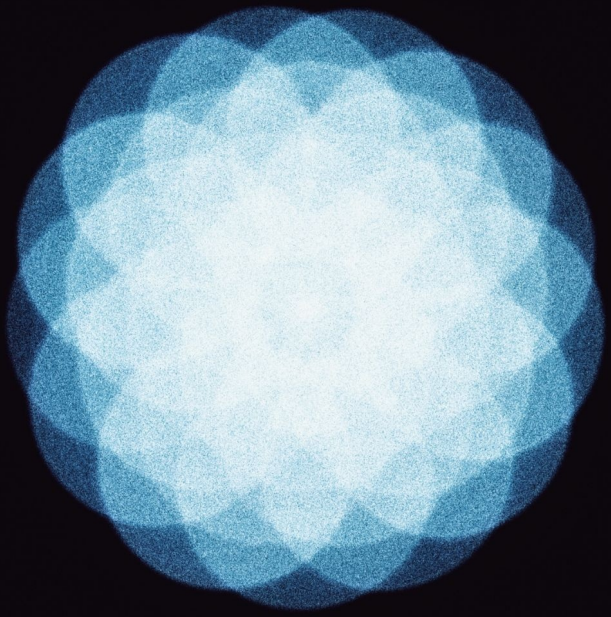
Искусство Автоматизации Тестирования с Selenide



Яков Крамаренко

Содержание

Вступление

Исскуство Автоматизации

Исскуство Автоматизации Тестирования с Selenide

Яков Крамаренко

Эта книга - практическое руководство по искусству автоматизации тестирования для начинающих ее изучение с самого нуля а также опытных автоматизаторов. Большое внимание уделено теме рефакторинга, написанию чистых, простых, читабельных и легких в поддержке тестов. Она раскрывает тему автоматизации пользовательского интерфейса веб-приложений с использованием следующих инструментов: Java, Selenide, Selenium WebDriver, JUnit5, Allure Reporting. Книга построена в виде сборника практических заданий и упражнений по Методу Кейсов.

Книгу можно купить по ссылке http://leanpub.com/selenide-automation-ru.

Автор фото на обложке - Dan Hodgkins.

Эта версия была опубликована 2019-08-19.

Предисловие

В 2015 году я начал обучать на платных "офлайн" и онлайн ИТ-курсах (по программированию, автоматизации тестирования, и т.д.). Сначала я относился к этому как к временному заработку, совсем маленькому, но заработку за счет занятия делом, которое мне всегда приносило удовольствие - делиться опытом с другими, при этом структурируя знания и развиваясь самому. Со временем это занятие обросло четкой концепцией создания учебной программы и материалов, позволяющих начинающим ИТ-специалистам в быстрые строки обучаться практическим навыкам, и главное - обучать обучаться. Помня и ценя свой собственный опыт само-обучения, я строил программу на основе практических заданий с минимумом теории. Я старался не приподнести все секреты на блюдечке, наоборот, - предоставить возможность набить шишки студентам самим, но сделать это в ускоренном режиме, пройдя по специально разработанному маршруту, где будут встречаться реальные рабочие проблемы, которые я собирал годами своего опыта в IT.

Сфера обучения также меня интересовала, и продолжает интересовать - как источник специалистов для моих проектов. Я заметил, что надежней находить начинающих способных инженеров и обучать их нужным навыкам, чем переучивать "старичков", уровень знаний которых часто несоизмеримо мал по сравнению с их эгом:)

Со временем я заметил, что обычный формат курса с преподавателями и менторами - сильно тяжел в поддержке, и сложнее масштабируется. Так и появилась идея перевести его в формат книги.

В этой книге учащемуся предлагается пройти полный путь построения автоматизации небольшого веб приложения с помощью решения серии заданий, которые так или иначе ждут его и в реальном проекте. При этом перед началом работы над каждым заданием дается минимум теоретических знаний, которые могут быть доступны либо в самой книге, либо по ссылкам на другие публичные ресурсы. Если знаний уже должно быть достаточно - урок с теорией может быть упущен совсем.

Далее, в процессе работы над заданием, встречаясь с затруднениями и проблемами, если "гугл не помог", студент может подсматривать в следующий за заданием раздел с решением. Раздел может содержать детальное описание процесса решения задачи, или список с частыми вопросами и ответами, или список с частыми ошибками и их решениями. После самостоятельного выполнения задания, учащийся сможет окончательно себя проверить по этому же разделу.

На данный момент книга все еще находится в разработке. Поддержать которую можно купив книгу по рекоммендуемой цене на этом сайте. Новые главы будут выходить со временем, без четких пределов по срокам. Но есть желание закончить основную часть до конца 2019 года. Главное и самое полезное содержание книги - это списки частых ошибок и их решений. Именно струкутурирование этой информации занимает самое большое время. Пока книга наполняется этой информацией, эта же програма доступна в виде онлайн-курса, где решения проверяются в стиле код-ревью мной и менторами. Записаться на курс можно обратившись по почте automician@gmail.com.

Программа и статус готовых разделов:

- 00 Введение в Selenide
- [TODO] 01 Начало. Проверка Концепции
- [TODO] 02 Пересмотр. Рефакторинг Теста - Часть 1
- [TODO] 03 Закалка. Расширение покрытия. Атомарные тесты
- [TODO] 04 Структура и переиспользование. Рефакторинг Тестов - Часть 2 (PageObjects)
- [TODO] 05 Отчетность
- [TODO] 06 Тест-сьюты
- [TODO] 07 Параметризация
- [TODO] 08 Масштабирование. Грид
- [TODO] 09 Интеграция. CI
- [TODO] 10 Оптимизация. Параллелизация
- [TODO] A1 Обзор Selenium Webdriver в сравнении с Selenide

В процессе написания книги, эта программа может изменятся.

К книге прилагаются рабочие образцы кода из теоретических разделов а также решения соответствующих заданий. Некоторые главы могут сопровождаться видео, выложенном в публичный доступ или доступном среди дополнительных материалов прилагаемых к книге.

Пока книга находится в разработке, будь готов к присутствию "багов", опечаток и неточностей. Список известных:

- не работают ссылки в содержании для разделов: "Задание: Selenide и XPATH/ Решение/*"

Как работать с этой книгой

Эта книга-курс рассчитана, в первую очередь, на самостоятельное обучение. Для ее "прохождения" нужны определенные базовые навыки. Полный список таких навыков указан в разделе Необходимые знания перед стартом. В нем же можно найти публичные материалы, рекомендуемые к освоению.

Книга состоит из разделов, каждый из которых может состоять из следующего типа глав:

- урок
- задание
    - решение

При наличии урока, ты можешь ознакомиться с самыми базовыми знаниями темы. Желательно повторить самому примеры показанные в нем.

Далее ты приступаешь к заданию, которое следует попробовать решить полностью самомстоятельно, обращаясь за помощью, в первую очередь - к google.com. Будь окуратен, ты можешь нагуглить и уже готовые решения других "студентов", выложенные где то на github.com - в них не подглядывай;). Если быстрый поиск не помог, тогда можно заглянуть в следующую главу с решением.

Глава с решением позволит либо полностью разобрать процесс работы над заданием, либо подсказать как решить очередную проблему. Либо после самостоятельно законченного задания - проверить себя по частым ошибкам и ответам. Обычно ошибки в начале - будут содержать подсказки а не ответы. Рекомендуется попробовать сначала сообразить решение по подсказкам, и только в самом конце - проверить правильный результат.

Удачи!

Необходимые знания перед стартом

Перед тем, как начать работу над упражнениями из книги, ты должен иметь следующую базу:

- HTML, CSS и JavaScript (самые основы)
  - в интернете полно бесплатных интерактивных туториалов, просто загугли;). Вот всего лишь несколько примеров:
    - Intro to Software Development [HTML, CSS, JavaScript]
    - Hexlet.io: Введение в программирование (JavaScript)
      - Как альтернатива: "CS50. Основы программирования" от javarush.ru
    - Hexlet.io: HTML & CSS
- Java
  - Гугл снова поможет;) Но вот несколько примеров ресурсов:
    - code-basics: java
    - LearnXinYminutes: Java
      - на русском
    - learnjavaonline.org
    - java koans

В целом, в некоторых случаях, даже прохождение половины материалов, упомянутых выше, - может быть достаточно, если ты начинаешь обучение автоматизации тестирования с самого нуля. Для автоматизации тестирования не нужно знать все особенности языка программирования. Вначале, автоматизация вообще может показаться довольно простой. И ты заметишь это по тому, как еще недоучив язык, сможешь пройти пол этой книги;) Обычно, для написания автоматизированных тестов, не придется использовать даже простые алгоритмы. Все сводится просто к вызову команд и структурированию кода. Но рано или поздно сложные задачи все равно появятся. Поэтому, хотя в этой книге сложных алгоритмических упражнений не будет, желательно сразу паралельно "набивать руку" в программировании более сложных алгоритмических задач, чтобы не обмануться простотой "первых проб в автоматизации". Следующие ресурсы могут в этом помочь:

- exercism.io
- replit
- CodinGame
- codewars

Также, тебе будет более удобно работать над заданиями книги, если ты будешь сохранять версии своих решений с помощью систем контроля версий. Я рекомендую использовать git и один из самых популярных серверов - github.com. Вот и бесплатных гайд по теме: Github - Hello World.

Введение в Selenide

Быстрый Старт. Исходные условия, зависимости, первые шаги и документация. [Видео]

Selenide - это инструмент для автоматизации действий пользователя в браузере, ориентированный на удобство и легкость реализации бизнес-логики в автотестах, на языке пользователя, не отвлекаясь на технические детали работы с "драйвером браузера".

Например, к техническим деталям - можно отнести работу с ожиданиями элементов при автоматизации тестирования динамических веб-приложений, реализацию высокоуровневых действий над элементами, и так далее.

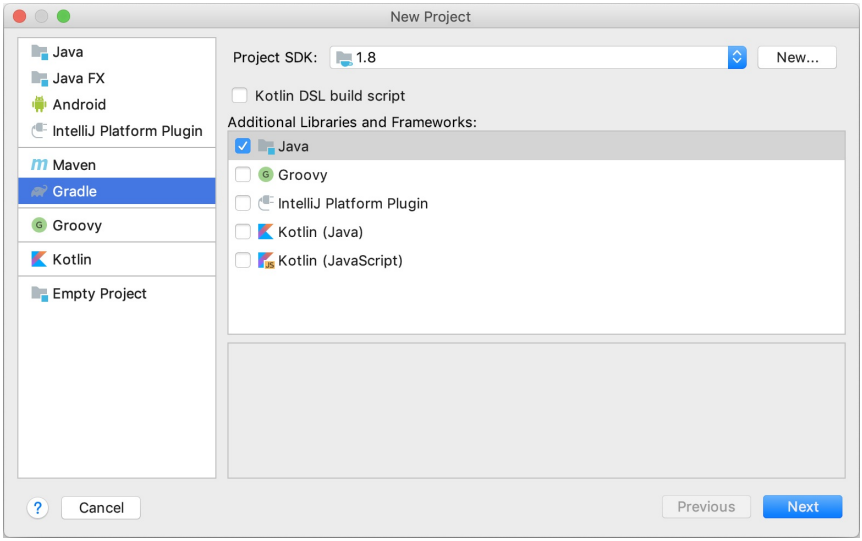
Давай с ним очень быстро познакомимся.

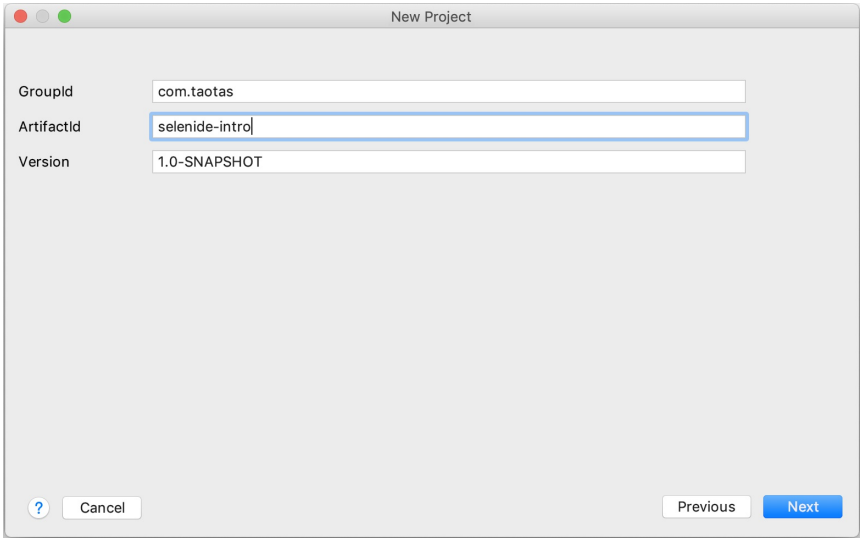
Итак, имея установленными следующие инструменты (можешь загуглить как - сам):

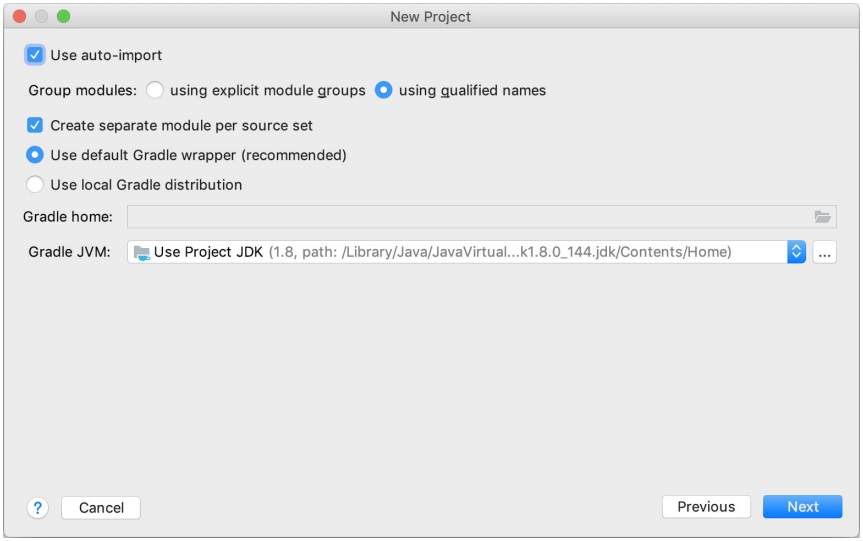
- JDK8 (Java Development Kit)
- Chrome Browser
- ChromeDriver
- Intellij Idea

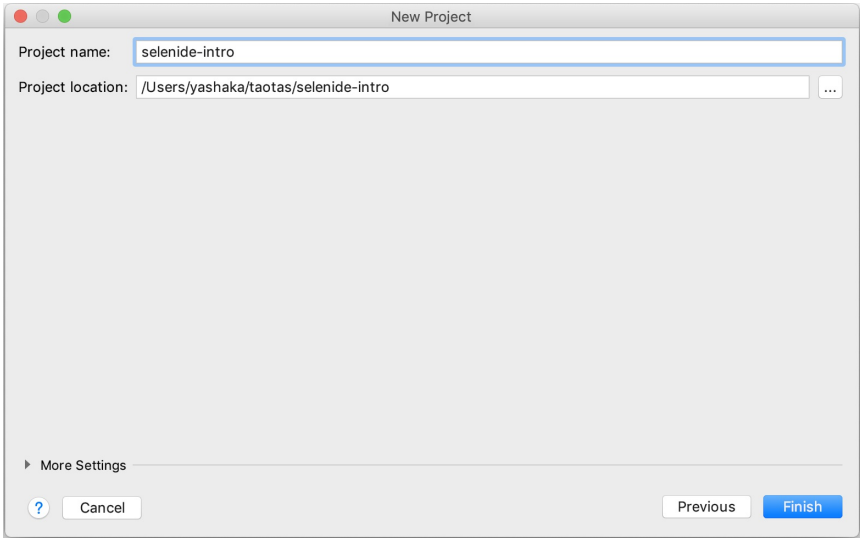
Создадим классический Gradle-проект в Intellij Idea...

