Watirways

Justin Ko

Watirways

Justin Ko

This book is for sale at http://leanpub.com/watirways

This version was published on 2014-12-17

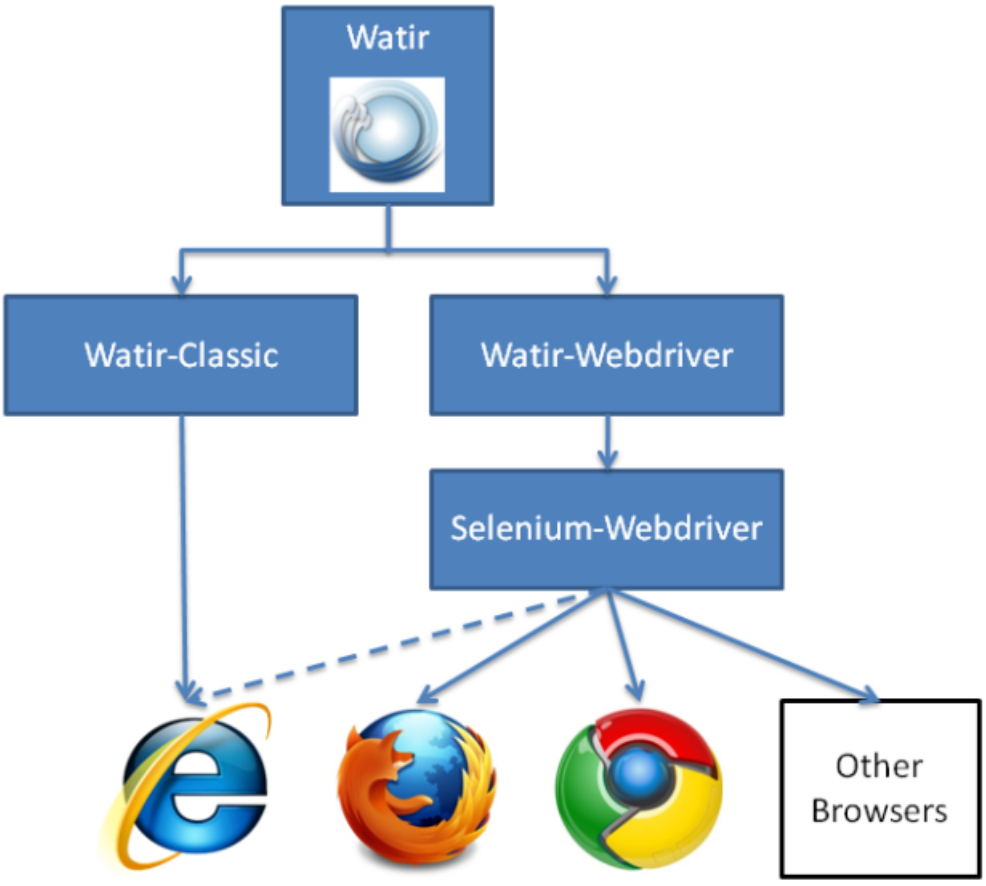
Contents

1. Introduction

1.1 Watir gems

Watir consists of three gems (or projects):

- Watir - This gem is just a loader. Based on the browser that you want to run and your operating system, the Watir gem will load either Watir-Classic or Watir-Webdriver.
- Watir-Classic - This is the original Watir gem that drives Internet Explorer.
- Watir-Webdriver - This gem allows the driving of additional browsers - eg Chrome and Firefox. It is an API wrapper around the Selenium-Webdriver gem (ie translates the Watir commands to Selenium-Webdriver commands).



For most users, the structure of the framework will have no significance or impact. Scripts only need to require Watir; making the use of Watir-Classic versus Watir-Webdriver transparent.

However, there are occasions where understanding the gem relationship is crucial:

- API differences - The Watir-Classic and Watir-Webdriver projects strive to be API compatible though a common specification called Watirspec. Unfortunately, due to historical reasons, implementation details, etc., there are still some API differences. The code found in this book can be assumed to work in both projects unless otherwise specified.
- Monkey patching - When adding functionality to Watir, the code may need to consider which browser/gem is being used.
- Getting help - Specifying the gem being used helps ensure that solutions are applicable.

1.2 Building a script

Say we want to explain to someone how to do a Google search for Watir. We might tell them to:

1. Open a browser
2. Enter the url to go the Google search page
3. Input 'watir' into the search field
4. Click the search button
5. Wait for the search results to load
6. Click the link for the main Watir page

If you look carefully, you will notice that each step consists of two parts:

1. Target - Something to interact with (eg browser, element)
2. Action - Something to do with the target (eg click, type, inspect)

We can make these two parts more clear in a table:

Step	Target	Action
1	browser	open
2	browser	goto url
3	search field	type 'watir'
4	search button	click
5	result link	wait for it to load
6	result link	click

Watir is designed to mimic a user's actions, which means that these same directions can be used when creating an automated script. The only complication is the language barrier. Watir understands Ruby, not English. In essence, building a Watir script is translation exercise.

