

Explorando APIs e bibliotecas Java

JDBC, IO, Threads, Java FX e mais



© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

“Às famílias Bertoldo, Ferreira e Turini.”

– Rodrigo Turini

Agradecimentos

Em primeiro lugar, gostaria muito de agradecer a você, o leitor. Foi para você que cuidadosamente escrevi esse livro, pensando sempre em como o conteúdo poderia ser aplicado em seu dia a dia. Espero atender e, quem sabe, superar as suas expectativas.

Para evitar consequências imprevisíveis, não posso deixar de agradecer à minha esposa Jordana. Sem seu valioso incentivo este livro não passaria de um plano. E também para pequena Katherine, que nesse momento provavelmente está chutando a sua barriga.

Não posso deixar de mencionar a família Silveira, que, pra mim, são os maiores incentivadores do mundo. Ao Paulo, pela presença, incentivo e *mentoring*. Ao Guilherme, pelas ideias, pareamentos e todo conhecimento compartilhado. E finalmente ao Sr. Carlos, pela cultura e inspiração diária.

Por fim, mas nem um pouco menos importante, a toda equipe da Caelum e Alura. Em especial ao Victor Harada, pois sem suas ideias, críticas e discussões, boa parte deste livro seria diferente.

Sumário

1	Introdução	1
1.1	O projeto e as tecnologias	1
1.2	Instalando e configurando o Eclipse	2
1.3	Download dos arquivos pro projeto	3
1.4	Acesse o código desse livro	4
1.5	Aproveitando ao máximo o conteúdo	5
1.6	Tirando suas dúvidas	5
2	Java FX	7
2.1	Nossa primeira App em Java FX	7
2.2	Configurando a livreria-base	11
2.3	Preparando nosso cenário	13
2.4	Uma listagem de produtos	18
3	Java IO	29
3.1	Entrada e saída de dados	29
3.2	Lendo um arquivo de texto	30
3.3	Lendo texto do teclado com System.in	34
3.4	Tornando tudo mais simples com Scanner	36
3.5	Saída de dados e o OutputStream	38
3.6	Escrita mais simples com PrintStream	42
3.7	Gerando um CSV de produtos	43
3.8	Botão de exportar produtos	48
3.9	Adicionando ações com setOnAction	49
3.10	JavaFx e Java 8	52

4	Banco de Dados e JDBC	57
4.1	Iniciando com MySQL	57
4.2	Criando a tabela de produtos	60
4.3	O pacote java.sql e o JDBC	61
4.4	Abrindo conexão com MySQL em Java	63
4.5	Listando todos os produtos do banco	66
4.6	Importando produtos de um dump	69
4.7	Para saber mais: Adicionando programaticamente	71
4.8	Qual a melhor forma de fechar a conexão?	75
4.9	O padrão de projeto DAO	78
5	Threads e Paralelismo	85
5.1	Processamento demorado, e agora?	86
5.2	Trabalhando com Threads em Java	87
5.3	O contrato Runnable	89
5.4	Threads com classes anônimas e lambdas	91
5.5	Exportando em uma thread separada	94
5.6	Um pouco mais sobre as Threads	99
5.7	Garbage Collector	101
5.8	Java FX assíncrono	103
5.9	Trabalhando com a classe Task	104
5.10	Código final com e sem lambdas	112
6	CSS no Java FX	117
6.1	Seu primeiro CSS no Java FX	118
6.2	Extraindo estilos pra um arquivo .css	120
7	JAR, bibliotecas e build	135
7.1	JAR	135
7.2	Gerando JAR executável pela IDE	136
7.3	Executando a livreria-fx.jar	138
7.4	Bibliotecas em Java	140
7.5	Documentando seu projeto com Javadoc	141

7.6	Automatizando build com Maven	146
7.7	Transformando nossa app em um projeto Maven	147
7.8	Adicionando as dependências com Maven	151
7.9	Executando algumas tasks do Maven	157
7.10	Adicionando plugin do Java FX	159
7.11	Maven na linha de comando	163
7.12	Como ficou nosso pom.xml	165
8	Refatorações	169
8.1	Refatoração	170
8.2	Os tão populares Design Patterns	175
9	Próximos passos com Java	179
9.1	Entre em contato conosco	180

CAPÍTULO 1

Introdução

1.1 O PROJETO E AS TECNOLOGIAS

Durante este livro, trabalharemos no projeto de uma livraria. A princípio, vamos criar uma listagem de produtos com *Java FX* e ao decorrer dos capítulos incrementaremos suas funcionalidades, passando pelas *APIs* de *IO*, *JDBC*, *Threads* e muito mais. Além disso, a todo o momento discutiremos sobre boas práticas da orientação a objetos e *design patterns*.

Ao final, o projeto deve ficar parecido com:



Nome	Descrição	Valor	ISBN
A Web Mobile	Programa para um mundo de m...	59.9	978-85-66250-23-7
Desbravando Java e OO	Um guia para o iniciante da ling...	59.9	123-45-67890-11-2
Google Android	crie aplicações para celulares	59.9	978-85-66250-02-2
HTML5 e CSS3	Domine a web do futuro	59.9	978-85-66250-05-3
JSF e JPA	Aplicações Java para a web	59.9	978-85-66250-01-5
Java 8 Prático	Novos recursos da linguagem	59.9	978-85-66250-46-6
Java SE 7 Programmer I	O guia para sua certificação	59.9	123-45-67890-11-2
Jogos em HTML5	Explore o mobile e física	59.9	123-45-67890-11-2
Lógica de Programação	Crie seus primeiros programas	59.9	978-85-66250-22-0
O Programador Apaixonado	Construindo uma carreira notável	59.9	978-85-66250-44-2
Test-Driven Development	Teste e Design no Mundo Real	59.9	123-45-67890-11-2
Thoughtworks Antologia	Histórias de aprendizado	59.9	123-45-67890-11-2
UX Design	Introdução e boas práticas	59.9	978-85-66250-48-0

Você tem R\$778.70 em estoque, com um total de 13 produtos.

Fig. 1.1: Aparência final do projeto deste livro

Essa listagem de livros é carregada de um banco de dados MySQL, utilizando a API de *JDBC*. O *build* e gerenciamento de dependências é totalmente automatizado, com uso do tão popular Maven. A ação de exportar os livros é feita em uma *Thread* diferente, que roda em paralelo com a *Thread* principal da aplicação.

Nosso código já usará a sintaxe e recursos do Java 8, como as famosas expressões lambdas. Mas não se preocupe, você não precisa conhecer Java 8 ou nenhuma outra tecnologia além de Java puro para continuar, cada novidade será muito bem detalhada. Pronto para começar?

1.2 INSTALANDO E CONFIGURANDO O ECLIPSE

Para começar, você precisará do Eclipse ou qualquer outra IDE de sua preferência. Utilizarei o Eclipse em meus exemplos, devido ao ganho de produtivi-

dade (que será mencionado em alguns pontos do livro), além de sua popularidade no mercado e instituições de ensino. Se quiser, você pode fazer o download do Eclipse em:

<https://www.eclipse.org/downloads/>

Para utilizar o Java FX no Eclipse, você também precisará do *e(fx)clipse*, que pode ser baixado pelo seguinte link, no qual você também encontrará um passo a passo para instalá-lo.