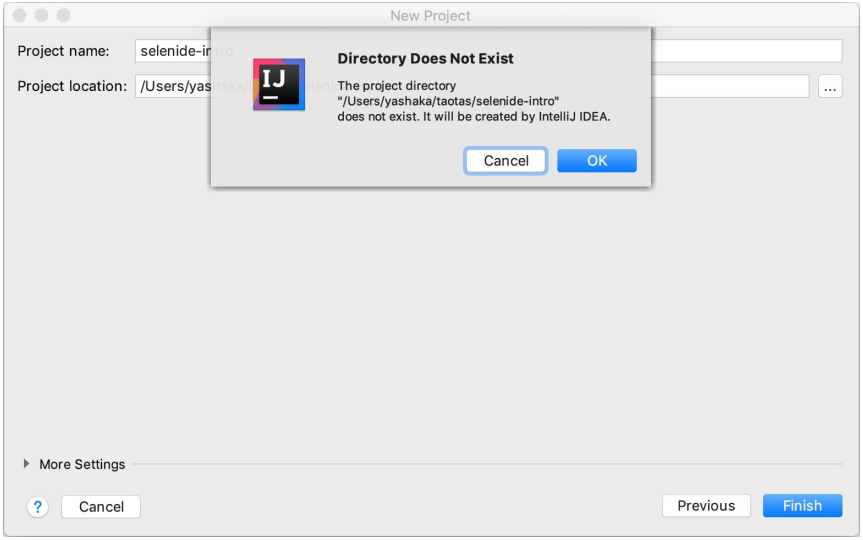

Welcome to IntelliJ IDEA
IntelliJ IDEA
Version 2019.1.3
Create New Project
Import Project
Open
Check out from Version Control
Configure
Get Help

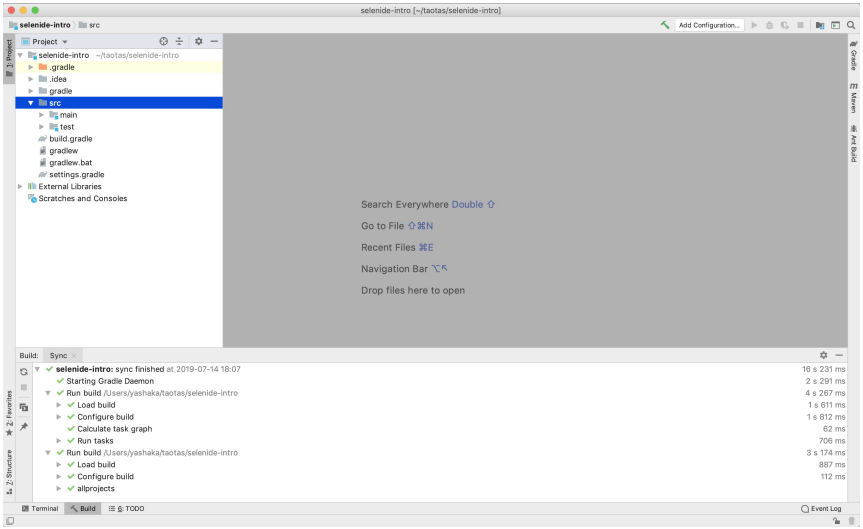

New Project
Java
Java FX
Android
IntelliJ Platform Plugin
Maven
Gradle
Groovy
Kotlin
Empty Project
Project SDK:
1.8
New...
Kotlin DSL build script
Additional Libraries and Frameworks:
Java
Groovy
IntelliJ Platform Plugin
Kotlin (Java)
Kotlin (JavaScript)
Cancel
Previous
Next


New Project
GroupId
com.taotas
ArtifactId
selenide-intro
Version
1.0-SNAPSHOT
Cancel
Previous
Next


New Project
Use auto-import
Group modules:
using explicit module groups
using qualified names
Create separate module per source set
Use default Gradle wrapper (recommended)
Use local Gradle distribution
Gradle home:
Gradle JVM:
Use Project JDK (1.8, path: /Library/Java/JavaVirtual...k1.8.0_144.jdk/Contents/Home)
Cancel
Previous
Next


New Project
Project name:
selenide-intro
Project location:
/Users/yashaka/taotas/selenide-intro
...
More Settings
Cancel
Previous
Finish


New Project
Project name:
Project location:
Directory Does Not Exist
The project directory
"/Users/yashaka/taotas/selenide-intro"
does not exist. It will be created by IntelliJ IDEA.
Cancel
OK
More Settings
Cancel
Previous
Finish



Структура проекта получилась довольно простая:

```
selenide-intro/
  gradle
    wrapper
      gradle-wrapper.jar
      gradle-wrapper.properties
  src
    main
    test
  build.gradle
  gradlew
  gradlew.bat
  settings.gradle
```

Тут...

`selenide-intro` - папка с проектом. `gradle-wrapper.jar` - испольнимый jar-файл с кодом градл-враппера для загрузки нужной версии градла `gradle-wrapper.properties` - настройки градл-враппера `build.gradle` - скрипт

конфигурации сборки проекта ("скрипт сборки") `settings.gradle` - скрипт настроек для скрипта сборки `gradlew` - скрипт для сборки проекта под операционными системамии симейства Unix `gradlew.bat` - скрипт для сборки проекта под Windows

Нас инересует в первую очередь `build.gradle`. Следующие конфигурации сборки проекта были сгенерированы автоматически при создании проекта, и значения их очевидны и говорят сами за себя:

```
plugins {
    id 'java'
}

group 'com.taotas'
version '1.0-SNAPSHOT'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

В зависимостях...

```
//...

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

уже подключена библиотека JUnit, которую мы и будем использовать для организации тестов и их запуска. Вот только версия JUnit не самая последняя. Чтобы использовать последний JUnit5 прийдется сделать некоторые махинации...

Подключить нужные версии библиотек (JUnit5 состоит из нескольких частей):

```
//...

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-engine:5.5.0'
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.5.0'
}
```

И указать градлу что мы будем использовать новую платформу JUnit для запуска тестов:

```
//...

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-engine:5.5.0'
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.5.0'
}

test {
    useJUnitPlatform()
}
```

Теперь, все что нужно сделать, чтобы начать работать с Selenide - это добавить и его в зависимости проекта:

```
//...

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-engine:5.5.0'
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.5.0'
    testImplementation 'com.codeborne:selenide:5.2.4'
}
```

Теперь, чтобы использовать всю прелесть Selenide, в коде достаточно просто сделать пару импортов в стиле:

```
import static com.codeborne.selenide.Selenide.*;
import static com.codeborne.selenide.Selectors.*;
import static com.codeborne.selenide.Condition.*;
```

```
import static com.codeborne.selenide.CollectionCondition.*;
//...
```

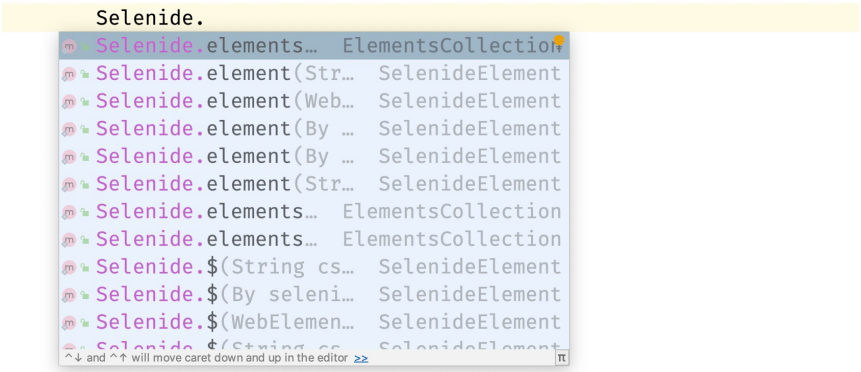
Самые-самые базовые команды Selenide (из класса `Selenide.*`) - это:

- `open` - загружает страницу по URL (и открывает браузер автоматически, если он еще не открыт)
- `element` (или `$`) - ищет элемент по селектору (css или специальному селектору типа `By`)
- `elements` (или `$$`) - ищет коллекцию элементов по селектору (css или специальному селектору типа `By`)

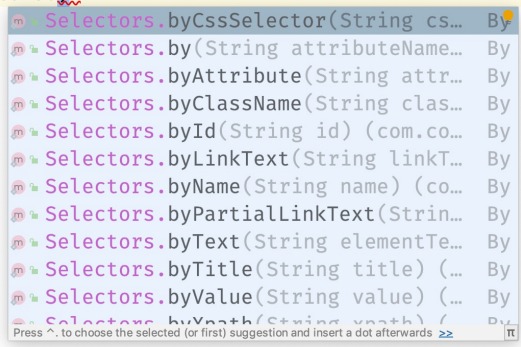
Специализированные селекторы, такие как поиск по аттрибутам (например `byName`) или по тексту (`byText`) или по XPath (`byXpath`) живут в классе `Selectors.*`.

В класах `Condition.*` и `CollectionCondition.*` живут условия для проверки элементов с помощью комманд `should*`, например `exactText` - для проверки точного текста у элемента.

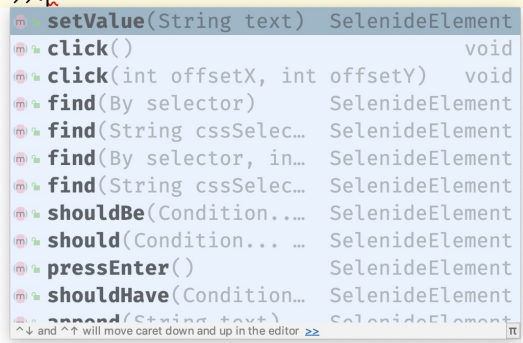
В принципе, этого достаточно, чтобы начать работу более опытному специалисту. Все остальные команды будут подсказаны любимым IDE. После ввода точки в коде, IDE подскажет все допустимые команды над объектом.



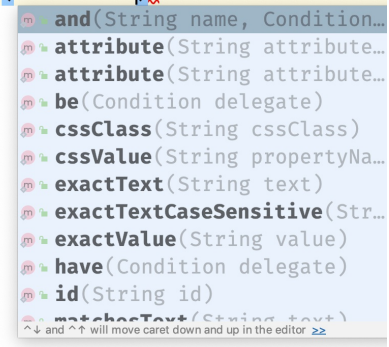
```
element(Selectors.)
```



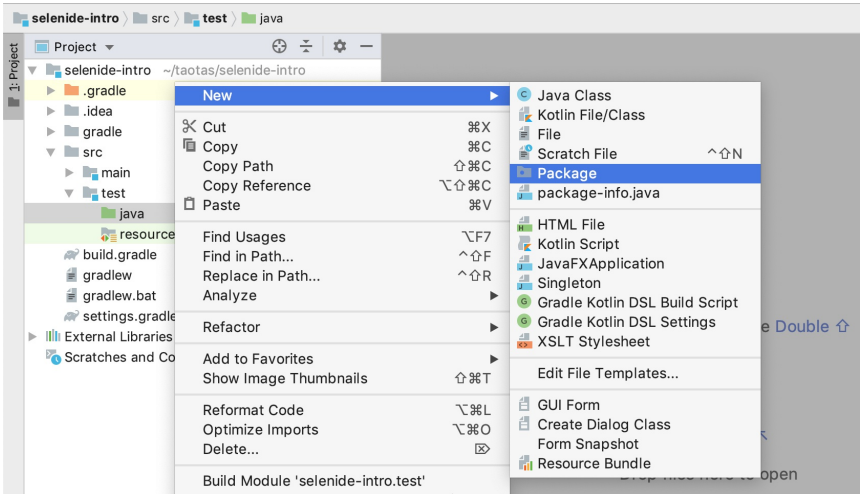

```
element(byName("q")).
```

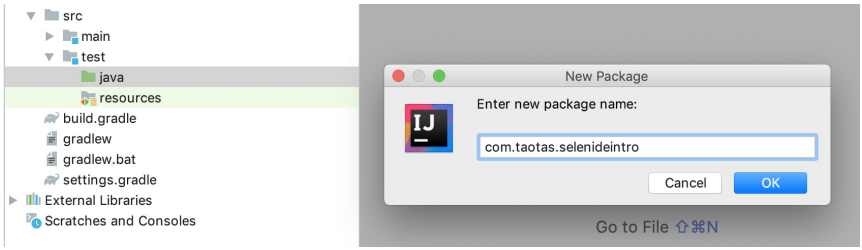


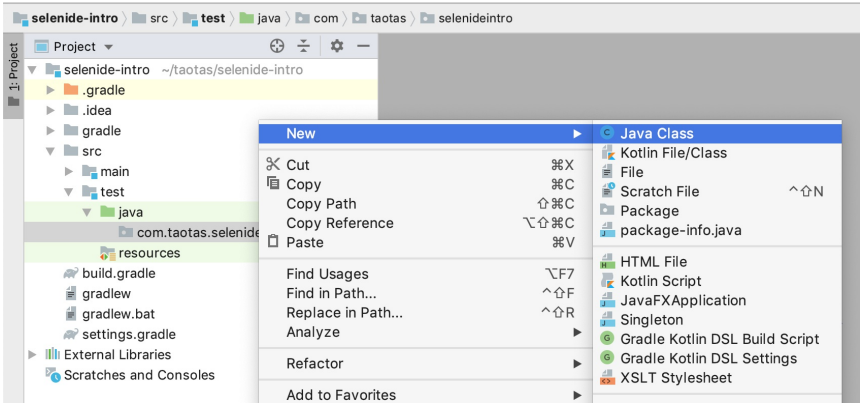

```
element(byName("q")).shouldHave(Condition.)
```

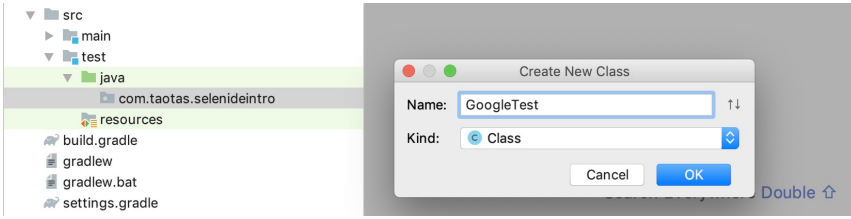


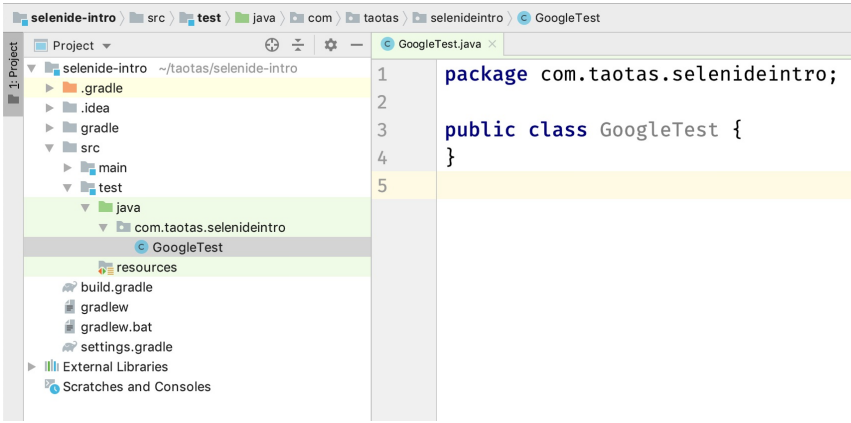
В таком вот иследовательском режиме можно и гугл успеть протестировать ;)











...

```
//selenide-intro/src/test/java/com/taotas/selenideintro/GoogleTest.java
package com.taotas.lselenideintro;
import static com.codeborne.selenide.Selenide.*;
import static com.codeborne.selenide.Selectors.*;
import static com.codeborne.selenide.Condition.*;
import static com.codeborne.selenide.CollectionCondition.*;
import static org.openqa.selenium.support.ui.ExpectedConditions.titleIs;

import org.junit.jupiter.api.Test;

public class GoogleTest {

    @Test
    void shouldSearch() {
        open("https://google.com/ncr");

        element(byName("q")).setValue("selenide").pressEnter();
```

```
        elements("#search .g").shouldHave(sizeGreaterThanOrEqual(6))
            .first().shouldHave(text("Selenide: concise UI tests in Java"))
            .find(".r>a").click();

        Wait().until(titleIs("Selenide: concise UI tests in Java"));
    }
}
```

Или чуть более многословно, но возможно более понятно:

```
//selenide-intro/src/test/java/com/taotas/selenideintro/GoogleTest.java
package com.taotas.lselenideintro;
import static com.codeborne.selenide.Selenide.*;
import static com.codeborne.selenide.Selectors.*;
import static com.codeborne.selenide.Condition.*;
import static com.codeborne.selenide.CollectionCondition.*;
import static org.openqa.selenium.support.ui.ExpectedConditions.titleIs;

import com.codeborne.selenide.ElementsCollection;
import org.junit.jupiter.api.Test;

public class GoogleTest {

    @Test
    void shouldSearch() {
        open("https://google.com/ncr");

        element(byName("q")).setValue("selenide").pressEnter();

        ElementsCollection results = elements("#search .g");
        results.shouldHave(sizeGreaterThanOrEqual(6))
            .first().shouldHave(text("Selenide: concise UI tests in Java"));
        results.first().find(".r>a").click();

        Wait().until(titleIs("Selenide: concise UI tests in Java"));
    }
}
```

Запуск прямо с Intellij Idea...