A translation of the previous table into Watir code would be:

Step	Target	Action
1	browser	Watir::Browser.new
2	browser	goto(‘http://www.google.com’)
3	text_field(:name ⇒ ‘q’)	set(‘watir’)
4	button(:name ⇒ ‘btnG’)	click
5	link(:text ⇒ /Watir.com/)	wait_until_present
6	link(:text ⇒ /Watir.com/)	click

To make an executable script, the target and action need to be combined back into sentences (or lines of code):

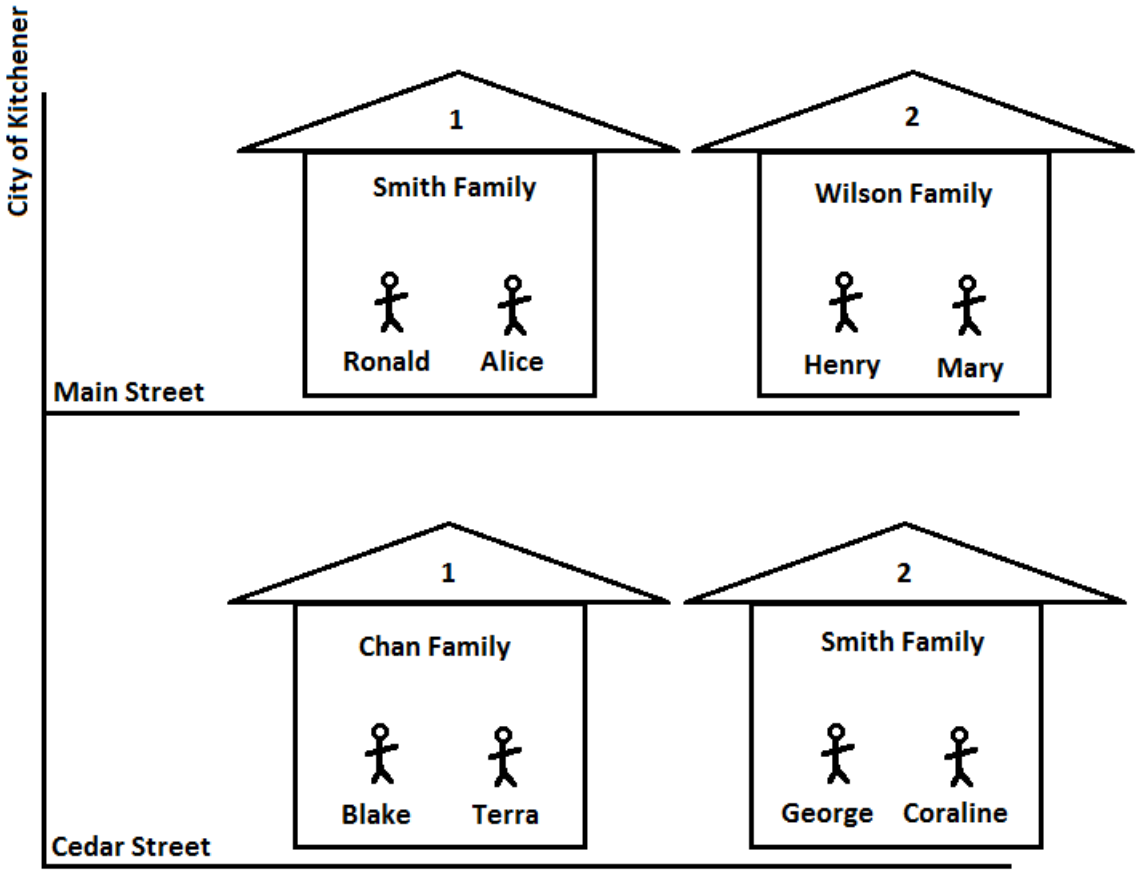
```
browser = Watir::Browser.new
browser.goto('http://www.google.com')
browser.text_field(:name => 'q').set('watir')
browser.button(:name => 'btnG').click
browser.link(:text => /Watir\.com/).wait_until_present
browser.link(:text => /Watir\.com/).click
```

The following chapters will discuss how to translate the targets and actions.

2. Locating an Element - Basics

A locator describes how to find an element amongst all of the other elements on the page. Think of it as giving directions to your house. The uniqueness of the house, who you are talking to, the complexity of your area, etc. affect how detailed the directions need to be.

Say our city has 2 streets and 4 houses with families.



It is unambiguous to say that you want to go the "Wilson Family" house. There is exactly one house that matches that criteria.

What if you say you want to go to the "Smith Family" house? Now it is ambiguous as there are 2 houses that meet that criteria. We can remove the ambiguity by doing one of the following:

1. Providing additional details about the house. For example, the house number is 1.
2. Providing details about where to look for the house. For example, the house is on Main Street.
3. Providing details about what is in the house. For example, the house where Ronald and Alice live.

4. Providing details about what is around the house. For example, the house is beside the Wilson's house.

The same strategies can be used to help Watir find the exact element you want to interact with.