<http://www.eclipse.org/efxclipse/install.html>

Algumas outras opções

Além do Eclipse, existem diversas outras IDEs que podem ser utilizadas no desenvolvimento desse projeto. *Netbeans* e *IntelliJ IDEA* são algumas delas. O Netbeans já vem com suporte ao Java FX, sendo assim você não precisará instalar nada. Você pode ler mais sobre ele e fazer o download em:

<https://netbeans.org/>

Segundo as pesquisas mais recentes, logo depois do Eclipse, o *IntelliJ IDEA* é a IDE mais utilizada pelos desenvolvedores Java. Mas há uma questão: ela é paga. Você pode baixar uma versão *Community Edition* (gratuita), mas ela deixa um pouco a desejar. Se quiser experimentar a versão completa, tem como opção baixar um *trial* por 30 dias.

<https://www.jetbrains.com/idea/>

Apesar de utilizamos Eclipse nos exemplos desse livro, você pode usar a IDE de sua preferência. Fique à vontade em sua escolha.

1.3 DOWNLOAD DOS ARQUIVOS PRO PROJETO

No link a seguir, você encontrará todos os arquivos necessários para acompanhar ativamente os códigos desse livro. Ele inclui a base do projeto, os *jars* necessários para você fazer os exercícios, um arquivo de *dump* com alguns livros já cadastrados no banco, entre outros.

<http://goo.gl/CfzTLB>

DESBRAVANDO A ORIENTAÇÃO A OBJETOS

Se você já leu meu livro *Desbravando a Orientação a Objetos*, perceberá que esta é uma evolução do projeto que desenvolvemos por lá. Você pode continuar com seu próprio projeto, ou baixar o projeto já pronto se preferir.



Fig. 1.2: Desbravando Java e Orientação a Objetos

[http://www.casadocodigo.com.br/products/
livro-orientacao-objetos-java](http://www.casadocodigo.com.br/products/livro-orientacao-objetos-java)

1.4 ACESSE O CÓDIGO DESSE LIVRO

Todos os exemplos desse livro e também seu código final, em Java 7 e 8, podem ser encontrados no seguinte repositório:

- <https://github.com/Turini/livro-java>

Não deixe de escrever todos os códigos e exercitá-los durante os capítulos. Assim você pode tirar um proveito maior do conteúdo. Vá além do que é aqui sugerido, escreva novos testes, novos layouts e não deixe de compartilhar sua experiência conosco.

1.5 APROVEITANDO AO MÁXIMO O CONTEÚDO

Além de colocar todos os exemplos em prática, não deixe de consultar também a documentação de cada API e biblioteca que vamos utilizar. Eu sempre disponibilizo alguns links que podem ser úteis quando um novo assunto é introduzido.

Como nosso foco principal é Java, tenha sempre por perto a documentação da linguagem:

<http://docs.oracle.com/javase/8/docs/api/>

Se estiver devidamente configurado, você pode ver a documentação das classes e métodos pelo próprio Eclipse. Sempre que estou conhecendo uma nova API, costumo usar o recurso de *autocomplete* (atalho `Control + space`) e navegar pelos seus métodos.

Veja que, quando paramos em um método específico, um *box* com sua documentação é exibido:

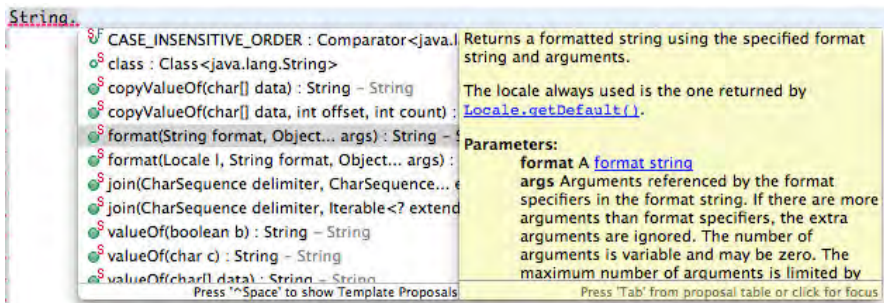


Fig. 1.3: Explorando a API do Java pelo Eclipse

1.6 TIRANDO SUAS DÚVIDAS

Ficou com alguma dúvida durante o livro? Mande um e-mail! Uma lista de discussões foi criada exclusivamente para facilitar seu contato comigo e com os demais leitores deste livro. Você pode e deve compartilhar suas dúvidas, ideias, sugestões e críticas. Todas serão muito bem-vindas.

<https://groups.google.com/group/livro-java>

Outro recurso que você pode utilizar para esclarecer suas dúvidas e participar ativamente na comunidade Java é o fórum do G.U.J. Lá você não só pode perguntar, mas também responder, editar, comentar e assistir a diversas discussões sobre o universo da programação.

<http://www.guj.com.br/>

CAPÍTULO 2

Java FX

2.1 NOSSA PRIMEIRA APP EM JAVA FX

Neste capítulo, criaremos uma tela simples usando Java FX, que servirá como base para integrarmos algumas das outras APIs que veremos no decorrer do livro. Apesar de nosso foco não ser a interface gráfica em si, escolhemos o Java FX por ser uma alternativa muito interessante e mais moderna do que o *swing*, além de ser bastante elegante. O Java FX já vem incluso no *JDK* e *JRE*.

Criando o projeto no Eclipse

Agora que já configuramos nosso Eclipse para escrever aplicações em Java FX, podemos criar um novo projeto. Para fazer isso, basta clicar em `File > New > Other...` e escolher a opção `Java FX Project`. Podemos chamar o projeto de **livraria-fx**, depois clicar em `finish` para concluir.

Se você estiver utilizando uma versão atual do Eclipse (o que é bastante recomendado), já deverá aparecer uma estrutura padrão para sua aplicação, como na seguinte imagem:

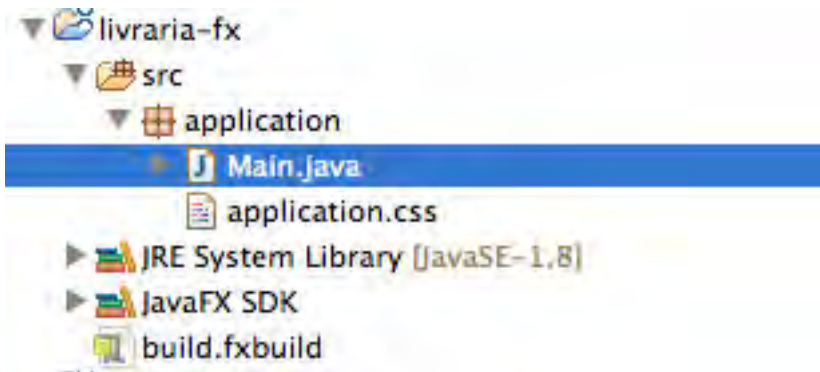


Fig. 2.1: Estrutura inicial do projeto no Eclipse

Note que dentro do `src`, temos uma pacote chamado `application` com a classe `Main.java` e o arquivo `application.css`. Veremos mais detalhes sobre esse arquivo `.css` mais à frente, por enquanto vamos focar no código Java. A classe `Main` já deve ter sido criada com o seguinte código:

```
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) {
        try {
            BorderPane root = new BorderPane();
            Scene scene = new Scene(root, 400, 400);

            scene.getStylesheets()
                .add(getClass().getResource("application.css")
                    .toExternalForm());

            primaryStage.setScene(scene);
            primaryStage.show();
        } catch (Exception e) {
```

```
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    launch(args);
}
}
```

