Parsing and traversing a Document

To parse a HTML document:

```
String html = "<html><head><title>First parse</title></head>"
  + "<body><p>Parsed HTML into a doc.</p></body></html>";
Document doc = Jsoup.parse(html);
```

(See parsing a document from a string for more info.)

The parser will make every attempt to create a clean parse from the HTML you provide, regardless of whether the HTML is well-formed or not. It handles:

- unclosed tags (e.g. `<p>Lorem <p>Ipsum` parses to `<p>Lorem</p> <p>Ipsum</p>`)
- implicit tags (e.g. a naked `<td>Table data</td>` is wrapped into a `<table><tr><td>?`)
- reliably creating the document structure (`html` containing a `head` and `body`, and only appropriate elements within the head)

The object model of a document

- Documents consist of Elements and TextNodes (and a couple of other misc nodes: see thenodes package tree).
- The inheritance chain is: `Document` extends `Element` extends `Node`. `TextNode` extends`Node`.
- An Element contains a list of children Nodes, and has one parent Element. They also have provide a filtered list of child Elements only.

Use DOM methods to navigate a document

Problem

You have a HTML document that you want to extract data from. You know generally the structure of the HTML document.

Solution

Use the DOM-like methods available after parsing HTML into a `Document`.

```
File input = new File("/tmp/input.html");
Document doc = Jsoup.parse(input, "UTF-8", "http://example.com/");

Element content = doc.getElementById("content");
Elements links = content.getElementsByTag("a");
for (Element link : links) {
  String linkHref = link.attr("href");
```

```
  String linkText = link.text();
}
```

Description

Elements provide a range of DOM-like methods to find elements, and extract and manipulate their data. The DOM getters are contextual: called on a parent Document they find matching elements under the document; called on a child element they find elements under that child. In this way you can winnow in on the data you want.

Finding elements

- `getElementById(String id)`
- `getElementsByTag(String tag)`
- `getElementsByClass(String className)`
- `getElementsByAttribute(String key)` (and related methods)
- Element siblings: `siblingElements()`, `firstElementSibling()`,`lastElementSibling()`; `nextElementSibling()`, `previousElementSibling()`
- Graph: `parent()`, `children()`, `child(int index)`

Element data

- `attr(String key)` to get and `attr(String key, String value)` to set attributes
- `attributes()` to get all attributes
- `id()`, `className()` and `classNames()`
- `text()` to get and `text(String value)` to set the text content
- `html()` to get and `html(String value)` to set the inner HTML content
- `outerHtml()` to get the outer HTML value
- `data()` to get data content (e.g. of `script` and `style` tags)
- `tag()` and `tagName()`

Manipulating HTML and text

- `append(String html)`, `prepend(String html)`
- `appendText(String text)`, `prependText(String text)`
- `appendElement(String tagName)`, `prependElement(String tagName)`
- `html(String value)`

Cookbook contents

Introduction

1. Parsing and traversing a Document

Input

Use selector-syntax to find elements

Problem

You want to find or manipulate elements using a CSS or jquery-like selector syntax.

Solution

Use the `Element.select(String selector)` and `Elements.select(String selector)` methods:

```
File input = new File("/tmp/input.html");
Document doc = Jsoup.parse(input, "UTF-8", "http://example.com/");

Elements links = doc.select("a[href]"); // a with href
Elements pngs = doc.select("img[src$=.png]");
  // img with src ending .png

Element masthead = doc.select("div.masthead").first();
  // div with class=masthead

Elements resultLinks = doc.select("h3.r > a"); // direct a after h3
```

Description

jsoup elements support a CSS (or jquery) like selector syntax to find matching elements, that allows very powerful and robust queries.
The `select` method is available in a `Document`, `Element`, or in `Elements`. It is contextual, so you can filter by selecting from a specific element, or by chaining select calls.
Select returns a list of Elements (as `Elements`), which provides a range of methods to extract and manipulate the results.

Selector overview

- `tagname`: find elements by tag, e.g. `a`
- `ns|tag`: find elements by tag in a namespace, e.g. `fb|name` finds `<fb:name>` elements
- `#id`: find elements by ID, e.g. `#logo`
- `.class`: find elements by class name, e.g. `.masthead`
- `[attribute]`: elements with attribute, e.g. `[href]`
- `[^attr]`: elements with an attribute name prefix, e.g. `[^data-]` finds elements with HTML5 dataset attributes
- `[attr=value]`: elements with attribute value, e.g. `[width=500]`
- `[attr^=value]`, `[attr$=value]`, `[attr*=value]`: elements with attributes that start with, end with, or contain the value, e.g. `[href*=/path/]`
- `[attr~=regex]`: elements with attribute values that match the regular expression; e.g.`img[src~=(?i)\.(png|jpe?g)]`
- `*`: all elements, e.g. `*`

Selector combinations

- `el#id`: elements with ID, e.g. `div#logo`
- `el.class`: elements with class, e.g. `div.masthead`
- `el[attr]`: elements with attribute, e.g. `a[href]`
- Any combination, e.g. `a[href].highlight`
- `ancestor child`: child elements that descend from ancestor, e.g. `.body p` finds `p`elements anywhere under a block with class "body"
- `parent > child`: child elements that descend directly from parent, e.g. `div.content > p` finds `p` elements; and `body > *` finds the direct children of the body tag
- `siblingA + siblingB`: finds sibling B element immediately preceded by sibling A, e.g.`div.head + div`
- `siblingA ~ siblingX`: finds sibling X element preceded by sibling A, e.g. `h1 ~ p`
- `el, el, el`: group multiple selectors, find unique elements that match any of the selectors; e.g. `div.masthead, div.logo`

Pseudo selectors

- `:lt(n)`: find elements whose sibling index (i.e. its position in the DOM tree relative to its

parent) is less than `n`; e.g. `td:lt(3)`
- `:gt(n)`: find elements whose sibling index is greater than `n`; e.g. `div p:gt(2)`
- `:eq(n)`: find elements whose sibling index is equal to `n`; e.g. `form input:eq(1)`
- `:has(seletor)`: find elements that contain elements matching the selector; e.g.`div:has(p)`
- `:not(selector)`: find elements that do not match the selector; e.g. `div:not(.logo)`
- `:contains(text)`: find elements that contain the given text. The search is case-insensitive; e.g. `p:contains(jsoup)`
- `:containsOwn(text)`: find elements that directly contain the given text
- `:matches(regex)`: find elements whose text matches the specified regular expression; e.g. `div:matches((?i)login)`
- `:matchesOwn(regex)`: find elements whose own text matches the specified regular expression
- Note that the above indexed pseudo-selectors are 0-based, that is, the first element is at index 0, the second at 1, etc

See the `Selector` API reference for the full supported list and details.