For example, the city's information could be presented as a web page.

```
<div class="city" id="Kitchener">
  <div class="street" id="MainStreet">
    <div class="house" data-name="SmithFamily" data-number="1">
      <span class="resident">Ronald</span>
      <span class="resident">Alice</span>
    </div>
    <div class="house" data-name="WilsonFamily" data-number="2">
      <span class="resident">Henry</span>
      <span class="resident">Mary</span>
    </div>
  </div>
  <div class="street" id="CedarStreet">
    <div class="house" data-name="ChanFamily" data-number="1">
      <span class="resident">Blake</span>
      <span class="resident">Terra</span>
    </div>
    <div class="house" data-name="SmithFamily" data-number="2">
      <span class="resident">George</span>
      <span class="resident">Coraline</span>
    </div>
  </div>
</div>
```

Again, assume that we want to find the "Smith Family" house. To tell Watir to find the div element with the data-name attribute value of "SmithFamily" is ambiguous (ie there are multiple). We can use the same four approaches to remove the ambiguity.

1. Providing additional details about the house. For example, the desired div element has a data-number attribute value of 1.
2. Providing details about where to look for the house. For example, the desired div element is within the div element that has the id of MainStreet.
3. Providing details about what is in the house. For example, the desired div element that includes the spans with text Ronald and Alice.
4. Providing details about what is around the house. For example, the desired div element is beside the div element with data-name attribute of "WilsonFamily".

These four strategies can be conceptually generalized based on the relationship of the element we want to find and the structure of elements. The strategies are to use the properties of:

1. The Element - This is the specific element that you want to find.
2. Ancestors - These are the elements that contain the desired element.
3. Decedents - These are the elements contained within the desired element.
4. Siblings - These are elements that share the same parent as the desired element.

The following image illustrates the relationship of the elements.


```
<div class="city" id="Kitchener">
    <div class="street" id="MainStreet">
        <div class="house" data-name="SmithFamily" data-number="1">
            <span class="resident">Ronald</span>
            <span class="resident">Alice</span>
        </div>
        <div class="house" data-name="WilsonFamily" data-number="2">
            <span class="resident">Henry</span>
            <span class="resident">Mary</span>
        </div>
    </div>
    <div class="street" id="CedarStreet">
        <div class="house" data-name="ChanFamily" data-number="1">
            <span class="resident">Blake</span>
            <span class="resident">Terra</span>
        </div>
        <div class="house" data-name="SmithFamily" data-number="2">
            <span class="resident">George</span>
            <span class="resident">Coraline</span>
        </div>
    </div>
</div>
```

2.1 Concepts

An element's properties, such as attributes, text, label, etc., can be used to be specific about which element to find.

Single vs multiple properties

An element can be located using 0 properties, which returns the first element.

```
browser.div
```

To locate an element with specific properties, create a hash of the properties. A hash looks like:

```
:property1 => 'value1', :property2 => 'value2'
```

Where:

- The keys of the hash are the properties to test or the how to match the element.

- The values of the hash are the expected property values or the what to match.

The hash can be used to match a single property. For example, the following says to find a div element that has the id “my_id”.

```
browser.div(:id => 'my_id')
```

The hash can be expanded to include any number of properties. The following locates a div element that has the class “enabled”, the text “Navigate” and name attribute containing the word “navigate”.

```
browser.div(:class => 'enabled', :text => 'Navigate', :name => /navigate/)
```

Exact vs partial match

When looking for matching elements, Watir can check if the property is an exact match or a partial match. This is controlled by the value in the locator.

- A String object will perform an exact match.
- A Regular Expression (Regexp) object will perform a partial match.

For example, the following locator is using a String (denoted by the quotation marks) for the value of the :id property. This means that it will only match elements whose id attribute is exactly “my_id”. It will not match elements whose id is “before_my_id” or “my_id_after”.

```
browser.div(:id => 'my_id')
```

If you want to match elements whose id attribute contains “my_id” anywhere (ie a partial match), you can use a Regexp (denoted by the forward slashes). The following locator will match elements that have the id “my_id”, as well as “before_my_id” or “my_id_after”.

```
browser.div(:id => /my_id/)
```

2.2 Element type

When locating an element, Watir needs to be told:

- What type of element to find and
- How many matches to return.

This is achieved by calling the corresponding element type method.

Element vs collection

Watir can find and return an:

- Element - The first (single) matching element or
- Element Collection - All matching elements.

There are two element type methods, for each supported element, that correlate to the return type:

- Singular - This version returns the element.
- Plual - This version returns the element collection.

For example, the *div* method, which is singular, will tell Watir to find the first div element on the page.

```
browser.div
```

The pluralization of *div* is *divs*, which will return all div elements on the page.

```
browser.divs
```

Note that the pluralized version is generally the singular version with an ‘s’ added to the end. However, following English conventions, there will be some element types that have ‘es’ added to the end or have the ending ‘y’ replaced by ‘ies’.

Scenario	HTML Element	Singular	Plural
Add ‘s’		span	spans
Add ‘es’	<progress>	progress	progresses
Add ‘ies’	<summary>	summary	summaries

Standard html element methods

In general, the singular element type method is the same as the html element tag.

For example, the html ul element:

```
<ul id="my_id">
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
```

Would be retrieved by Watir’s *ul* method.

```
browser.ul
```

The following table lists the methods to locate some common element types.

Element Method	Collection Method	HTML Elements Matched
a	as	<a>
div	divs	
h1	h1s	<h1>
li	lis	
span	spans	
table	tables	<table>

Convenience methods

Watir defines additional element type methods that:

- Are more specific. For example, *<input type="checkbox">* can be located using *browser.checkbox* instead of using *browser.input* (which would include non-checkbox inputs).
- Makes visually similar elements equivalent. For example, html has a variety of buttons - *<button>*, *<input type="button">*, etc. They are all equivalent in Watir as each is located with the same *browser.button* method.
- Provide more readable names through aliases. For example, *<a>* elements can be located via *browser.a* or *browser.link*.