Não se preocupe caso em seu Eclipse, ou qualquer outra IDE de sua preferência, essa classe não tenha sido criada. Você pode criá-la facilmente, mas aproveite e mantenha apenas o código que importa agora nesse começo. Vamos apagar esse `try catch` e linhas que criam a classe `Scene` e aplicam o estilo `css`. Por agora só precisamos do seguinte código:

```
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Sistema da livraria com Java FX");
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Note que já adicionei um título que não existia no arquivo Java, faça o mesmo e execute essa classe para ver o primeiro resultado. Se tudo correu bem, uma tela cinza com o título *Sistema da livraria com Java FX* deve aparecer:

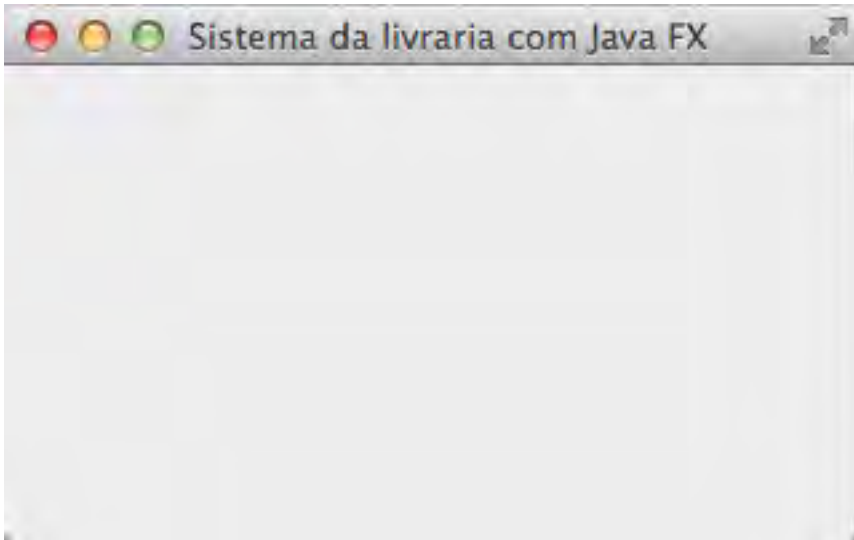


Fig. 2.2: Primeira tela em Java FX

EXECUTANDO CÓDIGO NO ECLIPSE

Caso não esteja familiarizado com o Eclipse, você pode executar seu código de diversas maneiras. Uma delas é clicando com o botão direito na classe, selecionando `Run As...` e depois `Java Application`. Outra alternativa bem prática é utilizando o atalho `Control + F11`.

Excelente, já temos algum resultado. Mas antes de seguir precisamos entender um pouco mais sobre a anatomia desse nosso primeiro código.

Podemos começar pela classe `javafx.application.Application`. A regra é simples, a classe principal de nossas aplicações em Java FX precisa herdar de `Application` e consequentemente sobrescrever seu único método abstrato, chamado `start`. Esse é o ponto de partida de toda aplicação que em Java FX.

Vale lembrar que como esse método é **abstrato**, somos obrigados a sobrescrevê-lo. Portanto, caso você não lembre disso, o compilador certamente lhe lembrará com um erro. Além disso, perceba que esse método re-

cebe um objeto do tipo `Stage` como parâmetro, que é o *container* principal da aplicação.

Por fim, note também que temos um método `main`. Nele, invocamos um outro método chamado `launch`, também presente na classe `Application`. Ele é o responsável por iniciar a aplicação.

2.2 CONFIGURANDO A LIVRARIA-BASE

Agora que já sabemos um pouco mais sobre esse código inicial, vamos começar a implementar a nossa listagem de `Produtos`. Para isso, precisaremos da interface `Produto` e de suas implementações, presentes no arquivo que você deve ter baixado no capítulo anterior. Não se preocupe se você ainda não baixou, você pode fazer isso agora pelo seguinte link:

<http://goo.gl/CfzTLB>

Após concluir o download, você precisará importar o projeto no Eclipse. Primeiro, extraia o conteúdo do ZIP que você baixou para alguma pasta de sua preferência. Agora, pelo Eclipse, clique no menu `File`, `Import...` e depois em `Existing Projects into Workspace`.

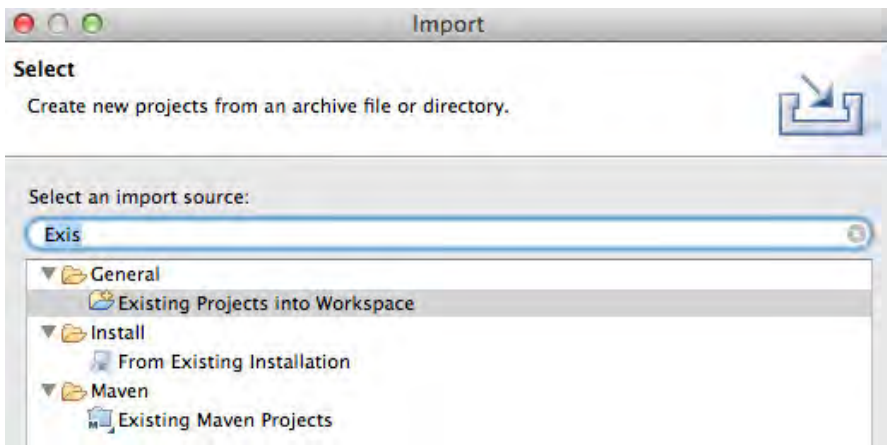


Fig. 2.3: Importando projeto pelo Eclipse.

Selecione o diretório (em `Browse...`) onde você colocou o projeto

livreria-base, que estava dentro do ZIP, e logo depois clique em Finish.

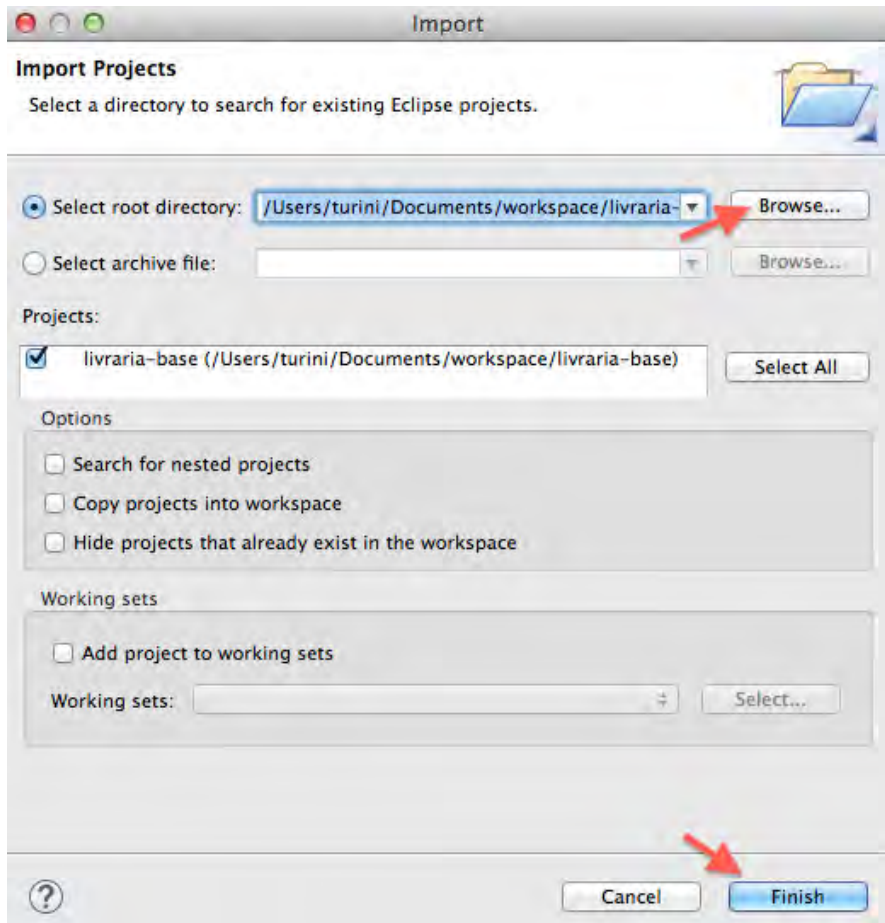


Fig. 2.4: Concluindo import do projeto base.

Queremos agora vincular os dois projetos. Por enquanto, faremos isso pelo próprio Eclipse, clicando com o botão direito no projeto `livreria-fx`, opção **Build Path** e depois **Configure Build Path...**



Fig. 2.5: Build Path > Configure Build Path...

Vá à aba *Projects*, clique em *Add*, escolha o projeto *livraria-base*, que acabamos de importar. Depois basta clicar em *Ok*.

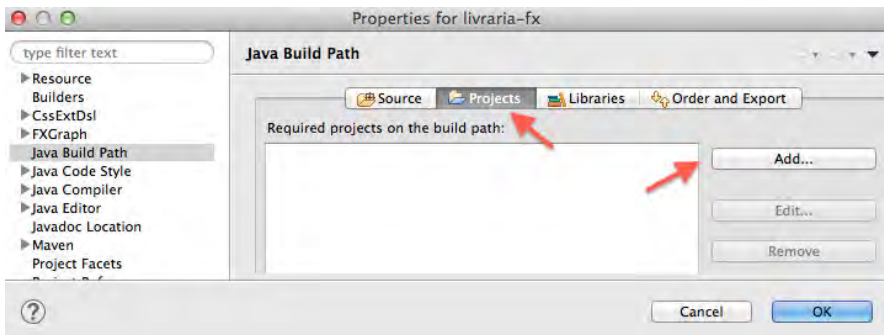


Fig. 2.6: Adicionando projeto no build path.

2.3 PREPARANDO NOSSO CENÁRIO

Antes de criar uma tabela, precisamos criar um cenário, onde ficará todo o conteúdo dessa nossa tela. Uma tela do Java FX sempre é composta por um *container* principal (*Stage*) e também por um cenário, representado pela classe *Scene*. Repare em sua estrutura:

```
Scene scene = new Scene(root, largura, altura)
```

Os parâmetros `largura` e `altura` representam bem seu significado, não é? Definimos nele o tamanho do nosso cenário, da tela da nossa aplicação. Já esse `root` nada mais é do que um auxiliar, que ajuda no processo de *layout*, como ao adicionar ou remover elementos.

O tipo do parâmetro `root` é `javafx.scene.Parent`, que é uma classe abstrata. De acordo com nossa necessidade, escolhemos uma de suas implementações. Como a tela será composta por 2 elementos (um *label* e uma tabela), podemos usar um `Group`.

OUTRAS IMPLEMENTAÇÕES

Além do `Group`, temos como opções diretamente as classes `Control`, `Region` e `WebView`. Além disso, cada uma dessas tem diversas outras *subclasses*. São várias possibilidades! Uma opção bastante usada é a `GridPane`, do pacote `javafx.scene.layout`. Ela é filha de `Pane`, que é filha de `Region` que, finalmente, é filha de `Parent`. Ufa!

Class GridPane

```
graph TD;
    A[java.lang.Object] --> B[javafx.scene.Node];
    B --> C[javafx.scene.Parent];
    C --> D[javafx.scene.layout.Region];
    D --> E[javafx.scene.layout.Pane];
    E --> F[javafx.scene.layout.GridPane];
```

Fig. 2.7: Hierarquia da classe `javafx.scene.layout.GridPane`

Você pode conhecer um pouco mais sobre essa e outras implementações em:

<https://docs.oracle.com/javafx/2/api/javafx/scene/Parent.html>

Veremos diversas delas durante o livro.

O código do nosso método `start` deve ficar parecido com:

```
public void start(Stage primaryStage) {
    Group group = new Group();
    Scene scene = new Scene(group, 690, 510);
    primaryStage.setScene(scene);
```

```
primaryStage.setTitle("Sistema da livraria com Java FX");
primaryStage.show();
}
```

Perceba que estamos chamando o método `setScene`, vinculando o cenário com o *container* principal. Quer ver o resultado? Não deixe de executar o código a cada evolução. Agora a tela estará maior e com um fundo branco, padrão do cenário.

O próximo passo será colocar um texto na tela, indicando que essa será a nossa listagem de produtos. Faremos isso com a auxílio da classe `Label`, do pacote `javafx.scene.control`.

```
Label label = new Label("Listagem de Livros");
```

Podemos agora utilizar a classe `Group`, que está vinculada ao nosso cenário, para adicionar esse novo elemento na página. Para fazer isso, basta chamar seu método `getChildren` e adicionar todos os elementos necessários, que neste caso será apenas o `label`:

```
group.getChildren().addAll(label);
```

O código completo do método `start` deverá ficar assim:

```
public void start(Stage primaryStage) {

    Group group = new Group();
    Scene scene = new Scene(group, 690, 510);

    Label label = new Label("Listagem de Livros");

    group.getChildren().addAll(label);
    primaryStage.setScene(scene);
    primaryStage.setTitle("Sistema da livraria com Java FX");
    primaryStage.show();
}
```

Ao rodar esse código, teremos a saída esperada, mas o texto está muito pequeno e grudado nas margens da tela!



Fig. 2.8: Listagem com o label desalinhado.

Para melhorar um pouco a aparência desse `label`, podemos mudar sua fonte e também adicionar um `padding`. Para isso, vamos utilizar os métodos `setFont` e `setPadding`, como a seguir:

```
Label label = new Label("Listagem de Livros");

label.setFont(Font
    .font("Lucida Grande", FontPosture.REGULAR, 30));

label.setPadding(new Insets(20, 0, 10, 10));
```

O método estático `font`, da classe `javafx.scene.text.Font`, nos possibilita passar a família, tipo e tamanho da fonte que queremos aplicar em nosso texto. Existem outras sobrecargas desse método para aplicar apenas o tamanho, apenas família e assim por diante. Repare também que no `setPadding` estamos utilizando uma classe auxiliar para passar as medidas, a `Insets`.

Execute seu código mais uma vez para perceber o resultado:

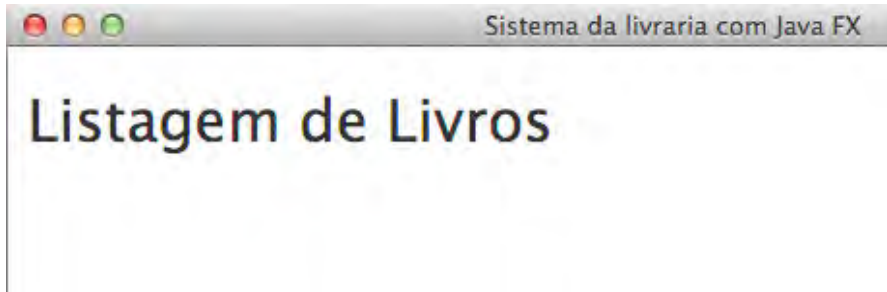


Fig. 2.9: Listagem com label alinhada.

Um pouco melhor, não acha? Agora podemos partir para a tabela!

2.4 UMA LISTAGEM DE PRODUTOS

Nosso próximo passo será criar uma tabela para exibir nossa lista de produtos. Antes de focar na tabela, vamos criar uma nova classe no pacote `repositorio`, que será responsável por retornar a lista de produtos que deverá ser exibida. Podemos chamá-la de `RepositorioDeProdutos`:

```
package repositorio;

public class RepositorioDeProdutos {

    public ObservableList<Produto> lista() {

        Autor autor = new Autor();
        autor.setNome("Rodrigo Turini");
        autor.setEmail("rodrigo.turini@caelum.com.br");
        autor.setCpf("123.456.789.10");

        Livro livro = new LivroFisico(author);
        livro.setNome("Java 8 Prático");
        livro.setDescricao("Novos recursos da linguagem");
        livro.setValor(59.90);
        livro.setIsbn("978-85-66250-46-6");
```

```
Livro maisUmlivro = new LivroFisico(autor);
maisUmlivro.setNome("Desbravando a 0.0.");
maisUmlivro.setDescricao("Livro de Java e 0.0");
maisUmlivro.setValor(59.90);
maisUmlivro.setIsbn("321-54-67890-11-2");

Autor outroAutor = new Autor();
outroAutor.setNome("Paulo Silveira");
outroAutor.setEmail("paulo.silveira@caelum.com.br");
outroAutor.setCpf("123.456.789.10");

Livro outroLivro = new LivroFisico(outroAutor);
outroLivro.setNome("Lógica de Programação");
outroLivro.setDescricao("Crie seus primeiros programas");
outroLivro.setValor(59.90);
outroLivro.setIsbn("978-85-66250-22-0");

return FXCollections
    .observableArrayList(livro, maisUmlivro, outroLivro);
}

}
```

Por enquanto, estamos criando os livros e autores manualmente, mas muito em breve faremos isso de uma forma bem mais interessante. Você já deve ter notado que o tipo de retorno dessa lista é um tanto diferente daquilo com que estamos acostumados, ela retorna uma `ObservableList<Produto>`. Esse é o tipo de lista que a tabela do Java FX espera receber. Criá-la é bem simples com o apoio da factory `FXCollections`, do pacote `javafx.collections`:

```
ObservableList lista =
    FXCollections.observableArrayList(...);
```

Agora que já temos uma lista, podemos voltar para a classe `Main` e criar um novo elemento em nosso cenário (`Scene`), uma tabela. Para fazer isso, utilizaremos a classe `TableView`, como a seguir:

```
ObservableList<Produto> produtos =
    new RepositorioDeProdutos().lista();
```

```
TableView tableView = new TableView<>(produtos);
```

Já estamos passando a lista de produtos do `RepositorioDeProdutos` como argumento em seu construtor. Nesse momento o código de nosso método `start` está assim:

```
public void start(Stage primaryStage) {
    Group group = new Group();
    Scene scene = new Scene(group, 690, 510);

    ObservableList<Produto> produtos =
        new RepositorioDeProdutos().lista();

    TableView tableView = new TableView<>(produtos);

    Label label = new Label("Listagem de Livros");

    label.setFont(Font
        .font("Lucida Grande", FontPosture.REGULAR, 30));

    label.setPadding(new Insets(20, 0, 10, 10));

    group.getChildren().addAll(label);

    primaryStage.setScene(scene);
    primaryStage.setTitle("Sistema da livraria com Java FX");
    primaryStage.show();
}
```

Podemos agora criar as colunas de nossa tabela. Basta criar uma instância do tipo `TableColumn`, recebendo o nome da coluna como parâmetro:

```
TableColumn nomeColumn = new TableColumn("Nome");
```

Além disso, para cada coluna precisamos definir sua fonte de dados, ou seja, qual propriedade da classe `Produto` deve ser exibida nessa determinada coluna. Para isso, chamamos seu método `setCellValueFactory`, passando um novo `PropertyValueFactory` como parâmetro:

```
TableColumn nomeColumn = new TableColumn("Nome");
```

```
nomeColumn.setCellValueFactory(  
    new PropertyValueFactory("nome"));
```

Opcionalmente, você pode definir um tamanho mínimo e máximo para essa coluna, utilizando seus métodos `setMinWidth` e `setMaxWidth`. Um exemplo seria:

```
descColumn.setMinWidth(230);
```

Vamos criar uma coluna para cada um dos quatro atributos principais de nossos `Produtos`, seu nome, descrição, valor e *ISBN* (que é seu código único).

Ao final, seu código deve ficar assim:

```
public void start(Stage primaryStage) {  
  
    Group group = new Group();  
    Scene scene = new Scene(group, 690, 510);  
  
    ObservableList<Produto> produtos =  
        new RepositorioDeProdutos().lista();  
  
    TableView tableView = new TableView(produtos);  
  
    TableColumn nomeColumn = new TableColumn("Nome");  
    nomeColumn.setMinWidth(180);  
    nomeColumn.setCellValueFactory(  
        new PropertyValueFactory("nome"));  
  
    TableColumn descColumn = new TableColumn("Descrição");  
    descColumn.setMinWidth(230);  
    descColumn.setCellValueFactory(  
        new PropertyValueFactory("descricao"));  
  
    TableColumn valorColumn = new TableColumn("Valor");  
    valorColumn.setMinWidth(60);  
    valorColumn.setCellValueFactory(  

```

```
        new PropertyValueFactory("valor"));

    TableColumn isbnColumn = new TableColumn("ISBN");
    isbnColumn.setMinWidth(180);
    isbnColumn.setCellValueFactory(
        new PropertyValueFactory("isbn"));

    Label label = new Label("Listagem de Livros");

    label.setFont(Font
        .font("Lucida Grande", FontPosture.REGULAR, 30));

    label.setPadding(new Insets(20, 0, 10, 10));

    group.getChildren().addAll(label);

    primaryStage.setScene(scene);
    primaryStage.setTitle("Sistema da livraria com Java FX");
    primaryStage.show();
}
```

Por fim, vamos adicionar as colunas na tabela e depois adicioná-la ao nosso cenário:

```
tableView.getColumns().addAll(nomeColumn,
    descColumn, valorColumn, isbnColumn);

group.getChildren().addAll(label, tableView);
```

Ao executar o código, a tabela estará populada como esperamos, mas totalmente desalinhada e escondendo a nossa `Label`:


```
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.stage.Stage;
import repositorio.RepositorioDeProdutos;
import br.com.casadocodigo.livraria.produtos.Produto;

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) {

        Group group = new Group();
        Scene scene = new Scene(group, 690, 510);

        ObservableList<Produto> produtos =
            new RepositorioDeProdutos().lista();

        TableView<Produto> tableView = new TableView<>(produtos);

