userName</code> , this will find and return elements with <code>_userName</code> at the end of the ID.
<code>*=</code>	<code>Input[id*='userName']</code>	Containing: For example, if the ID of an element is <code>panel_login_userName_textfield</code> , this will use the <code>userName</code> part in the middle to match and find the element.

How it works...

CSS selector is a pattern and the part of a CSS rule that matches a set of elements in an HTML or XML document.

The majority of browsers support CSS parsing for applying styles to these elements. Selenium WebDriver uses a CSS parsing engine to locate the elements on a page. CSS selectors provide various methods, rules, and patterns to locate the element on a page.

Using CSS selector, the test can find elements in multiple ways using Class, ID, attribute values, and text contents, as described in this recipe.

See also

- The *Finding elements using advanced CSS selectors* recipe

Locating elements using text

When testing web applications, you will also encounter situations where developers don't assign any attributes to the elements and it becomes difficult to locate elements.

Using the CSS selectors or XPath, we can find elements based on their text contents. In this recipe, we will explore methods to find elements using text values.

How to do it...

To find elements using their text contents, CSS selectors and XPath provide methods to find text within the elements. If an element contains specific text, this will return the element in the test.

Using XPath's text function

XPath provides the `text()` function, which can be used to see if an element contains the specified text in the following way:

```
WebElement cell = driver.findElement  
    (By.xpath("//td[contains(text(),'Item 1')]"));
```

We can also use a single period/dot, `."`, instead of the `text()` function in following way:

```
WebElement cell = driver.findElement  
    (By.xpath("//td[contains(.,'Item 1')]"));
```

Here, we are using the `contains` function along with the `text()` function. The `text()` function returns the complete text from the element and the `contains()` function checks for the specific value that we have mentioned.

XPath also offers the `normalize-space()` function to match the element using the element's and its sub-element's text.

Finding elements using exact text value in XPath

With XPath, elements can be searched by exact text value in the following way:

```
WebElement cell = driver.findElement  
    (By.xpath("//td[.='Item 1']"));
```

This will locate the `<td>` element matching the exact text.