The following table lists the convenience element type methods.

Element Method	Collection Method	HTML Elements Matched
button	buttons	<button> <input type="button"> <input type="image"> <input type="reset"> <input type="submit">
checkbox	checkboxes	<input type="checkbox">
element	elements	All
file_field	file_fields	<input type="file">
frame	frames	<frame> <iframe>
hidden	hiddens	<input type="hidden">
image	images	<img>
link	links	<a>
radio	radios	<input type="radio">
select_list	select_lists	<select>

Element Method	Collection Method	HTML Elements Matched
text_field	text_fields	<input type=”password”> <input type=”text”> <textarea>

Custom elements

Some applications use element types that are not in the specifications.

```
<li class="lastMove">
  <div id="81ae2" class="folder">
    <small onclick="someFunction1()"> </small>
  </div>
</li>
```

The small element type is not defined in the html specifications. As a result, Watir does not define a *small* method – ie you cannot do browser.small.click.

To locate these elements, you can use the genaric element method with a tag_name locator.

```
browser.element(:tag_name => 'small')
```

2.3 Attributes

Attributes provide additional infomation (meaning and context) about the element. They appear in the opening tag of an element in the form name=”value”.

In the following element, there are 3 attributes - id, name and class. They have the values “divs_id”, “divs_name” and “divs_class” respectively.

```
<div id="divs_id" name="divs_name" class="divs_class">text</div>
```

Watir can locate an element by its attributes by setting attribute name as the key and the attribute value as the value.

```
browser.div(:attribute_name => "attribute_value")
```

Standard attributes

The html specifications only allow certain attributes to be included on an element. In watir-webdriver, any of these attributes can be used to locate the element. In watir-classic, only the more common attributes are supported (the other attributes must be treated like non-standard attributes).

As an example, consider the id attribute used in the following html. The uniqueness of this attribute makes it an ideal unambiguous locator.

```
<div id="phone_number">
  <span id="area_code">123</span>
  <span id="exchange">456</span>
  <span id="subscriber">7890</span>
</div>
```

An element can be located by an exact id match:

```
browser.span(:id => "exchange").text
#=> "456"
```

Or by a partial id match:

```
browser.span(:id => /area/).text
#=> "123"
```

Class

If something on the page looks different (ex the text size, the background colour, etc), it means that a style has been applied to the element. Often, to reduce maintenance of the page, developers will extract these styles into a style sheet that defines groups of styles. These groups of styles are called classes. An element can use one or more classes by defining the class attribute.

While the class is a standard html attribute, there is special handling by Watir. Consider the following html:

```
<span class="number">123</span>
<span class="bold number">456</span>
<span class="bold number">7890</span>
```

The first span element has a single class called “number”. The other two spans have two classes - “bold” and “number”. Note that in the class attribute value, multiple classes are separated by a space.

When locating an element by its class, watir will try matching each individual class rather than the entire attribute value. For example, locating an element with class “number”, will return elements with a class attribute value “number” as well as “multiple number classes”.

Locating an element based on an exact class match would look like:

```
browser.span(:class => 'number').text
#=> "123"
```

A regular expression can be used to partially match a class name:

```
browser.span(:class => /bo/).text
#=> "456"
```

Data attributes

In some cases, the standard html attributes do not provide enough information for an application. Custom data attributes allow additional information to be added to the element, while still being valid html markup.

For example, a div element that represents a person might use custom data attributes to store the person's height and weight.

```
<div data-height="30cm" data-weight="9kg">Owen</div>
<div data-height="20cm" data-weight="7kg">Izzy</div>
```

The custom data attributes will have the form "data-customname", where "customname" can be any text.

These custom attributes can be used the same as the standard locators. However, for the attribute name, the dash (-) must be replaced by an underscore (_).

Locate an element based on an exact match of the attribute value:

```
browser.div(:data_height => '20cm').text
#=> "Izzy"
```

Locate an element based on a partial match of the attribute value:

```
browser.div(:data_weight => /9/).text
#=> "Owen"
```

Custom attributes

Some applications will still be using attributes that are not in the html specifications (likely due to the application being developed before data attributes were introduced). These attributes are not directly supported by watir as locators. In the case of watir-classic, this also includes some of the defined but less common attributes.

For these attributes, you can use a css or xpath selector.

```
<div>
    <span customattribute="custom1">Custom Attribute 1</span>
    <span customattribute="custom2">Custom Attribute 2</span>
    <span customattribute="custom3">Custom Attribute 3</span>
</div>
```

In css-locators, the square brackets are used for matching attributes.