или с помощью команды `./gradlew build` с терминала ОС семейсва Unix (`gradlew.bat build` на Windows), с активной папкой проекта, - должен нам показать красивое кино в браузере Chrome;)

Если не хватает информации о том, что да как работает, всегда можно "провалиться" (`Cmd+Click` на Mac OS, `Ctrl+Click` на Windows) в код реализации нужных методов, почитать "javadoc", или просто с кодом разобраться.

Единственное, что может сбить с толку - это то, что искать реализацию команд-действий над элементами нужно не в `SelenideElement`, куда мы попадаем сразу после Cmd+Click (Ctrl+Click на Windows)...

(допустим нас интересует реализация `pressEnter()`)

```
public interface SelenideElement extends WebElement, WrapsDriver, WrapsElement
, Locatable, TakesScreenshot {

    //...
    SelenideElement pressEnter();
```

а в `com.codeborne.selenide.commands.*`, О чем, в принципе, и упоминается в javadoc комментарии:

```
  /**
   * Press ENTER. Useful for input field and textareas: <pre>
   *  $("query").val("Aikido techniques").pressEnter();</pre>
   *
   * Implementation details:
   * Check that element is displayed and execute <pre>
   *  WebElement.sendKeys(Keys.ENTER)</pre>
   *
   * @see com.codeborne.selenide.commands.PressEnter  <<ВОТ ЗДЕСЬ
   */
  SelenideElement pressEnter();


package com.codeborne.selenide.commands;

//...
```

```
public class PressEnter implements Command... {
  @Override
  public WebElement execute(SelenideElement proxy, WebElementSource locator, O
bject[] args) {
    locator.findAndAssertElementIsInteractable().sendKeys(Keys.ENTER);
    return proxy;
  }
}
```

Откуда, например, нам сразу становится понятно, что под капотом `pressEnter` живет обычный `sendKeys(Keys.ENTER)`, но совершенный после того, как мы убедимся в интерактивности (как минимум видимости) элемента ;)

Также можно узнать что там еще входит в API, провалившись в импортированные классы соответственно:

```
import static com.codeborne.selenide.Selenide.*;
import static com.codeborne.selenide.Selectors.*;
import static com.codeborne.selenide.Condition.*;
import static com.codeborne.selenide.CollectionCondition.*;
```

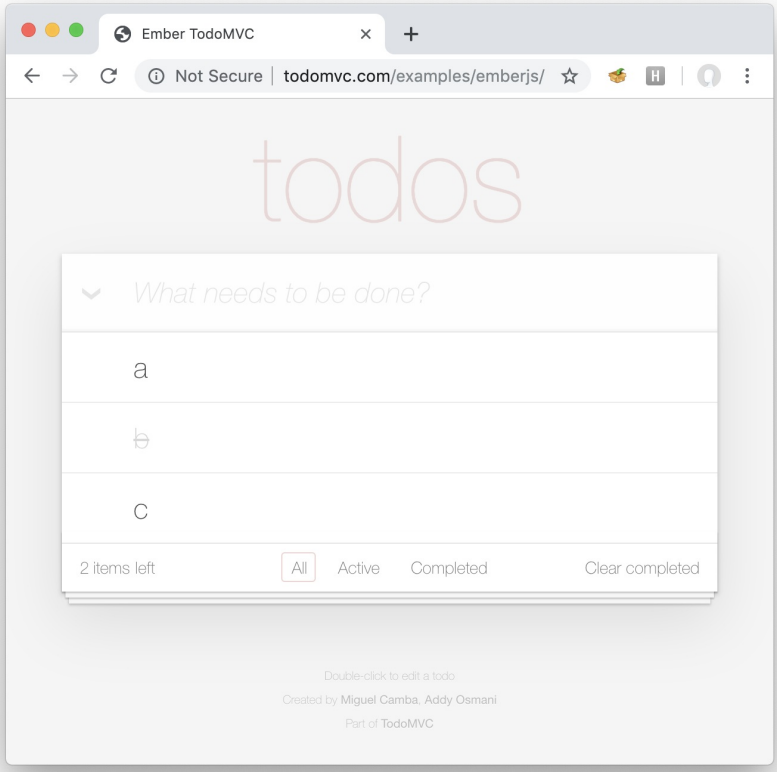
Если же путь истинного код-ниндзя, любящего порыться в дебрях кода библиотеки, тебе пока не близок, с использованием основных классов Selenide можно познакомиться в официальной документации. Она кратко описывает основной Selenide API. Обязательно прочти ее перед тем, как использовать Selenide в реальном проекте. Сама документация довольно краткая, но в ней есть также ссылки на более детальные объяснения в Selenide Gitbook. Гитбук повторяет информацию из документации, но с более глубокими объяснениями, как Selenide работает внутри, и как использовать эти знания чтобы еще более эффективно его использовать. Вначале можно гитбук пропустить, и вернуться к нему позже, когда будет больше опыта в написании тестов, или же начать "открывать секреты" уже сейчас, все зависит от того, насколько это тебе актуально именно в данный момент.

Selenide в действии. [Видео]

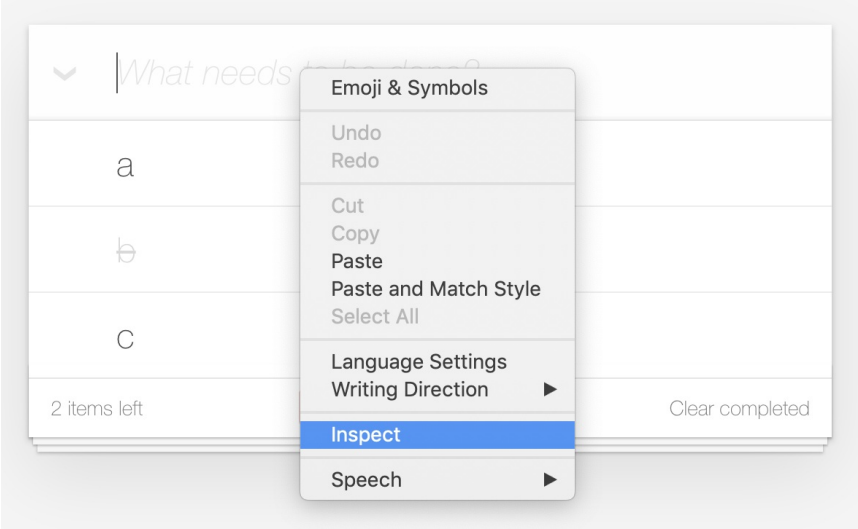
Давай теперь чуть более детально познакомимся с Selenide на примере реализации тест-сценария для приложения - менеджера задач - TodoMVC:

```
//selenide-intro/src/test/java/com/taotas/selenideintro/TodoMvcTest.java
package com.taotas.lselenideintro;

import org.junit.jupiter.api.Test;

public class TodoMvcTest {

    @Test
    void completesTask() {
        // open TodoMVC page

        // add tasks: "a", "b", "c"
        // tasks should be "a", "b", "c"

        // toggle b
        // completed tasks should be b
        // active tasks should be a, c
    }
}
```

Давай сразу поиграемся с этим приложением, попробуем вручную повторить этот сценарий...


Ember TodoMVC
Not Secure | todomvc.com/examples/emberjs/
todos
What needs to be done?
a
b
c
2 items left
All
Active
Completed
Clear completed
Double-click to edit a todo
Created by Miguel Camba, Addy Osmani
Part of TodoMVC

И сразу зароемся в структуру html для такого набора задач...



... обращая внимания на элементы с которыми мы взаимодействуем и игнорируя те элементы, которые нам пока не интересны:

```
<section id="todoapp">
  <header id="header">
    <!-- ... -->
    <input type="text" id="new-todo" placeholder="What needs to be done?" auto
focus="">
  </header>
  <section id="main" class="ember-view">
    <!-- ... -->
    <ul id="todo-list" class="todo-list">
      <li id="ember267" class="ember-view">
        <div class="view">
          <input type="checkbox" class="toggle">
          <label>a</label>
          <!-- ... -->
        </div>
        <!-- ... -->
      </li>
      <li id="ember317" class="completed ember-view">
```

```
        <div class="view">
          <input type="checkbox" class="toggle">
          <label>b</label>
          <!-- ... -->
        </div>
        <!-- ... -->
      </li>
      <li id="ember319" class="ember-view">
        <div class="view">
          <input type="checkbox" class="toggle">
          <label>c</label>
          <!-- ... -->
        </div>
        <!-- ... -->
      </li>
    </ul>
  </section>
  <footer id="footer">
    <!-- ... -->
    <ul id="filters">
      <li><a href="#/" id="ember275" class="selected ember-view">All</a></li>
      <li><a href="#/active" id="ember282" class="ember-view">Active</a></li>
      <li><a href="#/completed" id="ember298" class="ember-view">Completed</a>
</li>
    </ul>
    <!-- ... -->
  </footer>
</section>
```

Вот они, эти элементы, нужные для соответствующих действий нашего сценария:

- add "a", "b", "c"

```
      <input type="text" id="new-todo" placeholder="What needs to be done?" a
    utofocus="">
```

- tasks should be "a", "b", "c"

```
      <ul id="todo-list" class="todo-list">
        <li id="ember267" class="ember-view">
```

```
      <div class="view">
        <input type="checkbox" class="toggle">
        <label>a</label>
        <!-- ... -->
      </div>
      <!-- ... -->
    </li>
    <li id="ember317" class="completed ember-view">
      <div class="view">
        <input type="checkbox" class="toggle">
        <label>b</label>
        <!-- ... -->
      </div>
      <!-- ... -->
    </li>
    <li id="ember319" class="ember-view">
      <div class="view">
        <input type="checkbox" class="toggle">
        <label>c</label>
        <!-- ... -->
      </div>
      <!-- ... -->
    </li>
  </ul>
```

- toggle "b"

```
    <li id="ember317" class="completed ember-view">
      <div class="view">
        <input type="checkbox" class="toggle">    <!-- <<< -->
        <label>b</label>
```

- completed tasks should be b

```
    <li id="ember317" class="completed ember-view">
      <div class="view">
        <input type="checkbox" class="toggle">
        <label>b</label>
        <!-- ... -->
      </div>
```

```
        <!-- ... -->
```

- active tasks should be a, c

```
<ul id="todo-list" class="todo-list">
  <li id="ember267" class="ember-view">
    <div class="view">
      <input type="checkbox" class="toggle">
      <label>a</label>
      <!-- ... -->
    </div>
    <!-- ... -->
  </li>
  <li id="ember319" class="ember-view">
    <div class="view">
      <input type="checkbox" class="toggle">
      <label>c</label>
      <!-- ... -->
    </div>
    <!-- ... -->
  </li>
</ul>
```

Теперь, зная характеристики-атрибуты наших элементов, мы сможем их найти в коде и совершить нужные действия, знать бы только, чем их искать и как потом с ними взаимодействовать.

Ну вперед, начнем же, реализовывать наш сценарий:

```
@Test
void completesTask() {
    // open TodoMVC page

    // add tasks: "a", "b", "c"
    // tasks should be "a", "b", "c"

    // toggle b
    // completed tasks should be b
    // active tasks should be a, c
}
```

Для первого шага команда очевидна, нужно просто передать нужный URL как параметр в `open(...)`:

```
// open TodoMVC page
open("http://todomvc.com/examples/emberjs/");

// add tasks: "a", "b", "c"
// tasks should be "a", "b", "c"

// toggle b
// completed tasks should be b
// active tasks should be a, c
```

Для следующего шага, нам нужно вспомнить последовательность действий для добавления одной задачи:

```
//...

// add task "a":
//   1) find "new task" edit field 2) set value to "a" 3) press Enter
```

Теперь перевести это на "язык Selenide" не составит труда, используя подсказки IDE для поиска нужных нам команд-действий над элементом:

```
//...

// add task "a":
//   1) find "new task" edit field 2) set value to "a" 3) press Enter
element(byId("new-todo")).setValue("a").pressEnter();
```

Команду `element` мы используем, чтобы получить доступ к элементу на странице ...

```
<input type="text" id="new-todo" placeholder="What needs to be done?"
       autofocus="">
```

... по локатору, находящему элемент по уникальному идентификатору: `byId("new-todo")`

Можно использовать CSS селектор, напрямую передавая его команде `element`, и получить чуть более лаконичный код:

```
//...

// add task "a":
//   1) find "new task" edit field 2) set value to "a" 3) press Enter
element("#new-todo").setValue("a").pressEnter();
```

Еще более лаконично можно записуть ту же строку используя команду `$`:

```
//...

// add task "a":
//   1) find "new task" edit field 2) set value to "a" 3) press Enter
$("#new-todo").setValue("a").pressEnter();
```

Но давай пока ограничимся предыдущей версией, пускай менее лаконичной, но более читабельной для новичков:)

Теперь, можно добавить и другие задачи, используя очень полезный подход - "Copy & Paste Driven Development" ;)

```
//...

// add tasks "a", "b", "c"
element("#new-todo").setValue("a").pressEnter();
element("#new-todo").setValue("b").pressEnter();
element("#new-todo").setValue("c").pressEnter();

// tasks should be "a", "b", "c"
// ...
```

Один важный нюанс, который стоит иметь ввиду - команда `setValue` устанавливает соответствующее значениие в поле, в независимости от того что было введено ранее (старое значение будет очищено и заменено на новое). В нашем тесте, например, это приведет к тому, что мы не сможем отследить потенциальный баг о том, что после добавления задачи, поле введения текста новой задачи не очищается. Будем считать,

что в этом тесте это и не важно, потому что главная цель тестирования в нем - проверить завершение задач. Но на будущее - запомним этот нюанс и то, что команда которая не очищает раннее введеный текст, называется `append` ;)

Теперь найдем все задачи в списке с помощью команды `browser.all` и проверим, что они имеют соответствующие тексты:

```
//...

// add tasks "a", "b", "c"
element("#new-todo").setValue("a").pressEnter();
element("#new-todo").setValue("b").pressEnter();
element("#new-todo").setValue("c").pressEnter();

// tasks should be "a", "b", "c"
elements("#todo-list>li").shouldHave(exactTexts("a", "b", "c"));
```

Как видим, "язык Selenide" не очень отличается от "английского" ;) Проверки над элементами или коллекциями элементов, как в последнем случае, совершаются с помощью методов `should` / `shouldBe` / `shouldHave` , которым передается условие для проверки ("condition" - "кондишен"). В случае проверок над единичными элементами - используются свои "кондишены", живущие в классе `Condition` , а для проверок коллекций элементов - другие - живущие в классе `CollectionCondition` .

В нашем коде `exactTexts` - именно "collection condition":

```
import static com.codeborne.selenide.CollectionCondition.exactTexts;
//...

// add tasks "a", "b", "c"
element("#new-todo").setValue("a").pressEnter();
element("#new-todo").setValue("b").pressEnter();
element("#new-todo").setValue("c").pressEnter();

// tasks should be "a", "b", "c"
elements("#todo-list>li").shouldHave(exactTexts("a", "b", "c"));
```

А с примером "element condition" давай познакомимся на другом интересном примере.