To locate a span that has the customattribute attribute, of any value:

```
browser.span(:css, 'span[customattribute]').text
#=> "Custom Attribute 1"
```

To locate the span with a specific customattribute value:

```
browser.span(:css, 'span[customattribute="custom2"]').text
#=> "Custom Attribute 2"
```

Xpath is similar to css-locators, however the attribute name must be prefixed with an at symbol (@):

```
browser.span(:xpath, '//span[@customattribute="custom2"]').text
#=> "Custom Attribute 2"
```

2.4 Text

The :text locator allows elements to be located by their text.

```
<div>This is the text of the element.</div>
```

The element can located by an exact text match:

```
browser.div(:text => 'This is the text of the element.')
```

As well as by a partial text match:

```
browser.div(:text => /text of the element/)
```

Note that an element's text is considered all text nodes of the element as well as its descendents.

```
<span id="container">This line has <span id="inner">inner</span> text</span>
```

For this element, the span's text is "This line has inner text". It is not just the element's direct child text nodes - "This line has text".

```
browser.span(:text => 'This line has inner text').exists?
#=> true
```

This is important to remember when locating an element by its text using a regular expression. If locating a span that contains the text "inner", the outer span with id "container" will be returned rather than the inner span.

```
browser.span(:text => /inner/).id
#=> "container"
```

2.5 Label

Elements can also be located by their associated label element's text. Note that this only works in Watir-Webdriver.

In the following html, the input field has an associated label with the text "First Name:".

```
<label for="first_name">First Name:</label>
<input type="text" id="first_name" name="first_name" />
```

The input field can be located by the exact label text:

```
browser.text_field(:label => 'First Name:')
```

Or by part of the label text:

```
browser.text_field(:label => /Name/)
```

2.6 Index

When locating a single element, Watir will, by default, return the first element matching the criteria.

For example, given the html:

```
<div>
  <a href="http://www.watir.com">Watir</a>
  <a href="http://watirwebdriver.com">Watir-Webdriver</a>
</div>
```

The first matching element is returned when no :index is specified or when the :index value is 0.

```
browser.link.text
#=> "Watir"


browser.link(:index => 0).text
#=> "Watir"
```

The second matching element is found using an :index with value 1. Notice that Watir is using a 0-based index - ie 0 is the first element, 1 is the second element, etc.

```
browser.link(:index => 1).text
#=> "Watir-Webdriver"
```

A couple of items note:

- The value can be an integer or a string.
- The :index locator only applies to locating a single element. An exception will occur when using it to locate an element collection.

```
browser.divs(:text => /Watir/, :index => 1).length
#=> Watir::Exception::MissinWayOfFindingObjectException
```

2.7 CSS

The :css locator allows a css-selector to be used to find an element.

```
browser.div(:css => 'div#my_id')
```

For more details on creating a css-selector, see the css-selector specifications[1].

2.8 Xpath

The :xpath locator allows an element to be located based on its path.

```
browser.div(:xpath => '//div[@id="my_id"]')
```

For more details on determining an element's xpath, see the xpath specifications[2].

[1] http://www.w3.org/TR/css3-selectors/
[2] http://www.w3.org/TR/xpath20/

3. Locating an Element - Relationships

3.1 Based on its ancestors

Any element that contains the target element is considered an “ancestor”. Ancestor elements can be used to reduce the scope of where Watir searches for an element. The element type method will only look for the element within the html of the calling object.

- When the calling object is the browser, the entire page will be searched.
- When the calling object is an element, only the html of that element will be searched.

As an example, say a page lists a project’s team members by their role.

```
<span>Project Managers:</span>
<ul id="project_managers">
  <li>Alicia Mcbride</li>
  <li>Jake Woods</li>
</ul>
<span>Developers:</span>
<ul id="developers">
  <li>Jack Greer</li>
  <li>Cora Williamson</li>
  <li>Albert Gross</li>
</ul>
<span>Testers:</span>
<ul id="testers">
  <li>Owen Francis</li>
  <li>Luis Leonard</li>
</ul>
```

Consider a test that needs to retrieve the name of one of the testers. It is not possible to locate the tester li elements directly as there are no properties to differentiate them from the other roles. However, the ul element does have an identifying attribute - the id specifies the role being listed. In other words, the test needs to get the first li element within the ul element with id of “testers”. This is done by chaining element type methods that identify the unique ancestors.

```
browser.ul(:id => 'testers').li.text
#=> "Owen Francis"
```

In this code, Watir will look for an ul element with id “tester” anywhere within the browser. Then, only within the ul element found, Watir will look for the first li element.

3.2 Based on its decendants

There are several approaches that can be used to locate an element based on its decendants, which are the elements within the target element.

For example, consider locating the li elements in the following html. The li elements do not have any distinguishing attributes. However, they do have child link elements with unique attributes - the data-componentcode.

```
<ul>
  <li id="ctl00_EnabledComponentListItem">
    <a id="ctl00_Component" data-componentcode="ItemA" href="#">Item A</a>
    <a href="#">X</a>
  </li>
  <li id="ctl01_EnabledComponentListItem">
    <a id="ctl01_Component" data-componentcode="ItemB" href="X">Item B</a>
    <a href="#">X</a>
  </li>
  <li id="ctl02_EnabledComponentListItem">
    <a id="ctl02_Component" data-componentcode="ItemC" href="#">Item C</a>
    <a href="#">X</a>
  </li>
</ul>
```

Using find

One approach is to iterate through the li elements until one is found with specified link.

```
browser.lis(:id, /EnabledComponentListItem/).find do |li|
  li.a(:data_componentcode, 'ItemC').exists?
end
```

Navigate from descendant to parent

If the html structure is stable, you can locate the descendant link (using standard locators) and then traverse the DOM up to the parent (ie li element).

```
browser.a(:data_componentcode, 'ItemC').parent
```

Using xpath

Using xpath, it is possible to get the element directly.

```
browser.li(:xpath, "//li[.//a[@data-componentcode='ItemC']]")
```

3.3 Based on its siblings

Locating an element by its siblings is often seen when working with tables. For example, in the below table, you might need to find the colour of a specific creature. A user knows which colour belongs to the creature because they are in the same row. In terms of the html, the colour td element is a sibling of the creature td element since they share the same parent element (tr element).

Table:

Creature	Colour
Elf	Blue
Dragon	Green
Imp	Black

HTML:

```
<table>
  <tbody>
    <tr>
      <th>Creature</th>
      <th>Colour</th>
    </tr>
    <tr>
      <td>Elf</td>
      <td>Blue</td>
    </tr>
    <tr>
      <td>Dragon</td>
      <td>Green</td>
    </tr>
    <tr>
      <td>Imp</td>
      <td>Black</td>
    </tr>
  </tbody>
</table>
```

Sibling to parent to element

To find an element based on its sibling, the general strategy is:

1. Locate the unique sibling element.
2. Get the parent element using the *parent* method.
3. Locate the required element within the scope of the parent.

Applying this strategy to the table, the colour of the dragon can be obtained by:

```ruby
#Get the unique element
unique_element = browser.table.td(:text => 'Dragon')

#Get the parent element
parent_element = unique_element.parent

#Get the actual element
parent_element.td(:index => 1).text
#=> "Green"
```

Parent by descendent to element

When the unique element is a cousin (ie a descendent of the sibling element), it is easier to locate the parent based on its descendents. The reason being that it is less fragile - ie ensuring that the correct number of "parent" methods are called becomes more difficult.

```ruby
parent_row = browser.table.trs.find do |tr|
  tr.td(:text => 'Dragon').present?
end
parent_row.td(:index => 1).text
#=> "Green"
```

3.4 DOM Traversal

Most often elements are located by traversing down the DOM. However, there are also methods for when travelling up or horizontally is required.

Parent

The DOM can be traversed upwards to an element's parent.

In the html:

```ruby
<div class="parent">
  <span class="child">Target</span>
</div>
```

The parent element of the span tag can be retrieved by using the *parent* method:

```ruby
child = browser.span(:class => 'child')
parent = child.parent
parent.tag_name
#=> "div"
```

Previous/next sibling

The DOM can also be traversed horizontally between sibling elements. For example, the following page has 4 sibling elements in the body - 2 h1 and 2 ul elements.

```
<html>
  <body>
    <h1>Heading 1</h1>
    <ul>
      <li>Item 1a</li>
    </ul>
    <h1>Heading 2</h1>
    <ul>
      <li>Item 2a</li>
      <li>Item 2b</li>
    </ul>
  </body>
</html>
```

Watir-Webdriver

Xpath has a following-sibling axis that can be used to find the next sibling of a given Watir element. This example shows that the next sibling of "Heading 2" is the unordered list with 2 items.

```
h1 = browser.h1(:text => 'Heading 2')
ul = h1.element(:xpath => './following-sibling::*')
puts ul.lis.length
#=> 2
```

The preceding-sibling axis can find previous sibling of the Watir element. The [1] is added to get the adjacent sibling. Otherwise, you would get the first sibling in the tree, which is the "Heading 1" h1 element.

```
h1 = browser.h1(:text => 'Heading 2')
ul = h1.element(:xpath => './preceding-sibling::*[1]')
puts ul.lis.length
#=> 1
```

Note that the element type returned must match what you are looking for. In the above examples, we looked for any element type. You can replace the * and use a specific watir element to look for a specific type. For example, to get the preceding h1:

```
start_h1 = browser.h1(:text => 'Heading 2')
end_h1 = start_h1.h1(:xpath => './preceding-sibling::h1[1]')
puts end_h1.text
#=> "Heading 1"
```

Watir-Classic

For some reason, the same xpath solution produces a UnknownObjectException in Watir-Classic. As an alternative, you can use the nextSibling property of the OLE object.

```
h1 = browser.h1(:text => 'Heading 2')
ul = browser.ul(:ole_object => h1.document.nextSibling)
puts ul.lis.length
#=> 2
```

Similarly, you can use the previousSibling property to get the preceding element.

```
h1 = browser.h1(:text => 'Heading 2')
ul = browser.ul(:ole_object => h1.document.previousSibling)
puts ul.lis.length
#=> 1
```

4. Inspecting an Element

4.1 Text

The *text* method returns the visible text of an element.

Given the html:

```
<div>This is the text of the element.</div>
```

The text of the div element is:

```
browser.div.text
#=> "This is the text of the element."
```

Note that the text of an element includes:

- The text nodes of the element and
- The text nodes of the element's descendants.

The following span has its own text node, " is a Ruby gem.", as well as a descendant text node, "Watir".

```
<span><a href="watir.com">Watir</a> is a Ruby gem.</span>
```

The text of the span is the concatenation of the text nodes:

```
browser.span.text
#=> "Watir is a Ruby gem."
```

Also note that Watir will only include the text that is visible to the user. In the following div, the first span is visible to the user, while the second span is not.

```
<div>
  <span>visible text</span>
  <span style="display:none;">hidden text</span>
</div>
```

Therefore, text of the div will only be the first span:

```
browser.div.text
#=> "visible text"
```

4.2 Attribute value

Attributes provide additional information or behaviour for an element. They can be found within the element's start tag.