Обрати внимание, что если посмотреть "source" веб-страницы TodoMVC в браузере:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<title>Todomvc</title>
<meta name="description" content="">
<meta name="viewport" content="width=device-width, initial-scale=1">
<meta name="todomvc/config/environment" content="%7B%22modulePrefix%22%3A%22to
domvc%22%2C%22environment%22%3A%22production%22%2C%22baseURL%22%3Anull%2C%22lo
cationType%22%3A%22hash%22%2C%22EmberENV%22%3A%7B%22FEATURES%22%3A%7B%7D%2C%22
EXTEND_PROTOTYPES%22%3A%7B%22Date%22%3Afalse%7D%7D%2C%22APP%22%3A%7B%22name%22
%3A%22todomvc%22%2C%22version%22%3A%220.0.0+47754603%22%7D%2C%22exportApplicat
ionGlobal%22%3Afalse%7D" />

<link rel="stylesheet" href="assets/vendor7b5c98520910afa58d74e05ec86cd
873.css" integrity="sha256bsagGHduhay9QPLUFpddcZFq7Kmr2ScM3VKnWhdX8oM=sha512eN
sGN2aLecWPvoqNVH8oXK8o/IJ7rO5ti0zgS8lF8LiwmKUHdEIuFduwcDL1VLAt2r+3YjgDzoSNYK6c
57pJzw==" >
<link rel="stylesheet" href="assets/todomvcd41d8cd98f00b204e9800998ecf842
7e.css" integrity="sha25647DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU= sha51
2z4PhNX7vuL3xVChQ1m2AB9Yg5AULVxXcg/SpIdNs6c5H0NE8XYXysP+DGNKHfuwvY7kxvUdBeoGlO
DJ6+SfaPg==" >

</head>

<body>

<script src="assets/vendor-22a6a947beb9d4b28a782879e18b0f65.js" integrity="sha
256-M1pD1q8B9PyrHkKX/mlfsOLLHMrh/x7vvCGLRC63OyI= sha512-I4vC6+4Z29iHB4nJBCzcIj
rgMtDerq7sxYLE2lM/AhjkNLp6gf2+Zpne8OtGdTLRkyvTqzPm1V7gq9w4HQxOXg==" >
</script>
<script src="assets/todomvc-d191d5c1c9280b108d69413f052d3bb4.js" integrity="sh
a256-1wUxToLQP6yjsvm0/8e3xQnv7SbSYcj3P/OPEdThZOk= sha512-X5K7gsPRYsUSRvJcnj80S
LlclDd4X/g1Qgle1L1P4Zb63eUM0mYEfQECBcjcks7iZFItSGr8EVPOVELXO35HUA==" >
</script>

</body>
</html>
```

То окажется, что никаких знакомых нам элементов типа `#new-todo` - там нет :) Это значит, как минимум, то, что эти элементы добавляются динамически после загрузки html страницы с помощью JavaScript. То есть, они реально "появляются на странице после загрузки", а не сразу. И, конечно же, это занимает какое-то время. Возможно, нам просто повезло, ничего не тормозило, и JavaScript динамически добавил наш элемент `#new-todo` достаточно быстро, чтобы мы уже смогли продолжить наш сценарий. Ну, наверное не всегда ж нам будет везти, и стоит, наверное, этот момент предвидеть и дождаться видимости элемента перед тем, как совершать нужные действия:

```
//...

// add tasks "a", "b", "c"
element("#new-todo").shouldBe(visible).setValue("a").pressEnter();
element("#new-todo").setValue("b").pressEnter();
element("#new-todo").setValue("c").pressEnter();

// tasks should be "a", "b", "c"
elements("#todo-list>li").shouldHave(exactTexts("a", "b", "c"));
```

Вот этот `visible` - и есть тем другим "element condition" ;), живущим в отдельном классе `Condition`:

```
//...
import static com.codeborne.selenide.CollectionCondition.exactTexts;
import static com.codeborne.selenide.Condition.visible;
//...

// add tasks "a", "b", "c"
element("#new-todo").shouldBe(visible).setValue("a").pressEnter();
element("#new-todo").setValue("b").pressEnter();
element("#new-todo").setValue("c").pressEnter();

// tasks should be "a", "b", "c"
elements("#todo-list>li").shouldHave(exactTexts("a", "b", "c"));
```

А метод `shouldBe` (как и `should` или `shouldHave`) - играет как раз роль "ожидания момента, когда элемент будет удовлетворять условию", а не только совершает проверку по условию...

Также заметь, что мы использовали разные "синонимы" для should-методов с целью получить более читабельный код - для кондишена `visible` мы подобрали в компанию метод, имя которого заканчивается на `Be`:

```
//...
element("#new-todo").shouldBe(visible).setValue("a").pressEnter();
//...
```

А для "коллекшен-кондишена" (`CollectionCondition`) `exactTexts` - имя should-метода заканчивается на `Have`:

```
//...
elements("#todo-list>li").shouldHave(exactTexts("a", "b", "c"));
```

Ради той же читабельности мы использовали статический импорт:

```
//...
import static com.codeborne.selenide.Condition.visible;
//...
element("#new-todo").shouldBe(visible).setValue("a").pressEnter();
//...
```

Вместо "обычного":

```
//...
import com.codeborne.selenide.Condition;
//...
element("#new-todo").shouldBe(Condition.visible).setValue("a").pressEnter();
//...
```

Вот так мы и познакомились с одними из очень важных особенностей работы Selenide - использовании кондишенов разного вида в should-методах, как для проверок (асертов), так и явных ожиданий элементов и коллекций элементов.

А теперь хорошие новости - на самом деле, писать `.shouldBe(visible)` перед каждым действием не нужно. Selenide сам подождет пока элемент будет доступен для выполнения действия над ним:

```
//...
element("#new-todo").setValue("a").pressEnter();
//...
```

Это одно из главных отличий Selenide от "чистого" Selenium Webdriver. В Selenium неявные ожидания отключены по умолчанию и если включены - ждут только до появления элемента в DOM (html страницы), но при этом элемент все еще может быть невидимым и с ним не получится взаимодействовать, тест упадет. Или до окончания загрузки элемент может быть перекрыт другим элементом, и поэтому все еще быть недоступным для выполнения действия над ним. В Selenide же ожидания по умолчанию включены, с таймутом ожидания по умолчанию, равному `4000` миллисекунд, и доступным к конфигурированию через:

```
//...
import  com.codeborne.selenide.Configuration;
//...
Configuration.timeout = 6000;
//...

// add tasks "a", "b", "c"
element("#new-todo").setValue("a").pressEnter();
element("#new-todo").setValue("b").pressEnter();
element("#new-todo").setValue("c").pressEnter();

// tasks should be "a", "b", "c"
elements("#todo-list>li").shouldHave(exactTexts("a", "b", "c"));
```

Этот `Configuration` - целый склад всевозможных настроек, которыми можно тюнить поведение Selenide и его команд, обязательно поисследуй все возможные опции;)

Например, через соответствующее поле этого класса, можно поменять тип используемого браузера, и вместо запуска тестов в Chrome, запустить их в Firefox:

```
//...
```

```
Configuration.browser = "firefox"; // или = Browsers.FIREFOX

// add tasks "a", "b", "c"
element("#new-todo").setValue("a").pressEnter();
element("#new-todo").setValue("b").pressEnter();
element("#new-todo").setValue("c").pressEnter();

// tasks should be "a", "b", "c"
elements("#todo-list>li").shouldHave(exactTexts("a", "b", "c"));
```

Эти же конфигурации можно изменять не только из кода, но и из командной строки:

```
> cd $PROJECT_FOLDER
> ./gradlew build -Dselenide.browser=firefox
```

Все эти нюансы можно найти в том же исходном коде класса `Configuration` ;)

```
public class Configuration {

  //...

  /**
   * Which browser to use.
   * Can be configured either programmatically or by system property "-Dseleni
de.browser=ie".
   * Supported values: "chrome", "firefox", "legacy_firefox", "ie", "htmlunit"
, "phantomjs", "opera", "safari", "edge", "jbrowser"
   * <br>
   * Default value: "chrome"
   */
  public static String browser = defaults.browser();
```

Давай теперь вернемся к нашему сценарию...

```
// open TodoMVC page
await open("http://todomvc.com/examples/emberjs/");

// add tasks: "a", "b", "c"
element("#new-todo").setValue("a").pressEnter();
```

```
element("#new-todo").setValue("b").pressEnter();
element("#new-todo").setValue("c").pressEnter();
// tasks should be "a", "b", "c"
elements("#todo-list>li").shouldHave(exactTexts("a", "b", "c"));

// toggle b
// completed tasks should be b
// active tasks should be a, c
```

и закончим его реализацию;)

Следующий шаг - обозначить задачу как "completed". Для этого нужно:

```
// ...

// toggle b:
    /*1. among all tasks 2. find the one with "b" text
      3. find it's "toggle" checkbox 4. click it*/
}
```

В Selenide это сделать очень просто:

```
// ...

// toggle b

        /*among all tasks*/
                        /*find the one with "b" text*/
elements("#todo-list>li").findBy(exactText("b")).find(".toggle").click();
                                                                /*click it*/
                                            /*find it's "toggle" checkbox*/
}
```

Как видно, для поиска нужного элемента среди других элементов коллекции, мы используем такого же вида условие как и для "ожиданий-проверок".

```
import static com.codeborne.selenide.Condition.exactText;
// ...
```

```
// toggle b

        /*among all tasks*/
                        /*find the one with "b" text*/
elements("#todo-list>li").findBy(exactText("b")).find(".toggle").click();
                                                                /*click it*/
                                            /*find it's "toggle" checkbox*/
}
```

Без таких возможностей Selenide, в обычном Selenium Webdriver пришлось бы использовать намного более громоздкий и менее читабельный XPath селектор.

Финальные шаги отвечают за проверку результата предыдущего действия:

```
// ...

// tasks should be "a", "b", "c"
elements("#todo-list>li").shouldHave(exactTexts("a", "b", "c"));

// toggle b
elements("#todo-list>li").findBy(exactText("b")).find(".toggle").click();

// completed tasks should be "b":
    /*among all tasks - filter only completed ones - check their texts*/
// active tasks should be "a", "c":
    /*among all tasks - filter only not completed ones - check their texts*/
}
```

Перевод на "язык Selenide" все такой же простой:

```
// ...

// completed tasks should be "b"

        /*among all tasks*/
                        /*filter only completed ones*/
elements("#todo-list>li").filterBy(cssClass("completed"))
        .shouldHave(exactTexts("b"));
       /*check their texts*/
```

```
  // active tasks should be "a", "c"

          /*among all tasks*/
                        /*filter only not completed ones*/
  elements("#todo-list>li").filterBy(not(cssClass("completed")))
          .shouldHave(exactTexts("a", "c"));
         /*check their texts*/
      }
  }
```

Как видно, в целом код действительно получился настолько читабельным, что наши комментарии...

```
  // open TodoMVC page
  open("http://todomvc.com/examples/emberjs/");

  // add tasks: "a", "b", "c"
  element("#new-todo").setValue("a").pressEnter();
  element("#new-todo").setValue("b").pressEnter();
  element("#new-todo").setValue("c").pressEnter();

  // tasks should be "a", "b", "c"
  elements("#todo-list>li").shouldHave(exactTexts("a", "b", "c"));

  // toggle b
  elements("#todo-list>li").findBy(exactText("b")).find(".toggle").click();

  // completed tasks should be b
  elements("#todo-list>li").filterBy(cssClass("completed"))
              .shouldHave(exactTexts("b"));

  // active tasks should be a, c
  elements("#todo-list>li").filterBy(not(cssClass("completed")))
              .shouldHave(exactTexts("a", "c"));
```

... совсем не нужны:

```
  open("http://todomvc.com/examples/emberjs/");
```

```
element("#new-todo").setValue("a").pressEnter();
element("#new-todo").setValue("b").pressEnter();
element("#new-todo").setValue("c").pressEnter();
elements("#todo-list>li").shouldHave(exactTexts("a", "b", "c"));

elements("#todo-list>li").findBy(exactText("b")).find(".toggle").click();
elements("#todo-list>li").filterBy(cssClass("completed"))
        .shouldHave(exactTexts("b"));
elements("#todo-list>li").filterBy(not(cssClass("completed")))
        .shouldHave(exactTexts("a", "c"));
```

Этих основных команд, поддержки функций Autocomplete и Quick Fix от IDE, мужества покопаться в коде, и, в конце концов, официальной документации должно быть вполне достаточно, чтобы начать писать первые автотесты с Selenide + java.

Удачи;)

Задание: Selenide и CSS селекторы

Особенность Selenide в том, что при его использовании, обычно, не нужны сложные CSS или XPath селекторы. Достаточно самых простых селекторов, которые ищут элементы по атрибутам, например:

```
element(byName("q")) // ...

element(byId("new-todo")) // ...
//или еще проще для последнего примера:
element("#new-todo") // ...
```

Но чтобы прочувствовать все это на собственном опыте, да еще и закрепить знания в CSS селекторах, твоим заданием будет реализовать сценарий из раздела "Введении в Selenide":

```
@Test
void completesTask() {
    // open TodoMVC page

    // add tasks: "a", "b", "c"
    // tasks should be "a", "b", "c"

    // toggle b
    // completed tasks should be b
    // active tasks should be a, c
}
```

для приложения TodoMVC (emberjs version), не используя при этом следующих методов Selenide:

- `SelenideElement#find` (то есть `element(".parent").find(".child")`) или `SelenideElement#$`
- `Element#findAll` (то есть `element(".parent").findAll(".child")`) или `SelenideElement#$`
- `ElementsCollection#(int index)` (то есть `elements(".item").get(1)`)

- `ElementsCollection#first`
- `ElementsCollection#findBy(Condition condition)`
- `ElementsCollection#filterBy(Condition condition)`
- и других им подобных ;)

По сути, все, чем можно пользоваться - это:

- `element(...).*` , где `*` - это действия над элементом или ассерты (`should` / `shouldBe` / `shouldHave`)
- `elements(...).should(...)` (или с `shouldBe` / `shouldHave`)
- и CSS селекторами для составления локаторов, которые можно передавать просто как строки, например: `element("#new-todo")`

При этом важно стараться написать код как можно более аналогичный показанному в "Selenide в действии".

CSS селекторы - очень простые, удобные и читабельные (проще чем XPath селекторы), но при этом - довольно ограниченные в возможностях (по сравнению с теми же XPath селекторами). Поэтому, в любом случае - написать код, аналогичный тому, что был показан в "Selenide в действии" - не получится. Ну что ж, придется подумать и найти другой путь в поиске элементов и проверок над ними;)

Также, не спеши создавать переменные и методы для хранения/переиспользования локаторов либо тестовых данных. Мы к этому еще вернемся в следующих заданиях. Сейчас важно двигаться шаг за шагом, детально прорабатывая темы на которых мы фокусируемся в каждом из уроков.

Дополнительные материалы:

- Краткая шпаргалка по CSS селекторам от w3school
- Рецепты CSS и XPATH
- Видео от automated-testing.info о том, как писать локаторы (CSS, XPath)

В особо трудную минуту, можно подглядывать за подсказками в следующую главу с FAQ. В нем же можно будет найти список частых ошибок по которому проверить свое финальное решение;)

Удачи!

Решение: Список частых вопросов

"Строгие, но хрупкие" или "слабые, но стабильные" локаторы?

Строгими называются локаторы (или "селекторы", что, в принципе, примерно одно и то же) с более точной информацией о предмете поиска – элементе. Они более *хрупкие*, ведь если что-то поменяется в "пути селектора" – элемент уже не будет найден, и тест упадет, даже если тестируемая фича все еще будет работать. Все это будет увеличивать время поддержки существующих тестов. На некоторых проектах, где структура html часто меняется – это может привести к очень большим неоправданным затратам.

В противоположность строгим, **слабые** локаторы используют только минимум информации о элементе, его атрибутах и месте в структуре html. С такими локаторами, обычно, тесты – более *стабильные*. Обычно, это упрощает их поддержку, особенно учитывая то, что автоматизация пользовательского интерфейса веб-приложений сама по себе не очень "легковесна" (такие тесты сложнее и медленнее чем те же юнит-тесты или API-тесты).

При этом, на проектах, где автоматизация используется в первую очередь тестировщиками для полного контроля всего процесса тестирования, и где программисты, к сожалению, не так сильно заинтересованы в поддержке и использовании автотестов – "более строгие локаторы" могут быть полезны тем, что своей "нестабильностью" в тестах – всегда будут показывать зоны в приложении – "где что-то изменилось". Даже несмотря на то, что покрытая тестом фича работает, возможно, перестало работать что-то другое, с ней связанное. Ведь автотесты не могут заменить полностью ручное исследовательское тестирование, их покрытие намного слабее. Поэтому тестировщики могут использовать такую "хрупкость автотестов", как сигнал к более тщательному ручному тестированию в соответствующих "зонах приложения".

Если же автотесты используются разработчиками, с целью проверять, не поломали ли они те функциональные аспекты приложения, которые точно работали ранее – такие "строгие локаторы" – наоборот, будут приносить много проблем, и задержек. На самом деле, главная цель автотестов – как раз помогать разработчикам, давать им как можно более быстрый фидбек об их изменениях, а не уменьшать время ручной

регрессии для мануальных тестировщиков. При таком подходе автоматизация намного более эффективна, ведь влияет на обеспечение качества продукта на более ранних стадиях его разработки. Это то, к чему стоит стремиться. Поэтому "более строгие" локаторы – это не то, чего стоит "желать" на своем проекте. Но... Все зависит от контекста и от конкретного проекта;)

Примеры разного вида таких локаторов можно найти в соответствующих разделах списка частых ошибок и их решений (FAM) ниже.

Локаторы, автоматически сгенерированные в инспекторе, или подобранные вручную?

Тема хрупкости и стабильности локаторов также актуальна в контексте способа подбора локаторов. Их можно подбирать вручную, а можно получить уже готовый сгенерированный локатор в инспекторе браузера (Browser: Developer Tools/Inspector). Второй способ может быть быстрее, особенно в приложениях со сложной структурой элементов, но может выдавать и более "строгие локаторы", и от того – более хрупкие, и часто - намного менее читабельные. Это может привести к большим затратам в поддержке автотестов при условии, что структура html будет часто меняться на проекте. А если не будет? Тогда, возможно, действительно можно повысить свою эффективность, не заморачиваясь ручным подбором локаторов, и быть намного более быстрым при их подборе с помощью инспектора. А чтобы хоть как-то решить проблему читабельности – можно выносить такие сгенерированные локаторы в переменные, и уже их использовать в автотестах. Все, как всегда – зависит от условий на реальных проектах. Эта тема неплохо раскрыта в статье Сергея Пирогова "Мой взгляд на хорошие локаторы" и моих комментариях к ней ;)

В любом случае, стоит освоить оба способа – и иметь их в своем арсенале.

Когда выносить локаторы в отдельные переменные и/или классы?

Во-первых, давай не будем спешить;) Мы еще успеем навыносить и локаторы в переменные, и код действий над элементами в методы, и потом "скрыть" все эти "технические детали" в какой-то класс, и положить этот класс в укромное и удобное местечко в нашем проекте. Но сейчас не стоит этого делать осознанно, с целью "подождать момента, пока нам реально припечет", пока не увидим в этом еще большей нужды. Такой подход в контексте этого обучения - позволит нам на практике

осознать – когда действительно появляется эта нужда "что-то куда-то выносить" и чем это нам поможет. Поэтому пока давай "выжидать" и "собирать компромат" на эти "сырые" локаторы ;)

Но раз, вопрос появился, давай успокоим сердце некоторыми объяснениями, касающимися нашего случая.

Возможно самый яркий "компромат", который может бросаться в глаза – это уровень **читабельности** локаторов для "простых смертных". Если на проекте практикуется разделение между мунальными тестировщиками и автоматизаторами, то, например, мануальным тестировщикам, что-бы проверить, покрыли ли автоматизаторы то, что было нужно в автоматизированном сценарии – придется сначала разбираться, что такое CSS или XPath. Но если, автоматизаторы не использовали XPath, то разобрать синтаксис CSS не так уж сложно, и в любом случае полезно. И если автотест будет выглядеть в стиле оригинального решения на Selenide из урока, и атрибуты использованные для нахождения элементов читабельны сами по себе, то после небольшого знакомства и консультаций с автоматизаторами такие сценарии должны быть вполне понятными:

```
open("http://todomvc.com/examples/emberjs/");

element("#new-todo").setValue("a").pressEnter();
element("#new-todo").setValue("b").pressEnter();
element("#new-todo").setValue("c").pressEnter();
elements("#todo-list>li").shouldHave(exactTexts("a", "b", "c"));

elements("#todo-list>li").findBy(exactText("b")).element(".toggle").click();
elements("#todo-list>li").filterBy(cssClass("completed"))
            .shouldHave(exactTexts("b"));
elements("#todo-list>li").filterBy(not(cssClass("completed")))
            .shouldHave(exactTexts("a", "c"));
```

Второй компромат - это **повторение кода**. Обрати внимания на куски вида `element("#new-todo")` или `elements("#todo-list>li")` - они довольно часто повторяются в сценарии. Можно было бы пожаловаться на то что такие куски каждый раз медленней набирать чем имена "заготовленных переменных". Но в реальной жизни, скорее всего ты будеш просто дублировать целые строки кода с помощью

какой то комбинации гарячих клавиш (например `Command + D` в Intellij Idea IDE под Mac OS), и редактировать соответствующие места. Можно было бы тогда пожаловаться на то, что если что то в таких повторяющихся кусках поменяется, собственно часть самого локатора - то прийдется менять во всех местах (вспомни принцип DRY). Но и в этом случае проблема может неплохо решаться обычной функцией редактора "найти и заменить";) Предпочтение таким вот "более быстрым и простым" подходам соответствует следованию принципу KISS ("Keep It Simple, Stupid"). Конечно, все принципы и подходы имеют свой контекст применения. Все зависит от условий, и в определенный момент мы можем понять, что следование тому же принципу DRY и соотвтествующее вынесение локаторов в переменные - теперь более предпочтительно. Главное, как и в обычной жизни, стараться не сильно спешить, "что-бы людей не насмешить";). Это особенно важно если мы начинаем проект, который может оказаться не "шаблонным". Последние пожелания, кстати, отображает принцип YAGNI ("You Aren't Gonna Neede It"), который обычно всегда идет рука об руку с принципом KISS;)

Все вышеупомянутые "упрощения" особенно актуальны именно для программирования в сфере тестирования, то есть в написании авто-тестов. Это объясняется тем, что

- тестов очень много
- их пишут разные люди
- тест, вероятно, будет прочитан человеком, который его не писал
- а тесты часто требуют повышенного внимания людей и их быстрой реакции, потому что могут часто падать

Поэтому так важно, чтобы тесты были реализованы как можно проще, с минимумом усилий, с минимумом использований сложных конструкций языка, в том числе и таких безобидных на первый взгляд, как переменные, классы, и методы. Потому что разбираться со всем этим инженеру, который все это не писал, а теперь нуждается в быстром исправлении падающего теста – нет особого времени. И главное, что в силу небольшой сложности тестовых сценариев в сравнении с сложностью реализации внутренней логики продукта – тесты действительно реально сделать простыми.

Единственное, стоит учитывать, что "простота" должна распространяться не только на "реализацию" тестов, но и на их "использование" – то есть в частном случае "их чтение", а значит и "понятность". Причем, поскольку последняя избушка ("чтение") –

как раз к нам передом, а "к лесу" задом – то она и является главным маркером, контролирующим "уровень простоты реализации". Если для наших конечных пользователей тестов – тесты и так понятны и читабельны – то зачем вводить новые "абстракции" – переменные, методы, классы – при этом усложняя реализацию? Если же наоборот – читать тесты сложно, сложно быстро распознать за кодом теста тестовую логику – тогда наоборот, стоит подумать об использовании соответствующих абстракций. Ко всему этому, и ко всем упомянутым выше принципам - мы еще вернемся в одном из следующих разделов. А сейчас поиграемся в продвинутых инженеров, кому и так все понятно, чтобы ощутить разницу и приобрести важный опыт;)

Как найти элемент из списка однотипных элементов по нужному тексту его внутреннего элемента (CSS)?

> Допустим, есть список `<li>` элементов, внутри каждого из которых есть "лейблы" (`<label>`) с разными значениями – я не понял как сделать поиск по значениям этих "лейблов"... Точнее, сами лейблы-то с такими значениями найти можно, а вот именно чтобы найти нужный элемент `li` , у которого есть "лейбл" с определенным значением – не вышло. Пришлось делать через нахождение элемента по индексу.

Важно понимать, что все, что находится `<label>между открытым и закрытым тегом элемента</label>` технически считается текстом элемента, а не "значением". Значениями корректней называть "значения атрибута value некоторых элементов, таких как `<input>` ".

Так вот, дело в том, что CSS селекторы, действительно, не умеют искать элементы по тексту, и тем более не умеют выбирать нужный элемент из списка по какому-либо признаку (будь-то "значение" или "текст"), присущему одному из его внутренних элементов. Вот с помощью xPath селекторов это сделать можно, ну и, конечно же, это можно сделать с помощью некоторых методов Selenide, описанных в Selenide Demo, которые в этом задании использовать запрещено;)

Так что, если речь идет только о использовании CSS селекторов, все что остается – это искать элемент по индексу.

Список частых ошибок

Структурная информация в именах тест-сьюта и тестов, которая не несет пользы с точки зрения тестирования

Пример 1

```
public class TodoMvcCssVersionTest {

    @Test
    void completesTask() {
```

Пример 2

```
public class TodoMvcTest {

    @Test
    void completesTaskCssVersion {
```

Имена тест-класса и тест-метода играют, по сути, роль имен "тест-сьюта" и "тест-кейсов" соответственно, а последние – должны отображать то, что мы тестируем, а не, например, "нюансы реализации", как в примерах выше. Если хочется структурировать свои решения для заданий – лучше создать отдельные папки и разложить в них, как по полочкам – разные версии решений.

Например так... (ищи решение на следующей странице)

```
selenide-intro/
  src/
    test/
      java/
        com/
          yashaka/
            selenideintro/
              csstask/
                TodoMvcTest.java
              xpathtask/
                TodoMvcTest.java
```

Несоответствие имен тест-сьюта или тест-кейсов целям тестирования

Пример 1

```
public class TodoMvcTest {

    //...

    @Test
    void testTodoMvc() {
```

Пример 2

```
public class TodoMvcCompletesTest {

    //...

    @Test
    void completes() {
```

Имена тест-класса и тест-метода играют, по сути, роль имен "тест-сьюта" и "тест-кейсов" соответственно, а последние – должны отображать то, что мы тестируем. В примерах выше, имя тест-метода не несет совсем никакой полезной информации с точки зрения целей тестирования для нашего сценария. Во втором примере имя тест-класса недостаточно абстрактно в контексте нашего покрытия (мы как раз "завершение" задачи покрываем в тест-методе).

Решение...

```
public class TodoMvcTest {

    //...

    @Test
    void completesTask() {
```

Установка пути к chromedriver в коде

Пример

```
System.setProperty("webdriver.chrome.driver", "C:/work/web
drivers/chromedriver.exe");
```

Обычно, намного проще и универсальней **прописать путь к chromedriver в системную переменную окружения PATH** (возможно, чтобы изменения вступили в силу, придется перегрузить OS). Такое решение лучше по той причине, что тесты могут быть запущены на разных машинах, и если мы пропишем точный путь в коде тестов к chromedriver – нам придется на всех машинах подгонять местоположение драйвера к этому пути. А что делать, если мы захотим запустить тесты под другой OS, где и пути-то могут быть совсем в другом формате? ;) Поэтому лучше переложить обязанность за "установку пути драйвера" с кода тестов на конфигурации машин для запуска тестов.

Как установить PATH? – можно загуглить решение конкретно под нужную OS;)

Использование "запрещенных в задании методов Selenide"

Пример

```
element("#todo-list>li:nth-child(2)").find(".toggle").click();
```

В этом задании мы учимся находить элементы только силами CSS. В заданиях указано, какие типы методов нельзя использовать;)

В примере выше использован метод `SelenideElement#find` (метод `find`, вызванный для объекта класса `SelenideElement`, который возвращает статический метод `element` класса Selenide) – он позволяет найти внутренний элемент по селектору внутри другого, а наша задача научиться такому "внутреннему поиску" только с помощью CSS селекторов в этом задании;)

Решение...

```
element("#todo-list>li:nth-child(2) .toggle").click();
```

Хрупкие строгие локаторы

Обычно есть несколько способов написать локатор что-бы найти один тот же элемент на странице. Можно использовать очень много информации о структуре html (вложенность элементов, порядок их следования один за другим) и содержании (тексты, значения атрибутов элементов) и тогда получится "строгий" локатор. Или использовать минимум информации - и тогда получится "слабый" или "гибкий" локатор. В следующей серии ошибок мы расмотрим типичные случаи когда строгие локаторы приводят к нестабильным "хрупким" тестам (flaky tests).

Хрупкие строгие локаторы. Избыточность в путях - привязка к тегу

Допустим у нас есть элементы:

```
<li>
  <div class="crd">
    <label>1</label>
    <input type="button" class="destroy"/>
  </div>
  ...
</li>
<li>
  <div class="crd">
    <label>2</label>
    <input type="button" class="destroy"/>
  </div>
  ...
</li>
...
```

Тогда следующий селектор вероятно будет "хрупким":

```
li:nth-child(2) input.destroy
```

Для точного определения элементов лучшей практикой в разработке внутреннего представления html страниц принято использовать атрибуты элементов, в большинстве случаев - `id` (для определения уникальных элементов) и `class` (для

определения группы однотипных элементов). Поэтому использовать теги элементов особенно когда элемент можно определить по css классу - менее предпочтительно. В нашем случае внутрення реализация может например поменятся от:

```
<input type="button" class="destroy">
```

до

```
<button class="destroy"></button>
```

В таком случае наш селектор перестанет работать и тест упадет. Хотя функционал по сути не поменялся - как была кнопка удаления задачи так и осталась. То есть функциональный тест упасть не должен был бы.

Решение:

```
li:nth-child(2) .destroy
```

Хрупкие строгие локаторы. Избыточность в путях - привязка к точному пути

Допустим у нас есть элементы:

```
<li>
  <div class="crd">
    <label>1</label>
    <input type="button" class="destroy"/>
  </div>
  ...
</li>
<li>
  <div class="crd">
    <label>2</label>
    <input type="button" class="destroy"/>
  </div>
  ...
</li>
...
```

Тогда следующий селектор вероятно будет "хрупким":

```
li:nth-child(2)>div>.destroy
```

С развитием проекта внутренняя структура html может усложнятся. И там где сейчас у нас есть два "родительских элемента" по отношении к тому элементу что мы ищем:

```
<li>
  <div class="crd">
    <label>2</label>
    <input type="button" class="destroy"/>
  </div>
  ...
</li>
```

со временем может появится 3:

```
<li>
```

```
    <div>
      <div class="crd">
        <label>2</label>
        <input type="button" class="destroy"/>
      </div>
    </div>
    ...
  </li>
```

В таком случае наш селектор перестанет работать и тест упадет. Хотя функционал по сути не поменялся - как была кнопка удаления задачи так и осталась. То есть функциональный тест упасть не должен был бы.

Решение...

```
li:nth-child(2) .destroy
```

Хрупкие строгие локаторы. Привязка к полному значению аттрибута class либо его начала или конца

Допустим у нас есть элементы:

```
<li class="end">1</li>
<li class="middle end">2</li>
<li class="end">3</li>
```

И нам нужно уметь отдельно находить второй элемент (с текстом 2) и отдельно оба - первый (1) и третий (3). Мы можем увидеть что их отличает значение атрибута `class` . У второго элемента это значение начинается с текста `"middle"` , а у первого и третьего - целое значение равно `"end"` .

Соответственно мы можемо построить и локаторы соответственно:

- найти элемент(ы) у которого значение аттрибута класс начинается с текста `"middle"`

```
[class^="middle"]
```

- найти элемент(ы) у которого целое значение аттрибута класс равно тексту `"end"`

```
[class="end"]
```

Аналогичные примеры будут и для XPATH селекторов. Иногда такой подход единственное что остается. Но в случае когда мы работаем с атрибутом `class` стоит быть осторожным. Дело в том что новые CSS классы склонны появляться у элементов время от времени. И если, как в примере выше, мы намертво привяжемся к полному значению атрибута `class` , либо, его началу, то если вдруг програмисты решать повесить на наши элементы еще один цсс класс, например `"start"` :

```
<li class="start end">1</li>
<li class="start middle end">2</li>
<li class="start end">3</li>
```

то нашы локаторы "сломаются" и перестанут находить нужные нам элементы. При этом фича, которую мы должны были покрывать в нашем тесте где мы использовали эти локаторы – все еще должна работать, и, соответственно, результат теста – будет не показательным с точки зрения его "цели тестирования". Такая хрупкость локаторов может быть нежелательна на проектах где могут иметь место частые изменения такого вида. В таких случаях стоит поискать более "гибкий" и "стабильный" локатор.

Обычно можно построить локатор через поиск по вхождению определенного текста в значение атрибута, или даже по вхождению "слова" в значение аттрибута. В CSS селекторах конкретно в случае поиске по CSS классу для этого даже есть специальное сокращение. Когда же нужно исключить из поиска элементы с значением аттрибутов имеющим определенные слова - и для этого и в CSS и XPATH селекторах есть соответствующие модификаторы поиска.

Решение...

- найти второй элемент

  ```
  [class~="middle"]
  ```

  или лучше:

  ```
  .middle
  ```

- найти все элементы кроме второго

  ```
  :not(.middle)
  ```

Хрупкие гибкие локаторы. Привязка к частичному значению атрибута class

Иногда с "гибкостью" локаторов можно перестараться, и в результате получится не стабильный а хрупкий локатор.

Допустим у нас есть html код:

```
<li class="foo bar">1</li>
<li class="foo">2</li>
<li class="foo bar">3</li>
```

Нам нужно уметь находить элементы с классом `bar`. И вот мы строим наш локатор по принципу:

- найти элемент(ы) у которых значений атрибута `class` содержит (contains) текст `"bar"`.

```
[class*="bar"]
```

И тут девелоперы в какой то момент осознали, что есть особенный вид элементов `foobar` (привет утконосам из Австралии;))...

```
<li class="foo bar">1</li>
<li class="foo">2</li>
<li class="foo bar">3</li>
<li class="foobar">4</li>
```

И вуаля, локатор который был предназначен для нахождения только "bar" элементов - теперь находит и элементы "foobar".

Поэтому когда мы работаем с определением элементов исходя из значений атрибута class - обычно лучше искать "по CSS class-у", то есть вхождению соответствующего "**целого** слова" в значение атрибута (учти, что *CSS class* - это одно из слов в значении атрибута `class`, а *class* - это имя одноименного атрибута)

Решение...

```
.bar
```

Построение локатора с помощью менее читабельных и понятных аттрибутов либо их значений

Пример

```
elements("#todo-list li[class='ember-view']").shouldHave(exactTexts("a",
"c"));
```

Главная проблема в этом коде - это потенциальная хрупкость локатора как следствие привязки к целому значению аттрибута class.

Но есть и другая ошибка. Для построения локатора было выбрано значение `"ember-view"` - которое абсолютно ничего не говорит о том - о чем этот локатор. Этот локатор должен искать "активные задачи". А ищет какие-то "задачи с ember-view `o_O` ". Иногда выбора нет, и более понятный локатор не построешь. Тогда что бы код все еще был понятным и его было просто поддерживать в будущем - используют переменные с более четким именем. Но в этом случае, можно подобрать намного более читабельный и даже более лаконичный локатор.

Подсказка 1...

Сфокусируйся на том что "что должен найти локатор". А найти он должен "активные задачи". Каким другим способом можно перефразировать "активность" задачи?

Подсказка 2...

В этом случае можно построить более простой локатор исходя из того что "активные задачи" - это тоже самое что **"не завершенные задачи"**.

Решение...

```
elements("#todo-list li:not(.completed)").shouldHave(exactTexts("a", "c"));
```

Слишком краткие и потому менее информативные, неочевидные локаторы

Часто с целью сделать локатор покороче, мы не учитываем о том что краткость может навредить читабельности, и сделать тесты более сложными в восприятии (мы еще вернемся к этой теме в следующих уроках - смотри раздел о читабельности и лаконичности...)

Если тест откроет человек который его не писал, и увидит там код с локатором вида:

Пример 1

```
element("li:nth-child(2) .toggle").click();
```

... ему может быть не очевидно что `li` это элемент списка задач а не какого то другого списка.

В следующем коде также "идет речь о элементах списка задач" но при этом локатор никак об этом не "информирует":

Пример 2

```
elements(".completed").shouldHave(exactTexts("b"));
```

Решение...

Пример 1

```
element("#todo-list li:nth-child(2) .toggle").click();
```

Пример 2

```
elements("#todo-list li.completed").shouldHave(exactTexts("b"));
```

Неконсистентные селекторы

Пример

```
//...
elements("#todo-list li").shouldHave(exactTexts("a", "b", "c"));

element("#todo-list>li:nth-of-type(2) .toggle").click();
//...
```

Важно быть как можно более последовательным и единообразным в построении локаторов. Потому что чем больше мы сделаем код "разнообразным" – тем сложнее его будет понимать другим, тем менее читабельными будут тесты, тем больше у других членов команды будет уходить времени на поддержку тестов. А в тестах очень важна читабельность, учитывая то, что их очень много, их пишут разные люди и когда тесты падают – нужно как можно быстрее понять что произошло, найти и разобраться с причиной.

В коде выше для нахождения одних и тех же сущностей – элементов задач – используются разные селекторы:

- `#todo-list li` (ищет все элементы `li` внутри элемента `#todo-list` на любой глубине вложенности)
- `#todo-list>li` (ищет все элементы `li` внутри элемента `#todo-list` на первом уровне вложенности)

Это будет сбивать с толку во время разбора кода теста тех, кто его не писал. Ведь первый сигнал который поступит в мозг от глаз будет - "это разные селекторы". И только потом, затратив время на обработку - мозг поймет что это один и тот же селектор. Зачем же утруждать его лишней работой? ;)

Еще один минус в том, что при глобальной замене этой части селектора на какую то другую - не получится заменить все вхождения сразу, как раз за счет того что эти части селектора - неконсистентны. Конечно, можно сказать что если бы мы вынесли эту часть в переменную - то и проблемы такой не было бы. Но контексты могут быть разными на разных проектах, и не всегда может быть полезно использовать переменные.

Решение...

либо:

```
//...
elements("#todo-list>li").shouldHave(exactTexts("a", "b", "c"));

element("#todo-list>li:nth-of-type(2)").element(".toggle").click();
//...
```

либо:

```
//...
elements("#todo-list li").shouldHave(exactTexts("a", "b", "c"));

element("#todo-list li:nth-of-type(2)").element(".toggle").click();
//...
```

Первый вариант более строгий, поэтому если на проекте велика вероятность, что появятся другие элементы `li` внутри `#todo-list` (например глубже...), то такой селектор будет менее "стабильным".

Второй вариант менее строгий, поэтому если вдруг текущие элементы `li` вследствие каких-то изменений "опустятся ниже" – он все еще будет работать. Такой селектор может быть более "стабильным" и "живучим".

Лучший вариант выбрать между двух невозможно, все зависит от проекта ;) Но можно начать с второго, а там видно будет ;)

Другие примеры ошибок...

```
elements("#todo-list li").shouldHave(exactTexts("a", "b", "c"));
element("li:nth-child(2) .toggle").click();
element(".completed").shouldHave(exactText("b"));
elements("#todo-list [class|=ember-view]").shouldHave(exactTexts("a", "c"));
```

Определившись с локатором для поиска "лист-айтема" (`li`) списка задач (`todo-list`):

```
elements("#todo-list li").shouldHave(exactTexts("a", "b", "c"));
```

... уже в следующей строке кода для отображания того самого элемента списка мы используем более другой, пускай более короткий, но "неконсистентный" и менее информативный локатор:

```
element("li:nth-child(2) .toggle").click();
```

... а теперь мы вообще забыли хоть как то упомянуть в локаторе о том что мы работаем с задачами:

```
element(".completed").shouldHave(exactText("b"));
```

... и последней вишенькой на тортике, мы решили "для разнообразия" в этот раз опустить `li`

```
elements("#todo-list [class|=ember-view]").shouldHave(exactTexts("a", "c"));
```

А здесь кто-то решил побеспокоится о других разработчиках которые этот код не писали, что-бы сразу как только они открыли код упавшего теста - глаза сразу увидели сквозь весь код где в каких строчках ведется работа с одними и теми же сущностям - `todo-list li` :

```
elements("#todo-list li").shouldHave(exactTexts("a", "b", "c"));
element("#todo-list li:nth-child(2) .toggle").click();
element("#todo-list li.completed").shouldHave(exactText("b"));
elements("#todo-list li*[class|=ember-view]*").shouldHave(exactTexts("a",
"c"));
```

При этом одна ошибка все еще осталась в последней строке (виделенная курсивом), но о ней в другой раз;)

Фэншуй

Есть в программировании вещи, которые могут показаться неоправданными усилиями ради какого-то "сомнительного фэншуя". Например "отступы в коде", "дополнительные пустые строчки", подбор читабельных понятных имен, и т. д. Но, на самом деле, чем больше растет количество написанного кода в проекте, тем он более становится похожим на сложный организм человека, в котором при проведении операций хирургу непростительно быть "неаккуратным", да и операции проще проводить в теле с более-менее хорошим состоянием. Поэтому стоит задуматься о том, чтобы выработать у себя некоторые "задротские но здоровые привычки" ;)

Фэншуй. Неравномерные отступы в коде

```
public class TodoMvcTest {

    @Test
__void completesTask() {
        open("http://todomvc.com/examples/emberjs");

        //...code

        __//...code
  _}
__}
```

Отступы в коде помогают отобразить его вложенную структуру – выделить соответвующие блоки кода, которые отображают определенный контекст, например: тест-сьют (тест-класс), тест-кейс (тест-метод), и т. д. Если же не следить за правильными отступами – код станет плохо читаемым, что сильно затруднит его понимание и поддержку.

Решение...

```
public class TodoMvcTest {

    @Test
    void completesTask() {
        open("http://todomvc.com/examples/emberjs");

        //...code

        //...code
    });
});
```

Фэншуй. Лишние пустые строки

```
public class TodoMvcTest {


-
    @Test
    void completesTask() {
        open('http://todomvc.com/examples/emberjs');

        //...code

        //...code
-
    }
-
-
}
```

Лишние пустые строки, на первый взгляд, достаточно безобидны, но если их использовать "с умом" – то с их помощью можно также выделять структуру в коде, например – блоки связанных шагов и проверок в коде тест-метода (~ "тест-кейса"). Да и основные части кода, такие как методы – также удобно отделять друг от друга пустыми строками, для "усиления" выразительности кода. И если где-то их использовать "с умом", а где-то "как попало", то от "усилий ума" толку не будет никакого;)

Решение...

```
public class TodoMvcTest {

    @Test
    void completesTask() {
        open('http://todomvc.com/examples/emberjs');

        //...code

        //...code
    }
}
```

Задание: Selenide и XPath селекторы

Особенность Selenide в том, что при его использовании, обычно, не нужны сложные CSS или XPath селекторы. Достаточно самых простых локаторов, которые ищут элементы по атрибутам, например:

```
element(byName("q")) // ...

element(byId("new-todo")) // ...
//или еще проще для последнего примера:
element("#new-todo") // ...
```

Но чтобы прочувствовать все это на собственном опыте, да еще и закрепить знания в XPath селекторах, твоим заданием будет реализовать сценарий из раздела "Selenide в действии":

```
@Test
void completesTask() {
    // open TodoMVC page

    // add tasks: "a", "b", "c"
    // tasks should be "a", "b", "c"

    // toggle b
    // completed tasks should be b
    // active tasks should be a, c
}
```

для приложения TodoMVC (emberjs version), не используя при этом следующих методов Selenide:

- `SelenideElement#find` (то есть `element(".parent").find(".child")`) или `SelenideElement#$`
- `Element#findAll` (то есть `element(".parent").findAll(".child")`) или `SelenideElement#$`
- `ElementsCollection#(int index)` (то есть `elements(".item").get(1)`)

- `ElementsCollection#first`
- `ElementsCollection#findBy(Condition condition)`
- `ElementsCollection#filterBy(Condition condition)`
- и других им подобных ;)

По сути, все, чем можно пользоваться - это:

- `element(...).*` , где `*` - это действия над элементом или ассерты (`should` / `shouldBe` / `shouldHave`)
- `elements(...).should(...)` (или с `shouldBe` / `shouldHave`)
- и XPath селекторами для составления локаторов, которые можно передавать просто как строки, например: `browser.element('//*[@name="q"]')`

Цель - написать как можно более близкий по аналогии к реализации на Selenide код (из раздела "Selenide в действии"). Например, если в версии Selenide мы используем поиск элемента по тексту среди других элементов - то нужно построить полностью аналогичный XPath селектор;)

Все еще, как и для задания о CSS, не спеши создавать переменные и методы для хранения/переиспользования локаторов либо тестовых данных. Мы к этому еще вернемся в следующиих заданиях. Сейчас важно двигаться шаг за шагом, детально прорабатывая темы на которых мы фокусируемся в каждом из уроков. Единственное, что некоторые xpath локаторы будут настолько чрезмерно сложными, что все таки будет неплохо создать некоторые универсальные вспомогательные методы для их построения. Если тебе это не понадобиться, то и славно, но в решении к заданию ты найдешь соответствующие примеры и сможешь сравнить их со своим решением ;)

Дополнительные материалы:

- Рецепты CSS и XPATH
- Видео от automated-testing.info о том, как писать локаторы (CSS, XPath)

В особо трудную минуту, можно подглядывать за подсказками в следующую главу с FAQ. В нем же можно будет найти список частых ошибок по которому проверить свое финальное решение;) Удачи!

Решение: Список частых вопросов

Где лучше сохранить вспомогательный методов для построения сложных xPath локаторов? (Стратегия сохранения методов)

Программирование очень похоже на реальную жизнь, серьезно:) В нем нет никаких секретов, которые бы не относились к обычному быту. Любая хорошая домохозяйка, которая умеет убираться и превращать хаос в порядок, у которой на кухне все по "феншую" – имеет все возможности стать прекрасной программисткой! А все потому, что одна из главных стратегий в программировании – **"раскладывать яйца по своим корзинам"**, или **"раскладывать чашки/тарелки по своим полочкам"**.

Разные методы, но в небольшом количестве, в самом самом начале? – допустимо один класс.

Как и в реальной жизни, в программировании, в самом самом начале, "яиц" может быть не так много, чтобы раскладывать их по разным корзинам. Если у нас всего одно гусиное яйцо, одно куриное белое, и одно куриное красное, и кроме яиц больше никаких других продуктов нет – нам незачем под каждый "вид" выделять свою корзину. Весь этот "натюрморт" будет выглядеть достаточно "феншуйно" в одной корзине, и по надобности – будет достаточно удобно сразу бежать к одной корзине, а не перебирать множество.

Кардинально отличные по использованию методы? – отдельный класс.

При этом, если вдруг у нас появляется что-то кардинально отличное от яиц, например столовые приборы – наверное, таки сразу стоит им выделить отдельный ящичек, или хотя бы "стаканчик", если их немного.

Много методов одного вида? – отдельный класс

Как только яиц определенного типа будет становиться много – уже станет сложней в одной куче находить нужные, и тут уже понадобится отдельная корзина, например, под куриные яйца.

Очень много методов одного вида? – выделяем подвиды и раскладываем по отдельным модулям/классам, которые группируем в выделенную под них под-папку

Как только яиц определенного вида станет так уж много, что одной корзины им будет маловато (будет сложно копаться, и быстро находить то что нужно) – придется выделить еще парочку, и чтобы как-то "организовать и структурировать" такие яйца – будет удобно выделить их подвиды – и хранить, например, красные яйца отдельно от белых, большие от маленьких, более свежие от менее свежих... И, конечно же, ни одна хозяйка не станет раскладывать корзины с яйцами по разным углам на кухне – она положит их все в одно место.

Также и в программировании стоит группировать классы разных подвидов одного вида внутри одного пакета. В джава-проектах обычно, кроме пакетов, для структурирования кода есть еще два отдельных "шкафа" – папка `src/main` и папка `src/test` – по которым, соответственно, можно раскладывать инструментарий соответствующего контекста использования. Все, что касается тестирования продукта – идет в `src/test`, все, что касается самого продукта – в `src/main`. Если же проект только о тестировании, то в `src/main` обычно складывают ту часть проекта, которую можно будет переиспользовать в других проектах, и, возможно, даже можно будет вынести в отдельную независимую библиотеку. Еще часто в таких "тест-проектах" в `src/main` складывают не сам продукт, а его "модель" (в контексте тестирования пользовательского интерфейса – это могут быть классы, описывающие объекты-страницы веб-приложения, так называемые PageObjects).

Также, как и в быту уже есть общепринятые в употреблении слова для обозначения некоторых мест хранения, так и в программировании, например, папка с классами с методами, которые могут потенциально быть использованы не только в текущем проекте, но и в других – могут называть `common`, или `core`, если набор таких классов уже больше похож на потенциальную библиотеку, решающую определенные проблемы, а не просто набор инструментов.

Важность имен

Хорошая хозяйка также подготовит понятные таблички с надписями для каждой корзинки, чтобы если ее не будет дома, другие члены семьи смогли разобраться, что, где и как. Таблички будут однозначно описывать то, что в них лежит, будут понятными и читабельными для всех членов семьи, написанные на "общем для всех языке", использовать "общепринятые термины". Прочтя надпись, должно быть сразу понятно, что лежит внутри. Надписи также будут по возможности достаточно краткими. Ни одна хозяйка на полке с приправами не будет на каждой из баночек

писать: "Приправа: перец", "Приправа: карри", "Приправа: тмин" – и так ведь понятно что все, что лежит на полочке с надписью "приправы" – приправы и есть, достаточно на каждой из баночек просто написать название приправы: "Перец", "Карри", "Тмин"...

Как в XPATH искать по одному CSS классу а не по всему значению соответствуеющего аттрибута?

CSS-селекторы хотя и менее универасальны чем XPATH селекторы, но более лаконичны и удобны, например поиск по одному css классу `toggle` реализуется очень просто:

```
.toggle
```

В случае XPath же нам придется извратиться, чтобы достичь той же цели:

```
element(byXpath("//*[contains(concat(' ', normalize-space(@class), ' '), '
toggle ')]")).click();
```

Можно улучшить ситуацию, создав дополнительный "хелпер-метод" (helper-method):

```
// src/main/java/com/taotas/selenideintro/common/selectors/X.java

//...

public class X {

    public static String hasCssClass(String value) {
        return "contains(concat(' ', normalize-space(@class), ' '), ' " +
value + " ')";
    }
}


element(byXpath("//*[" + x.hasCssClass("toggle")" + "])).click();
```

(в даном случае дополнительный хелпер, как видно, вынесен в отдельный класс, на вопрос "зачем?" - отвечает следующий пункт в FAQ: Где лучше сохранить вспомогательный метод для построения сложных xPath локаторов? (Стратегия сохранения методов))

Выше мы чуть пожертвовали читабельностью в имени класса в пользу лаконичности в использовании (`X` вместо `XPath` или `XPathHelpers`). Мы смогли позволить себе это сделать, исходя из того, что `X` – вероятно известное или интуитивно понятное сокращение в контексте поиска по XPath (в той же библиотеке JQuery есть команда `$x` , которая ищет элемент именно по XPath). Вспомни цитату из "стратегии хороших домохозяек о важности имен":

> Таблички будут однозначно описывать то, что в них лежит, будут понятными и читабельными для всех членов семьи, **написанные на "общем для всех языке", будут использовать "общепринятые термины"**

Как найти элемент из списка однотипных элементов по нужному тексту его внутреннего элемента (XPath)?

Первый же результат поиска в Google по

> xpath find element by text of its inner element

даст ответ ;)

Если не помогло...

Вся идея в том, что любое "уточнение поиска элемента в XPath селекторах" прописывается в квадратных скобках после элемента, например, для (следующий пример взят с stackoverflow):

```
<list>
   <book>
      <author>
         <name>John</name>
         <number>4324234</number>
      </author>
      <title>New Book</title>
      <isbn>dsdaassda</isbn>
   </book>
   <book>...</book>
   <book>...</book>
</list>
```

следующим образом можно уточнять, какая именно книга нужна:

```
//book[author/name='John']
```

или явно указывая, что ищем от "текущего места":

```
//book[./author/name='John']
```

или без точного полного пути (ищем элемент name на любой глубине вложенности внутри элемента book):

```
//book[.//name='John']
```

или без привязки к тегу внутреннего элемента:

```
//book[.//text()='John']
```

Или используя сокращенный синтаксис (хотя и менее читабельный):

```
//book[.//.='John']
```

Ниже приведены примеры, актуальные для домашнего задания...

версия в стиле Selenide:

```
elements("#todo-list li").findBy(exactText("b")).find(".toggle").click();
```

версия с XPath (точный аналог, с точной альтернативой поиска по css классу):

```
element(byXpath("//*[@id='todo-list']//li[.//text()='b']//*[contains(concat('
', normalize-space(@class), ' '), ' toggle ')]")).click();
```

или так, для прощего понимания:

```
element(byXpath(
    "//*[@id="todo-list"]//li" +
    "[.//text()='b']" +
    "//*[contains(concat(' ', normalize-space(@class), ' '), ' toggle ')]"
)).click();
```

версия с XPath (упрощенная, не полний аналог версии на Selenide - будет работать, потому что у искомого элемента только один css class – "toggle", если бы было больше, а так в будущем может и случиться, то приидеться использовать предыдущую версию):

```
element(byXpath("//*[@id='todo-list']//li[.//text()='b']//*
[@class='toggle']")).click();
```

Решение: Список частых ошибок

Большинство вопросов и ошибок для решений задания "Selenide и CSS селекторы" будут актуальны и для текущего задания "Selenide и XPath селекторы". Поэтому обязательно просмотри их в контексте проверки твоего решения;)

Избыточность имен

Допустим у нас есть код как решение проблемы сложного xpath селектора для поиска по css-классу:

Пример 1

```
// src/main/java/com/taotas/selenideintro/common/selectors/XPathSelectors.java

//...

public class XPathSelectors {

    public static String filterByCssClass(String value) {
        return "[contains(concat(' ', normalize-space(@class), ' '), ' " +
value + " ')]";
    }

    // ...
}
```

Пример 2

```
// src/main/java/com/taotas/selenideintro/common/selectors/XPathFilters.java

//...

public class XPathFilters {

    public static String filterByCssClass(String value) {
        return "[contains(concat(' ', normalize-space(@class), ' '), ' " +
value + " ')]";
    }
```

```
    // ...
}
```

Налицо избыточность в именах класса или метода...

Вспомни правило хороших домохозяек о важности надписей;) Там есть фраза:

> Надписи также будут по возможности достаточно краткими. Ни одна хозяйка на полке с приправами не будет на каждой из баночек писать: "Приправа: перец", "Приправа: карри", "Приправа: тмин" – и так ведь понятно, что все, что лежит на полочке с надписью "приправы" – приправы и есть, достаточно на каждой из баночек просто написать название приправы: "Перец", "Карри", "Тмин"...

Решения...

Пример 1

В первом примере незачем повторять в имени класса слово `Selectors` . Во-первых, файл с кодом уже лежит в пакете с таким именем. Во-вторых, ничего другого кроме "селекторов" в контексте использования xPath в нашем проекте быть не может, и так очевидно что речь идет о хелперах для составления селекторов.

```
// src/main/java/com/taotas/selenideintro/common/selectors/XPath.java

//...

public class XPath {

    public static String filterByCssClass(String value) {
        return "[contains(concat(' ', normalize-space(@class), ' '), ' " +
value + " ')]";
    }

    // ...
}
```

Пример 2

Во втором примере слово `filter` повторяется как в имени класса так и метода. Получается "масло маслянное", такая детализация излишня, она только усложняет-удлиняет код:

```
element(byXpath("//*" + XPathFilters.filterByCssClass("toggle"))).click();
```

Достаточно упомянуть только в одном месте, например так

```
// src/main/java/com/taotas/selenideintro/common/selectors/XPath.java

//...

public class XPath {

    public static String filterByCssClass(String value) {
        return "[contains(concat(' ', normalize-space(@class), ' '), ' " +
```

```
value + " ')]";
    }

    // ...
}
```

или так:

```
// src/main/java/com/taotas/selenideintro/common/selectors/XPathFilter.java

//...

public class XPathFilter {

    public static String byCssClass(String value) {
        return "[contains(concat(' ', normalize-space(@class), ' '), ' " +
value + " ')]";
    }

    public static String byNoCssClass(String value) {
        //...
    }

    // ...
}
```

Последняя версия более оптимизирована, так как уже упомянув слово "filter" в имени класса - не приидется повторять его каждый раз в именах методов при их определении. При использовании - контекст "filter" все равно будет задан именем класса:

```
...xPathFilter.byCssClass("toggle")
//...
...xPathFilter.byNoCssClass("complete")
```

В целом, даже такие решения не до конца оптимальны и универсальны. Есть и другие потенциальные проблемы в этом коде. Но об этом - в следующих пунктах этого FAM ;)

Неясные неполные имена вспомогательных методов

Допустим, есть код, как решение проблемы сложного xpath селектора для поиска по css-классу:

```
// src/main/java/com/taotas/selenideintro/utils/XPath.java

//...

public class XPath {

    public static String byCssClass(String value) {
        return "contains(concat(' ', normalize-space(@class), ' '), ' " +
value + " ')";
    }

    // ...
}


//...
element(byXpath(
    "//*[@id='todo-list']/li[2]//*[" + XPath.byCssClass("toggle") + "]"
)).click();
```

Вспоминаем правило хороших домохозяек о важности надписей;) В нем есть следующая фраза:

> Прочтя надпись, должно быть сразу понятно, что лежит внутри.

Теперь вопрос. Понятно ли по имени нашей "корзины" с хелпером – `XPath` – что внутри лежит хелпер-метод который можно использовать именно внутри квадратных скобок xpath-селектора? Возможно, этот хелпер можно использовать так:

```
//...
    "//*[@id='todo-list']/li[2]//*" + XPath.byCssClass("toggle")
//...
```

или так:

```
"//*[@id='todo-list']/li[2]//" + XPath.byCssClass("toggle")
```

или даже так:

```
element(byXpath(XPath.byCssClass("toggle"))).click();
```

o_O ?

Да, с одной стороны мы, можно сказать, следовали одной из стратегий все той же домохозяйки:

> *Разные методы, но в небольшом количестве, в самом самом начале? – допустимо один класс.* Как и в реальной жизни, в программировании, в самом самом начале, "яиц" может быть не так много, чтобы раскладывать их по разным корзинам...

Но получается в этом случае, следование только этой стратегии - недостаточно, ведь с имени нельзя полностью понять контекст использования. Все это конечно относительно. "Очевидность" имен - в любом случае относительна. В какой то из команд такого имени будет достаточно. Но давай в нашем случае попробуем хотя-бы слегка улучшить ситуацию ;)

Решение...

Получается, нам нужно как то отобразить в полном имени метода `XPath.byCssClass` то, что он должен быть использован внутри скобок:

```
"//*[@id='todo-list']/li[2]//*[" + XPath.byCssClass("toggle") + "]"
```

Скобки вместе с тем что у них внутри - по сути есть "уточнением селектора" или "фильтром элементов по пути отображенному в селекторе ранее". А фильтр этот - может состоять из одного или более "штук" которые в программировании принято называть "предикатами" - "методами возвращающими `true` или `false` ". Получается наш `byCssClass` - и есть таким методом, то есть предикатом (а не фильтром например, который бы требовал наличия квадратных скобок в контексте использования в xPath селекторе). Понимая эту терминологию мы можем отобразить ее в имени нашего метода:

```
"//*[@id='todo-list']/li[2]//*[" + XPath.predicateCssClass("toggle") + "]"
```

или так:

```
"//*[@id='todo-list']/li[2]//*[" + XPathPredicate.cssClass("toggle") + "]"
```

или так:

```
"//*[@id='todo-list']/li[2]//*[" + XPath.Predicate.cssClass("toggle") +
"]"
```

Последний способ - может быть самым универсальным, учитывая то, что набор наших "яиц в корзине" может быть достаточно разнообразным:

```
public class XPath {

    public static class Predicate {

        public static String cssClass(String value) {
            return "contains(concat(' ', normalize-space(@class), ' '), ' " +
value + " ')";
        }
```

```
        public static String not(String predicate) {
            return "not(" + predicate + ")";
        }
    }

    public static String filteredBy(String predicate) {
        return "[" + predicate + "]";
    }

    public static class FilterBy {

        public static String cssClass(String value) {
            return XPath.filteredBy(XPath.Predicate.cssClass(value));
        }

        public static String noCssClass(String value) {
            return
XPath.filteredBy(XPath.Predicate.not(XPath.Predicate.cssClass(value)));
        }
    }
}
```

... что позволит нам достичь цели теперь и без явного заворачивания в квадратные скобки и сделать код еще чуть "чище" и читабельней:

```
    "//*[@id='todo-list']/li[2]//*" + XPath.FilterBy.cssClass("toggle")
```

С такими темпами, можно пойти еще дальше... Если на каком то проекте мы вынуждены использовать часто XPath селекторы а тесты стараются писать не опытные автоматизаторы, то может стоит им помочь и еще больше расширить набор таких хелперов для составления XPath-селекторов:

```
element(byXpath(
    XPath.all + XPath.FilterBy.id("todo-list") +
    XPath.child("li") + XPath.FilterBy.index(2) +
    XPath.descendant() + XPath.FilterBy.cssClass("toggle")
)).click();
```

Можешь попробовать реализовать все эти хелперы ради тренировки ;)

А я пока подниму еще одну тему... Кому то может показаться, что хотя синтаксис использования наших хелперов получился абсолютно читабельным - он при этом достаточно громоздкий, и возможно для кого-то - чересчур громоздкий. В какой то довольно сильной команде, могут отдавать предпочтение более лаконичному стилю... В таком случае, мы можем попробовать упростить наш стиль до чего-то такого:

```
element(byXpath(
    "//*[@id='todo-list']/li[2]//*[" + X.hasCssClass('toggle') + "]"
)).click();
```

Мы сделали имя класса более лаконичным, исходя из того что в нашей команде такое сокращение кажется довольно натуральным и очевидным, и никак не конфликтует ни с какими другими именами классов. И мы убрали вложенность классов, добавляя к имени методов, которые являются предикатами - префикс, указывающий на то, что они предикаты. В программировании как раз и есть общепринятой практикой добавлять к методам возвращающим `boolean` значения (`true` или `false`) - префиксы типа `is` / `has` / `have` .

Таким образом мы и реализацию упростили, сделав ее структуру менее вложенной:

```
public class X {

    public static String hasCssClass(String value) {
        return "contains(concat(' ', normalize-space(@class), ' '), ' " +
value + " ')";
    }

    public static String not(String predicate) {
        return "not(" + predicate + ")";
    }

    public static String filteredBy(String predicate) {
        return "[" + predicate + "]";
    }
}
```

И упростили использование. Ведь, поскольку вложенность только одноуровневая - функция автодополнения кода в редакторе будет подсказывать сразу весь список хелперов после введения `X.` , что позволит быстрее находить пользователям то что им нужно, ведь они получают доступ сразу ко всему...

При этом методы типа:

```
public static String filterByCssClass(String value) {
    return X.filteredBy(X.hasCssClass(value));
}

public static String filterByNoCssClass(String value) {
    return X.filteredBy(X.not(X.hasCssClass(value)));
}
```

... можно уже и не добавлять, учитывая например то "команда наша сильная" и ей незачем плодить кучу хелперов, достаточно и чистого использования кода вида `X.filteredBy(X.not(X.hasCssClass(value)))` там где это нужно;)

Хрупкие строгие локаторы. Избыточность в путях - привязка к тегу

Ошибки, типичные и для задания о CSS-селекторах;)

Пример 1

```
element(byXpath("//input[@id='new-todo']")).setValue("a")
```

Пример 2

```
element(byXpath("//*[@id='todo-list']/li[2]//input[@class='toggle']"))
```

Пример 3

```
element(byXpath("//ul[@id='todo-list']/li[2]//*[@class='toggle']"))
```

Пример 4

```
element(byXpath("//*[@id='todo-list']/li[2]//input"))
```

Последний локатор еще более хрупкий чем предыдущие, поскольку совсем не использует уникальные атрибуты.

Решения...

Пример 1

```
element(byXpath("//*[@id='new-todo']"))
```

Пример 2

```
element(byXpath("//*[@id='todo-list']/li[2]//*[@class='toggle']"))
```

Пример 3

```
element(byXpath("//*[@id='todo-list']/li[2]//*[@class='toggle']"))
```

Пример 4

```
element(byXpath("//*[@id='todo-list']/li[2]//*[@class='toggle']"))
```

Двойные кавычки внутри двойных кавычек

Пример

```
element(byXpath("//*[@id=\"new-todo\"]"))
```

Всегда стоит помнить о читабельности кода, ведь это прямым образом влияет на поддержку этого кода в будущем, особенно людьми которые его не писали изначально. В даном случае "бэкслеши" явно ухудшают читабельность, а ведь их можно избежать используя другой тип "внутренних" кавычек по отношению к "внешним", который также будет валидным;)

Решение...

```
element(byXpath("//*[@id='new-todo']"))
```

Использование `position()` для поиска элемента по индексу

```
element(byXpath("//*[@id='todo-list']/li[position()=2]//*[@class='toggle']"))
```

Всегда стоит стремиться к лаконичности, если это не ухудшает читабельности. В примере выше использование `position()=` избыточно. В XPath есть сокращенный синтаксис, который полностью очевидный всем, поскольку повсеместно используется в программировании.

Решение...