Watir provides 2 ways to get attribute values:

1. Using the method that returns the specific attribute's value.
2. Using the *attribute_value* method.

Standard attributes

For attributes that are defined in the HTML specification, there is a corresponding method that returns the attribute value. In most cases, the attribute name and the method name are exactly the same. However, Ruby naming conventions are applied in some situations.

Scenario	Naming Rule	Example Attribute	Corresponding Method
Single word attribute	Match attribute name	id	id
Multi-word attribute	Words separated by underscore	maxlength	max_length
Boolean attribute	End with question mark	disabled	disabled?
Data attribute	Dashes replaced by underscores	data-field	data_field
Aria attribute	Dashes replaced by underscores	aria-describedby	aria_describedby
Class attribute	Due to Ruby objects already having a *class* method, which returns the instance's class, the "class" attribute is a special case.	class	class_name

The following HTML has several attributes:

```
<div data-field="first_name">
  <label id="tp1-label" for="first">First Name:</label>
  <input type="text" id="first" maxlength="50" required="required"
    aria-labelledby="tp1-label" aria-describedby="tp1">
  <div id="tp1" class="tooltip" role="tooltip" hidden="hidden"
    aria-hidden="true">Your first name is a required</div>
</div>
```

The attribute methods can be used to get the values:

```
# Retrieve a single word attribute
browser.text_field.id
#=> "first"

# Retrieve a multi-word attribute
browser.text_field.max_length
#=> 50

# Retrieve a boolean attribute
browser.text_field.required?
#=> true
browser.text_field.disabled?
#=> false

# Retrieve a data attribute
browser.div.data_field
#=> "first_name"

# Retrieve an aria attribute
browser.input.aria_labelledby
#=> "tp1-label"

# Retrieve a class attribute
browser.div(id: 'tp1').class_name
#=> "tooltip"
```

Note that the object type returned by the method will depend on the attribute:

- Boolean attributes will return a TrueClass or FalseClass object.
- Numeric attributes will return a Fixnum object.
- The rest of the attributes will return a String object.

Custom attributes

There will be some attributes that do not have an associated method:

- Attributes that are not defined in the HTML specification.
- Standard attribute methods that are not yet implemented in Watir-Classic.

In these cases, the *attribute_value* method is required.

Given an element with a custom attribute:

```
<div myCustomAttribute="custom" id="div_id">text</div>
```

The value of the custom attribute can be obtained by passing the name of the attribute to the *attribute_value* method:

```
browser.div.attribute_value('myCustomAttribute')
#=> "custom"
```

The method can also be useful when dynamically retrieving attribute values. For example, the following collects the attribute values specified in an array.

```
attrs = ['id', 'myCustomAttribute']
values = attrs.map { |attr| browser.div.attribute_value(attr) }
#=> ["div_id", "custom"]
```

4.3 Computed style

These days elements are rarely styled with just inline styles. For example, a, inline style can be applied to an error message to make the text red:

```
<div style="color:red;">Error</div>
```

Styles are now often moved into classes so that they can be consistently applied across different elements and pages. For example, using a class, the element might become:

```
<div class="error_message">Error</div>
```

To check that the error message text is red, it is not sufficient to just check the style attribute. Instead, you need to check the computed style - ie the actual style applied by the browser.

For Watir-Webdriver, this can be checked by using the Element#style method and specifying a style property:

```
browser.div.style('color')
#=> "rgba(255, 0, 0, 1)"
```

Note that shorthand CSS properties (e.g. background, font, etc.) are not supported. The longhand properties should be used instead - eg background-color.

In Watir-Classic, the Element#style method will only return the value of the inline styles. The computed style can be retrieved by:

```
browser.div.document.currentStyle.color
#=> "red"
```

Note that depending on the browser, the formatting of the property may vary.

4.4 Existence and visibility

Watir has three methods to check whether an elements exists on the page:

- exists? – Returns true if the element exists (in the DOM).
- visible? - Returns true if the element is visible to the user.
- present? – Returns true if the element exists and is visible to the user.

These methods are called similar to:

```
browser.div.exists?
browser.div.visible?
browser.div.present?
```

Consider the following page, which has one div tag that is visible (ie displayed to the user) and one that is not.

```
<body>
  <div id="1" style="block:display;">This text is visible</div>
  <div id="2" style="block:none;">This text is not visible</div>
</body>
```

When each method (exists?, present?, visible?) is run for the different elements (displayed, not displayed, non-existent), the following results are seen:

Element	.exists?	.visible?	.present?
Displayed (browser.div(:id ⇒ '1'))	true	true	true
Not Displayed (browser.div(:id ⇒ '2'))	true	false	false
Non-Existent (browser.div(:id ⇒ '3'))	false	exception	false

Element#exists? checks if the element is in the DOM (or HTML). Element#visible? checks if the element can be seen by the user, but throws an exception if the element is not in the DOM. Element#present? is the same as Element#visible? except that it returns false, instead of an exception, when the element is not in the DOM.