```
element(byXpath("//*[@id='todo-list']/li[2]//*[@class='toggle']"))
```

Поиск элемента по индексу вместо поиска по тексту

```
element(byXpath("//*[@id='todo-list']/li[2]//*[@class='toggle']"))
```

Во-первых, в задании было указано для решения на XPath использовать код, как можно более близкий к версии локаторов в стиле Selenide:

```
elements("#todo-list>li").elementBy(exactText("b")).element(".toggle")
```

Во-вторых, выбор элемента среди списка элементов именно по тексту имеет два преимущества:

- код будет более читабельным и информативным – не нужно будет лишний раз задумываться – какая там у нас была задача под индексом `2`. Это было бы особенно актуально, если бы у нас в сценарии ранее были шаги, которые, например, удаляют некоторые задачи - соответственно индексы задач будут меняться со временем, что может еще больше запутывать.
- плюс код будет ближе к тому как взаимодействует с приложением пользователь. Если мы хотим "завершить" задачу как пользователь – мы ищем ее именно по тексту глазами в списке, а не помним, по какому номеру она была создана, и отсчитываем все задачи от начала, пока ищем нужную;)

Подсказки, как выбрать нужный элемент по тексту, есть в этом FAQ.

Решение...

```
element(byXpath("//*[@id='todo-list']/li[.//label='b']//*[@class='toggle']"))
```

или, возможно, чуть лучше, потому что "менее строго" (а вдруг label поменяется на какой-то другой элемент?):

```
element(byXpath("//*[@id='todo-list']/li[.//text()='b']//*[@class='toggle']"))
```

или, используя сокращенный синтаксис, хотя, возможно, чуть менее очевидный:

```
element(byXpath("//*[@id='todo-list']/li[.//.='b']//*[@class='toggle']"))
```

I Начало. Проверка Концепции

Шаблон первого теста

Итак, представим себе, что мы в самом начале пути построения автоматизации пользовательского интерфейса веб-приложения на проекте TodoMVC.

Обычно перед тем как серьезно браться за любую работу, стоит "попробовать как оно пойдет", познакомиться с тем, с чем мы будем иметь дело. Еще говорят: "проверить концепцию" или "реализовать Proof of Concept (POC)". Этим мы и займемся - напишем первый черновой тест, который покроет какие-то базовые части функционала, даст нам почувствовать то с чем м будем иметь дело, и понять стоит ли оно вкладывания усилий и каких именно.

Начнем со скелета нашего теста:

```
package com.taotas.lpoctest;

import org.junit.jupiter.api.Test;

public class PocTest {

    @Test
    public void basicTestCase(){

    }
}
```

Тест черновой, пробный - поэтому пока над именами заморачиваться не будем:)

Итак, перед тем как сделать что-то полезное, нужно загрузить страницу приложения:

```
package com.taotas.lpoctest;

import org.junit.jupiter.api.Test;
```

```
import static com.codeborne.selenide.Selenide.open;

public class PocTest {

    @Test
    public void basicTestCase(){
        open("http://todomvc4tasj.herokuapp.com/");
    }
}
```

"Быстрые в реализации" тесты. Читабельность, очевидность и лаконичность

Теперь пора поработать с задачами, но для начала нужно хотя бы одну создать.

С помощью команды `element` - найдем поле ввода текста новой задачи по CSS-селектору: нам нужен элемент `input` с `id="new-todo"`

```
//...
import static com.codeborne.selenide.Selenide.$;
import static com.codeborne.selenide.Selenide.open;
//...

        open("http://todomvc4tasj.herokuapp.com/");
        element("input#new-todo")
```

И сразу вспоминаем одно из главных правил тестов: *"тесты должны быть простыми"*.

> Это объясняется тем, что
>
> - тестов очень много
> - их пишут разные люди
> - тесты часто требуют внимания людей и их быстрой реакции, потому что падают
> - тест, вероятно, будет прочитан человеком, который его не писал

Поэтому вариант - писать тесты так, чтобы человек сидел и долго думал, что да как тут автор имел в виду - не проходит.

Тесты должны быть понятны с одного взгляда, иначе их поддержка будет занимать очень много времени, и автоматизация перестанет быть эффективной.

Это правило можно обобщить как: *"тесты должны быть **быстрыми в реализации**"*, на их разработку и поддержку реализации должно идти мало времени.

Близкий друг и напарник читабельности - лаконичность. Главное, чтобы они не ссорились и не подставляли друг друга. Еще одно правило можно сформулировать так: *"лаконичность для читабельности"* или другими словами - *"всегда стремится к лаконичности, если это не ухудшает читабельность"*.

Последний принцип мы и используем:

```
element("#new-todo")
```

Упоминание тега `input` нашего элемента излишне. Чтобы селектор был однозначным, достаточно указать id, значение которого настолько информативно, чтобы с одного взгляда было понятно, что ищется элемент для создания новой задачи.

Теперь добавим в поле текст новой задачи:

```
element("#new-todo").append("do something")
```

И нажмем `Enter`, чтобы задача создалась:

```
element("#new-todo").append("do something").pressEnter();
```

Как видим, код получился довольно читабельным, связным и лаконичным.

Можно поблагодарить Selenide за дизайн методов-действий - большинство из них возвращают объект `SelenideElement`, над которым действие совершалось, поэтому мы можем строить такие цепочки методов.

Но не все методы-действия такие - у `SelenideElement` есть набор методов, которые отображают действия из чистого Selenium (`click` , `sendKeys` , `clear` , и т.д.). Они, как и в чистом Selenium - ничего не возвращают (вместо типа возвращаемого значения имеют `void` в сигнатуре метода).

Вот аналогичный код с использованием таких "родных методов" Selenium:

```
element("#new-todo").sendKeys("do something");
element("#new-todo").sendKeys(Keys.ENTER);
```

или:

```
element("#new-todo").sendKeys("do something" + Keys.ENTER);
```

Как видим, Selenide дает возможность симулировать нажатие часто используемых клавиш, таких как `ENTER` - с помощью удобных читабельных методов (`pressEnter()`):

```
element("#new-todo").append("do something").pressEnter();
```

"Быстрые в выполнении" тесты. Уровень сложности тестовых данных

Обратим внимание на то, какие тестовые данные мы используем. Можно было ведь создать задачу с более сложным текстом с целью покрыть еще больше кейсов:

```
element("#new-todo").append(" lk@i\nиⷾ\"j;k*7 \n").pressEnter();
```

Но тут вступает второе главное правило тестов: "тесты должны быть быстрыми в выполнении". Ведь даже если наши тесты требуют мало времени на написание и поддержку, но их выполнения приходится ждать очень долго - то эффективность таких тестов падает, много ли толку от тестов, от которых нужно ждать отклик целую ночь, в мире где компании типа Amazon или Google должны и умеют деплоить последние фичи по первому запросу?

Из этого следует, что "покрывать все возможные кейсы" через Selenium тесты не выгодно, потому что Selenium тесты - не быстрые сами по себе. Намного более эффективно оставить на Selenium тесты только самые высокоприоритетные функциональные кейсы, которые отображают основные "business flow" в приложении. А покрытие разных нюансов и аспектов функционала перенести на уровень юнит-тестов ("unit tests").

И поэтому же, обычно, в Selenium тестах для нас не важна сложность тестовых значений. И все еще важна их читабельность.

Получается, в нашем случае достаточно упростить имя задачи до:

```
element("#new-todo").append("1").pressEnter();
```

или использовать букву вместо цифры

```
element("#new-todo").append("a").pressEnter();
```

Что немного лучше, так как (забегая чуть наперед) в нашем приложении цифры используются уже для представления других сущностей - количество оставшихся незавершенных задач, количество задач в списке. И когда мы начнем реализовывать эти сущности в тесте - было бы неплохо, чтобы визуально значения "цифровых тестовых данных" отличались от значений "текстовых тестовых данных";)

Более быстрый ввод текста с помощью `Configuration.fastSetValue`

Стоит заметить, что ввод "длинных" строк типа `" lk@ï\nи₴\"j;k*7 \n"` занимает больше времении чем короткиих типа `"a"` , потому что комманда Selenide `append` построена на основе комманды Selenium WebDriver `sendKeys` , которая вводит целую строку посимвольно. При этом в Selenide есть способ как даже длинную строку ввести "быстро". Для этого можно использовать метод `setValue` вместе с дополнительной настройкой `Configuration.setValue` :

```
@Test
public void basicTestCase(){
```

```
        Configuration.fastSetValue = true;

        open("http://todomvc4tasj.herokuapp.com/");
        element("#new-todo").setValue(" lk@ï\nиѫ\"j;k*7 \n").pressEnter();
```

Дополнительная конфигурация `Configuration.fastSetValue = true;` укажет Selenide устанавливать значения полей ввода при вызове команды `setValue` с помощью JavaScript, что будет одинаково быстро для любого количества вводимых символов. Для нас такой "хак" только в пользу, ведь обычно нам не нужно тестировать работу полей редактирования по вводу текста - это часть уже протестирована тестировщиками браузера. Единственный нюанс в том, что "fastSetValue" не будет работать для `append`, у которого есть преимущество в том, что если мы решим добавлять больше одной задачи последовательно, то неявно мы будем проверять и то что после нажатия `Enter` поле очищается. А используя `setValue` мы этого не увидим, потому что эта команда не добавляет новый текст в конец, а заменяет полностью изначальный текст поля ввода. Поэтому пока что мы этот способ опустим, не будем спешить с решением в его пользу;) Тем более что для нас в любом случае важна читабельность и в `append` мы достигаем быстрого ввода из-за использования читабельных и "коротких" тестовых данных.

Проверки (Assertions)

Двигаемся дальше, тест-кейс - это не только шаги, это и проверки - сравнения актуального результата с ожидаемым.

Чтобы проверить, что создалась одна задача, найдем список всех задач с помощью команды `elements`:

```
//...
import static com.codeborne.selenide.Selenide.elements;
//...

        element("#new-todo").append("a").pressEnter();
        elements("#todo-list li")
```

Возможно, стоит уточнить CSS-селектор "#todo-list li" до более строгого

```
        element("#new-todo").append("a").pressEnter();
        elements("#todo-list>li")
```

Так как, возможно, в будущем появятся другие элементы с тегом `li` внутри элементов задач... Но это не значит, что всегда нужно подбирать "строгие" локаторы. Все зависит от проекта, от того, существенны ли для нас риски того, что "нестрогий локатор" может перестать находить то, что нужно, или наоборот - существенна ли польза от "более слабого", но более универсального локатора.

Далее проверим, что список найденных задач должен иметь размер `1` :

```
//...
import com.codeborne.selenide.CollectionCondition;
//...

        element("#new-todo").setValue("a").pressEnter();
        elements("#todo-list>li").shouldHave(CollectionCondition.size(1));
```

Условие-condition `size` есть "статическим" методом, создающим объект "условия проверки" по нужным нам параметрам. Поэтому, если использовать статический импорт java то можно получить более читабельный код (помним о важности читабельности и лаконичности):

```
//...
import static com.codeborne.selenide.CollectionCondition.size;
//...

        element("#new-todo").append("a").pressEnter();
        elements("#todo-list>li").shouldHave(size(1));
```

Полнота проверок

Конечно же, проверить размер списка задач недостаточно (проверка будет не полной), более важно - проверить, что создалась задача с указанным текстом:

```
//...
import com.codeborne.selenide.Condition;
```

```
import static com.codeborne.selenide.CollectionCondition.size;
//...

        element("#new-todo").setValue("a").pressEnter();
        elements("#todo-list>li").get(0).shouldHave(Condition.exactText("a"));
        elements("#todo-list>li").shouldHave(size(1));
```

В этом случае мы использовали "обычный кондишен" (условие для проверки элементов по отдельности), не "коллекшен кондишен" (условие, использующееся для проверки коллекций элементов, как в случае с `size(1)`). Такие кондишены живут в виде статических методов (или полей объекта, если у кондишена нет никаких параметров) в отдельном классе - `Condition`. Все еще есть смысл использовать статический импорт для более читабельного и лаконичного кода:

```
//...
import static com.codeborne.selenide.Condition.exactText;
import static com.codeborne.selenide.CollectionCondition.size;
//...

        element("#new-todo").append("a").pressEnter();
        elements("#todo-list>li").get(0).shouldHave(exactText("a"));
        elements("#todo-list>li").shouldHave(size(1));
```

Есть и другие места, где можно улучшить читабельность - `first()` вместо `get(0)`:

```
        element("#new-todo").append("a").pressEnter();
        elements("#todo-list>li").first().shouldHave(exactText("a"));
        elements("#todo-list>li").shouldHave(size(1));
```

Ну и самое сладкое на десерт:

```
//...
import static com.codeborne.selenide.CollectionCondition.exactTexts;
//...

        element("#new-todo").append("a").pressEnter();
        elements("#todo-list>li").shouldHave(exactTexts("a"));
```

Условие `exactTexts` включает в себя и проверку размера, и проверку текста, и проверку порядка указанных текстов. Последнее было бы особенно актуально, если бы мы создали более чем одну задачу:

```
element("#new-todo").append("a").pressEnter();
element("#new-todo").append("b").pressEnter();
elements("#todo-list>li").shouldHave(exactTexts("a", "b"));
```

Но не будем пока спешить, оставим только одну:

```
element("#new-todo").append("a").pressEnter();
elements("#todo-list>li").shouldHave(exactTexts("a"));
```

"Флоу" End-to-End-теста. Упрощенный тест-план

Закончив с первыми пробами, теперь стоит определить примерный план действий. Поскольку идея нашего теста - "попробовать как взлетит автоматизация нашего приложения", то имеет смысл проверить это на всех основных, самых часто используемых действиях пользователя. Пользователь также в реальной жизни использует несколько функций приложения вместе, в определенных последовательностях, которые еще называют "workflow" или "business flow". И конечно стоит и тест наш построить исходя из идеи автоматизации таких более приближенных к реальности сценариев. Такие автотесты, которые проверяют совместную работу нескольких фич называются "интеграционными" и когда они отображают реальный "флоу" пользователя при работе с приложением - "End to End" тестами (допустимо сокращение "E2E").

Получается, нам нужно накидать упрощенный "тест-план": определить основные операции в нашем приложении, уточнить приоритеты, и придумать порядок покрытия этих операций. Это и собственно реализация нужных шагов - и будет твоим следующим заданием ;)

Задание: POC-тест

Напиши один черновой "End-to-End" тест что-бы "проверить концепцию" (Proof Of Concept) для веб-приложения TodoMVC (версия https://todomvc4tasj.herokuapp.com)], так сказать, "прощупать почву" на старте нового проекта по автоматизации. Постарайся покрыть основные части функционала с точки зрения "Smoke Acceptance Testing". Исходи из того, что главным для заказчика является основной функционал менеджмента задач в виде набора основных действий над ними (CRUD i.e. Create Read Update Delete). Учти, что некоторые из "CRUD" действий проявляются в разных "фичах" ("features") приложения ;)

В процессе написания теста тебе придется определить весь спектр возможных операций в TodoMVC, выбрать те из них, которые стоит проверить (исходя из условий, описанных выше, и здравого смысла), а также подобрать оптимальный порядок тестирования этих "избранных" операций. Конечно, это всего лишь POC тест, и в реальной жизни обо всем этом думать стоит, но не стоит прям сильно заморачиваться, ведь мы должны успеть как можно быстрее. Но можешь все-таки подумать как следуют, ради тренировки, по сути, "построения и генерации тест-кейсов". Внимательно происследуй все возможности, которые есть у пользователя при работе с TodoMVC, занотируй ("notate") отдельно такой "упрощенный тест-план" и потом сможешь сверить его с решением.

Для реализации можешь использовать все возможности Java, JUnit5, Selenide;)

P.S. Принципы YAGNI (You ain't gonna need it) и Lean

Исходя из того, что это все-таки только POC-тест, игнорируй приниципп DRY, не создавай дополнительных методов и переменных, дабы как можно "быстрее" закончить задачу. Плюс селекторы довольно читабельные как для технически подкованных людей, поэтому код и так должен быть понятным, без использования дополнительных абстракций. Мы также еще не знаем, подвердит ли менеджмент идею написания автотестов, и сами не знаем насколько тяжело будет автоматизировать наше приложение. Этот подход - "не делать супер-улучшений до тех пор пока не

припечет" - отображает принцып - YAGNI (You Ain't Gonna Need It). Те же идеи упоминаются и в принципах Бережливой Разработки (смотри на вики: Lean Software Development). Цитируя википедию, вот о чем говорит один из принципов Lean:

- "Исключение потерь. Потерями считается всё, что не добавляет ценности для потребителя. В частности: **излишняя функциональность**; ..."