While the three methods are definitely different, I think, at least from my experience, that you should typically be using Element#present?.

One point to keep in mind, especially for those working with legacy watir libraries/tests, is that .exists? can lead to false positives. Imagine you have a form. When the form is submitted, an error message is displayed for fields that fail validation (ex missing a required field). In some applications this will be implemented by simply changing the style of the element containing the error message from display:none to display:block. Now consider writing a test to check that the validation message is displayed. If you use .exists?, the test will have false positives (ie it will always pass since the element is always in the DOM). In this situation, .visible? or present? should have been used.

5. Forms

5.1 Text Field

Input

To simulate a user typing into a text field, use the *set* method:

```
browser.text_field.set("first")
p browser.text_field.value
#=> "first"
```

Note that the *set* method will overwrite the previous text:

```
browser.text_field.set("first")
browser.text_field.set("second")
p browser.text_field.value
#=> "second"
```

If you want to add additional text, use the *append* method instead:

```
browser.text_field.set("first")
browser.text_field.append("second")
p browser.text_field.value
#=> "firstsecond"
```

To make the text field blank, use the *clear* method:

```
browser.text_field.clear
p browser.text_field.value
#=> ""
```

Inspect

The text that a user sees in a text field is stored in the input's value attribute. Therefore, to get the text of the text field, use the standard attribute methods:

```
browser.text_field.set('some text')
p browser.text_field.value
#=> "some text"
```

Textarea

While textareas can be located differently than text fields, they are set and inspected using the same methods. What makes textareas special on web pages is that they can have multi-line text. A real user would create multiple lines by pressing the Enter key on the keyboard. With Watir, you can do the exact same keystrokes:

```
browser.textarea.set('first line', :return, 'second line')
puts browser.textarea.value
#=> first line
#=> second line
```

However, it is often more convenient to send a double-quoted String with the line break character ("\n"):

```
browser.textarea.set("first line\nsecond line")
puts browser.textarea.value
#=> first line
#=> second line
```

5.2 Checkbox

Input

A checkbox is checked by using the *set* method:

```
browser.checkbox.set
```

It can be unchecked by using the *clear* method:

```
browser.checkbox.clear
```

The *set* method can take a boolean parameter, where true will check the box and false will uncheck the box:

```
# Check
browser.checkbox.set(true)

# Uncheck
browser.checkbox.set(false)
```

This is useful when the checkbox needs to be set based on a condition. For example, an if-else statement:

```
if condition_met?
  browser.checkbox.set
else
  browser.checkbox.clear
end
```

Can be re-written as:

```
browser.checkbox.set(condition_met?)
```

Inspect

The state of the checkbox, whether it is checked or unchecked, can be determined by the *set?* method:

```
browser.checkbox.set
p browser.checkbox.set?
#=> true

browser.checkbox.clear
p browser.checkbox.set?
#=> false
```

5.3 Radio

Input

A radio button is selected by using the *set* method:

```
browser.radio.set
```

Inspect

The *set?* method is used to determine if a radio button is turned on:

```
p browser.radio.set?
#=> false

browser.radio.set

p browser.radio.set?
#=> true
```

6. Element Collections

6.1 Counting

Given an element collection, you can check how many elements were found by using the *size* method.

```
<html>
  <body>
    <div class="title">Title</div>
    <div class="content">Content A</div>
    <div class="content">Content B</div>
  </body>
</html>
```

There are 3 div elements on the page.

```
browser.divs.size
#=> 3
```

With 2 of the div elements having the class “content”.

```
browser.divs(:class => 'content').size
#=> 2
```

Depending on your preferences, you can also use the method *length* instead of *size*.

```
browser.divs.length
#=> 3
```

7. Troubleshooting

7.1 UnknownObjectException - Unable to Locate Element

Have you tried to interact with an element, only to get an exception like:

```
Watir::Exception::UnknownObjectException:
  Unable to locate element, using {:tag_name=>["div"]}
  from ../lib/watir-classic/element.rb:66:in `assert_exists'
  from ../lib/watir-classic/element.rb:125:in `text'
  from (irb):5
  from C:/Ruby193/bin/irb:12:in `<main>'
```

Watir is saying that it cannot find the element. Yet when you manually go and look at the page, you can see the element! Why can't Watir locate it?

There are a variety of possible reasons, which have been summarized in the table below.

Problem	Solution
Element is in a frame	You must explicitly tell Watir that the element is within a frame. `browser.frame.div.text`
Element has not finished loading	Add an explicit wait. `browser.div.when_present.text`

Element locator uses dynamic attribute values	Use a regex to only match the stable part of the attribute.
	
	The following locator includes random values (bad):
	`browser.div(:id, 'static_rand5').text`
	
	Change the locator to a regex and just match the static part:
	`browser.div(:id, /static/).text`

Element is in a popup window	You must explicitly tell Watir to use the popup window.
	`b.window(:title => /window title/) do`
	` b.div(:id => 'div_in_popup').text`
	`end`

Element type is incorrect	Verify that you are trying to find the right element type.

Problem	Solution

	For example, what looks like a button might actually be a link.
	
	Results in an UnknownObjectException:
	`browser.button.click`
	
	This will work:
	`browser.link.click`