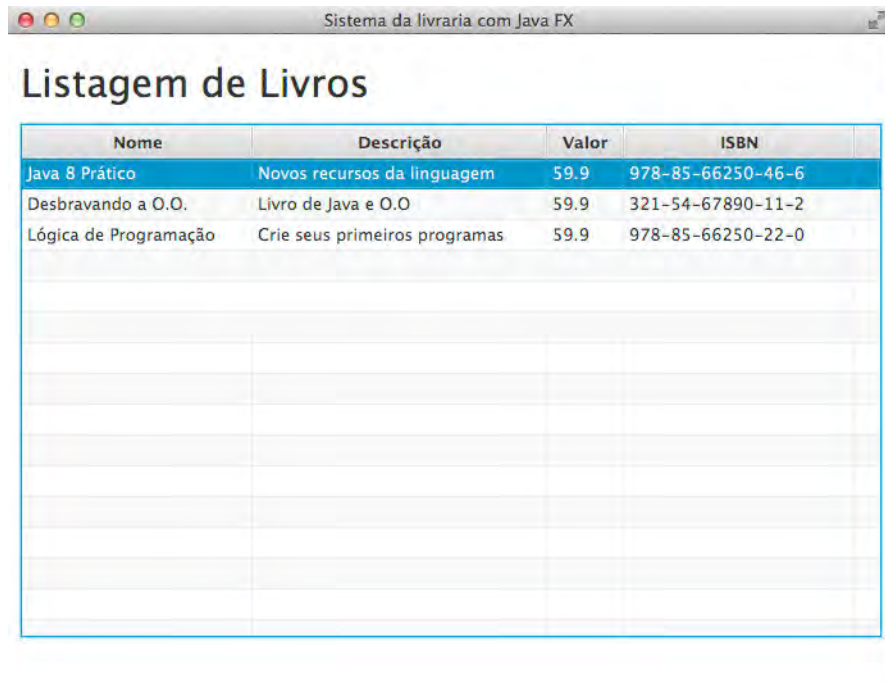
        TableColumn nomeColumn = new TableColumn("Nome");
        nomeColumn.setMinWidth(180);
        nomeColumn.setCellValueFactory(
            new PropertyValueFactory("nome"));

        TableColumn descColumn = new TableColumn("Descrição");
        descColumn.setMinWidth(230);
        descColumn.setCellValueFactory(
            new PropertyValueFactory("descricao"));

        TableColumn valorColumn = new TableColumn("Valor");
        valorColumn.setMinWidth(60);
```

```
valorColumn.setCellValueFactory(  
    new PropertyValueFactory("valor"));  
  
TableColumn isbnColumn = new TableColumn("ISBN");  
isbnColumn.setMinWidth(180);  
isbnColumn.setCellValueFactory(  
    new PropertyValueFactory("isbn"));  
  
tableView.getColumns().addAll(nomeColumn, descColumn,  
    valorColumn, isbnColumn);  
  
final VBox vbox = new VBox(tableView);  
vbox.setPadding(new Insets(70, 0, 0 ,10));  
  
Label label = new Label("Listagem de Livros");  
  
label.setFont(Font  
    .font("Lucida Grande", FontPosture.REGULAR, 30));  
  
label.setPadding(new Insets(20, 0, 10, 10));  
  
group.getChildren().addAll(label, vbox);  
  
primaryStage.setTitle("Sistema da livraria com Java FX");  
primaryStage.setScene(scene);  
primaryStage.show();  
}  
  
public static void main(String[] args) {  
    launch(args);  
}  
}
```

Ao executá-lo, teremos como resultado uma tela parecida com:



Nome	Descrição	Valor	ISBN
Java 8 Prático	Novos recursos da linguagem	59.9	978-85-66250-46-6
Desbravando a O.O.	Livro de Java e O.O	59.9	321-54-67890-11-2
Lógica de Programação	Crie seus primeiros programas	59.9	978-85-66250-22-0

Fig. 2.11: Listagem de produtos com os elementos alinhados.

USANDO @SuppressWarnings NO ECLIPSE

Para evitar os diversos alertas (*warnings*) em amarelo no Eclipse, você pode adicionar sob a sua classe a anotação `@SuppressWarnings`:

```
@SuppressWarnings({ "unchecked", "rawtypes" })
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) {

        Group group = new Group();
        Scene scene = new Scene(group, 690, 510);

        // restante do código omitido
    }
}
```

Esses *warnings* acontecem por causa dos parâmetros genéricos, que optamos não passar para simplificar e evitar tanta repetição em nosso código. Não se preocupe com isso, voltaremos a falar disso no final do livro. Por enquanto, você pode deixar os *warnings* (o código funcionará normalmente), ou adicionar a anotação `@SuppressWarnings`.

Excelente, já temos uma interface visual para nosso projeto. Ao decorrer do livro, trabalharemos com leitura e escrita em arquivos, acesso a banco de dados e mais. A cada novo capítulo uma ou mais novas funcionalidades serão adicionadas à nossa aplicação.

MAIS SOBRE JAVA FX

Se você gostou do Java FX e quiser estudar um pouco mais a fundo, talvez queira dar uma olhada em sua página no site da Oracle:

<http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html?ssSourceSiteId=otnpt>

Nela você encontra um *overview* sobre a tecnologia e links para documentação, exemplos e muito mais.

CAPÍTULO 3

Java IO

3.1 ENTRADA E SAÍDA DE DADOS

Nossa listagem ainda trabalha com valores fixos, definidos no `RepositorioDeProdutos`. Apesar disso, já queremos adicionar uma função de exportar os dados dessa listagem para um arquivo de texto. Ao clicar no botão de exportar, um arquivo deverá ser gerado com os dados da listagem.

Trabalhar com entrada e saída de dados é uma necessidade completamente natural nos mais diferentes tipos de projetos. Ler valores digitados pelo teclado, ler ou escrever em um arquivo de diferentes formatos etc., conhecer a API de controle de entrada e saída, ou **IO** como é comumente chamada, é um passo fundamental. Esse será nosso objetivo durante esse capítulo.

3.2 LENDO UM ARQUIVO DE TEXTO

Já vamos começar de forma prática, queremos ler o conteúdo de um arquivo. Para começar a testar, crie na raiz do seu projeto (pasta `livraria-fx`) um arquivo chamado `teste.txt`.

Adicione nesse arquivo o seguinte conteúdo:

```
Esse é um teste de leitura de arquivo
Queremos ler o conteúdo de todas as linhas
Mas como ler arquivos em Java?
```

Uma forma de ler o arquivo seria utilizando um `FileInputStream`, responsável por fazer a leitura de bytes de um arquivo. O construtor dessa classe nos obriga passar o nome do arquivo que será lido, que neste caso será `teste.txt`:

```
try {
    InputStream is = new FileInputStream("teste.txt");
} catch (FileNotFoundException e) {
    System.out.println("Arquivo não encontrado "+ e);
}
```

Quando estamos trabalhando com o pacote `java.io`, grande parte de seus métodos lançam `IOException`. Como essa é uma *Checked Exception* (não evitável), somos obrigados a tratá-la ou declará-la com um `throws`. Em meu exemplo, eu optei por tratar, imprimindo uma mensagem amigável no *console*.

Outro detalhe importante é que declaramos o tipo da classe `FileInputStream` como `InputStream`, que é sua *superclasse*. Essa é uma classe abstrata e tem diversas outras implementações, como `AudioInputStream` e `ByteArrayInputStream`.

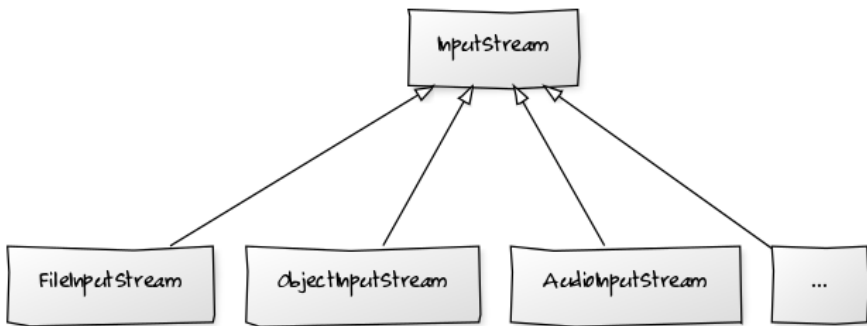


Fig. 3.1: `InputStream` e algumas de suas subclasses

A grande ideia do `InputStream` é que ela implementa o comportamento em comum entre os diferentes tipos de entrada. Além de ler um arquivo de texto, podemos precisar ler de um áudio, uma conexão remota (com *sockets*) etc. Cada tipo de leitura terá suas particularidades, mas a classe `InputStream` **define o comportamento padrão entre elas**.

Qual a vantagem dessa abstração? É bem simples, podemos tirar proveito do **polimorfismo**. Imagine um método que recebe um `InputStream` para fazer uma leitura de bytes. De onde os dados virão? De um `txt`? De um *blob* do banco de dados? Não importa. Independente de onde os dados vêm, o processo de leitura será o mesmo. Quando precisarmos ler bytes, não importando de onde, poderemos chamar um método que aceita qualquer implementação de `InputStream`.

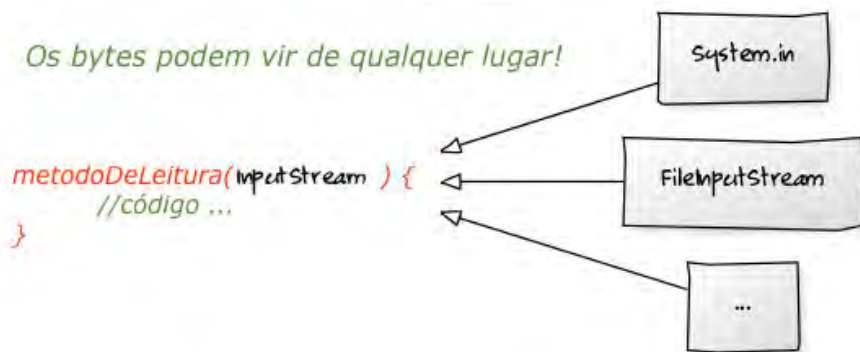


Fig. 3.2: Método recebendo um `InputStream` como parâmetro.

O PADRÃO DE PROJETO TEMPLATE METHOD

Essa abstração utilizada na classe `InputStream` ilustra um padrão de projeto, ou *Design Pattern*, bastante conhecido e interessante. Estamos falando do *Template Method*.

Há um post no blog da Caelum que se aprofunda bem nesse exemplo:

<http://blog.caelum.com.br/design-patterns-no-java-se-o-template-method/>

O mesmo acontece com `OutputStream`, que logo veremos. Assim como usaremos `InputStream` para leitura de bytes, podemos usar o `OutputStream` para escrita. O pacote de `java.io` usa e abusa da orientação a objetos, com classes abstratas, interfaces e **muito polimorfismo**.

Para concluir a leitura do arquivo, precisamos transformar os bytes desse arquivo em unicode, guardando um conjunto de `chars`. Em seguida, concatenar esse conjunto de `chars` para formar as `Strings` com o conteúdo de cada uma de suas linhas.

Parece bem trabalhoso, não é? Mas a API é bem simples, você verá que esse código é bem padrão. Logo também veremos algumas classes do pacote `java.util` que simplificam todo esse processo, mas primeiro, é importante entendê-lo.

De forma visual, o fluxo padrão de entrada é:



Fig. 3.3: Etapas do fluxo padrão de entrada

E as responsáveis por esse trabalho:

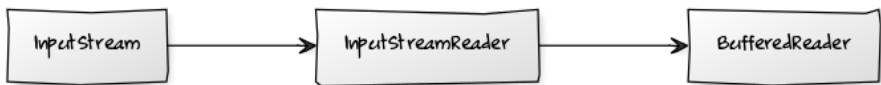


Fig. 3.4: InputStream, InputStreamReader e BufferedReader

Já conhecemos a `InputStream`. As classes `InputStreamReader` e `BufferedReader` são as responsáveis por fazer o restante do trabalho, ler `chars` e transformá-las em `Strings`:

```
try {
    InputStream is = new FileInputStream("teste.txt");
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader reader = new BufferedReader(isr);
    String linha = reader.readLine();
    while (linha != null) {
        System.out.println(linha);
        linha = reader.readLine();
    }
    reader.close();
} catch (IOException e) {
    System.out.println("Erro ao tentar ler o arquivo " + e);
}
```

A cada chamada, o método `readLine` lê uma linha e muda o cursor para a próxima, até que o arquivo acabe. Neste caso, retornará `null`.

Escreva e execute esse código e você verá que todas as linhas do arquivo serão impressas no console. Legal, não é?

3.3 LENDO TEXTO DO TECLADO COM SYSTEM.IN

Quer uma prova de que a abstração utilizada no `InputStream` é interessante? Veja como é simples mudar nosso código para que passe a ler do teclado em vez de um arquivo:

```
try {
    InputStream is = System.in;
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader reader = new BufferedReader(isr);
    String linha = reader.readLine();
    while (linha != null) {
        System.out.println(linha);
        linha = reader.readLine();
    }
    reader.close();
} catch (IOException e) {
    System.out.println("Erro ao tentar ler o arquivo " + e);
}
```

Percebeu a diferença? Ela está apenas na primeira linha, quando declaramos o `InputStream`. Tudo que precisamos fazer foi utilizar o `System.in`, que é a implementação do `InputStream` responsável por esse tipo de leitura, do teclado.

Execute o método para ver o resultado. O console ficará em branco, esperando que um texto seja digitado. Digite qualquer coisa e pressione `Enter`, o valor será impresso de volta no console. Perceba que, diferente de quando estamos lendo de um arquivo, neste caso a `linha` nunca será nula. Portanto, o código ficará em *looping*, sempre esperando uma nova entrada.

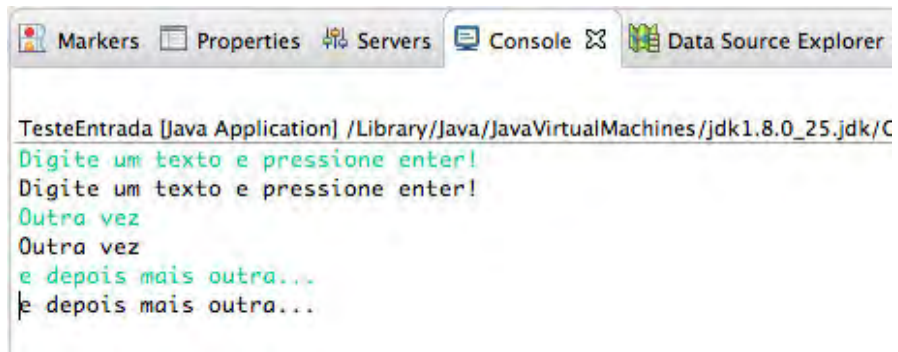


Fig. 3.5: Exemplo de saída do System.in

ENCERRANDO EXECUÇÃO MANUALMENTE OU COM EOF

Para parar a aplicação que está em *looping*, aguardando a próxima entrada do teclado, você pode pelo Eclipse clicar no ícone de *stop*.

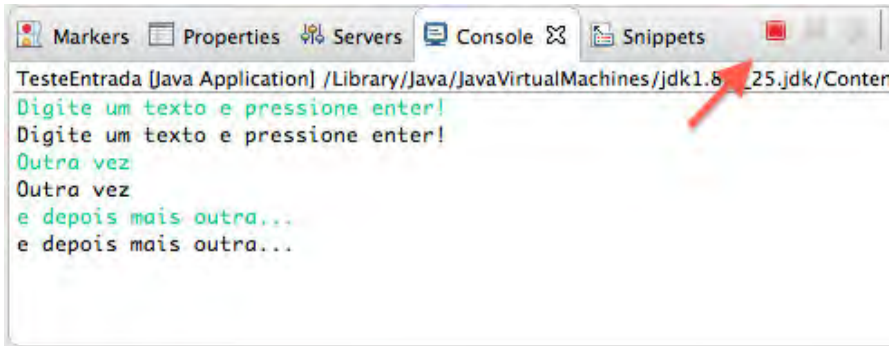


Fig. 3.6: Como interromper a execução pelo Eclipse

Outra alternativa seria mandar um sinal de fim de *stream*, o conhecido **EOF** (*end of file*). Em sistemas operacionais de base *Unix*, como Linux e Mac OS, você faz isso com o comando `Control + D`. Pelo Windows o comando será `Control + Z`.

3.4 TORNANDO TUDO MAIS SIMPLES COM SCANNER

É muito interessante conhecer essa anatomia do `InputStream` para entrada de dados, mas a realidade é que dessa forma o processo é muito verboso, com todas essas etapas. Há uma maneira mais atual e interessante.

Podemos fazer a mesma leitura do arquivo `teste.txt` utilizando a classe `Scanner`, presente no pacote `java.util`. Repare como o código é mais declarativo:

```
Scanner sc = new Scanner(new File("teste.txt"));  
while (sc.hasNextLine()) {
```

```
    System.out.println(sc.nextLine());  
}
```

Execute o código e você perceberá que a saída será a mesma. Muito mais simples, não acha? O `java.util.Scanner` surgiu no Java 5 e tem uma série de métodos úteis para facilitar seu trabalho com leitura de um `InputStream`. Métodos como `nextDouble`, que já lê os bytes e faz a conversão, podem ser bastante convenientes no trabalho do dia a dia.

CONHECENDO A API

Durante a leitura, não deixe de se aprofundar ainda mais pelo assunto consultando a documentação das classes mencionadas. O `java.util.Scanner`, por exemplo, está bem documentado em:

<http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html>

Veja que a própria documentação da classe, logo no início, já mostra alguns trechos de código como exemplo.

Note como fica nosso código de entrada do teclado com essa classe:

```
Scanner sc = new Scanner(System.in);  
System.out.println("Digite seu nome");  
String nomeDigitado = sc.nextLine();  
System.out.println("Digite sua idade");  
int idadeDigitada = sc.nextInt();  
System.out.println("Nome: " + nomeDigitado);  
System.out.println("Idade: " + idadeDigitada);
```

Bastou mudar o `InputStream` passado no construtor do `Scanner` para o `System.in`, que já vimos. Note também que estamos utilizando o método `nextInt` para recuperar o valor e já converter para um inteiro, caso contrário seria necessário fazer essa conversão manualmente. Por exemplo:

```
String idadeDigitada = sc.nextLine();  
int idade = Integer.parseInt(idadeDigitada);
```

Exercite esse código para ir se familiarizando com a API, escreva e execute em seu projeto. Um exemplo de saída seria:

```
Digite seu nome
  > Rodrigo Turini
Digite sua idade
  > 25
Nome: Rodrigo Turini
Idade: 25
```

3.5 SAÍDA DE DADOS E O OUTPUTSTREAM

O processo de escrita (saída de dados) é muito parecido, mas no lugar de `InputStream`, `InputStreamReader` e `BufferedReader`, temos respectivamente o `OutputStream`, `OutputStreamWriter` e `BufferedWriter`.



Fig. 3.7: OutputStream, OutputStreamWriter e BufferedWriter

Um exemplo de uso seria:

```
OutputStream os = new FileOutputStream("saida.txt");
OutputStreamWriter osw = new OutputStreamWriter(os);
BufferedWriter bw = new BufferedWriter(osw);
```

Note que, agora, no lugar de ler de um arquivo existente, estamos criando um novo, neste caso chamado `saida.txt`. Também somos obrigados a passar essa informação no construtor da classe `FileOutputStream`.

Por sinal, note novamente o polimorfismo em ação. Usamos um `FileOutputStream`, mas dissemos que seu tipo é `OutputStream`. Há uma implementação para cada tipo de escrita:

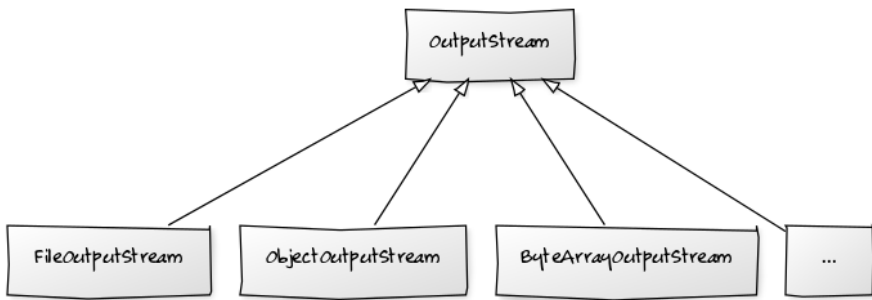


Fig. 3.8: OutputStream e algumas de suas subclasses

Podemos utilizar o método `write` para escrita do arquivo, mas ele não faz uma quebra de linha. Uma alternativa para isso seria chamar o método `newline`, que carrega essa responsabilidade.

```
bw.write("Testando a escrita em arquivo");  
bw.newLine();  
bw.write("Conteúdo na próxima linha");
```

Ou simplesmente concatenar um `/n` na `String`:

```
bw.write("Testando a escrita em arquivo\n");  
bw.write("Conteúdo na próxima linha");
```

FECHANDO O ARQUIVO COM CLOSE

É fundamental sempre fecharmos o arquivo, chamando seu método `close`. Ao chamar o método `close` de um `BufferedWriter`, ele já fará o trabalho de fechar o `OutputStreamWriter` e `FileOutputStream` para você. Em cascata.

Caso você se esqueça de fechar o arquivo nesse exemplo, que é um programa minúsculo, ele provavelmente não vai fazer a escrita no arquivo (os *bytes* ficam no *buffer* do escritor). Se for uma aplicação maior, o próprio Java fechará o `BufferedWriter` para você em algum momento (por causa do *garbage collector*, que veremos mais à frente). Mas nunca devemos deixar de fechar um arquivo esperando que isso aconteça, sempre devemos chamar o `close` quando terminarmos de fazer uma escrita ou leitura (um **io**).

No próximo capítulo, veremos uma forma bem interessante de fechar recursos, como um arquivo ou mesmo uma conexão com banco de dados. Vamos aprender que isso pode e deve ser feito em um bloco `finally`, ou ainda com um recurso conhecido como `try-with-resources`.

Pronto, agora que já sabemos isso podemos criar uma nova classe de teste para ver esse código em ação. Crie a classe `TesteSaida` para fazer alguns testes, ela pode se parecer com:

```
public class TesteSaida {  
  
    public static void main(String[] args)  
        throws IOException {  
  
        OutputStream os = new FileOutputStream("saida.txt");  
        OutputStreamWriter osw = new OutputStreamWriter(os);  
        BufferedWriter bw = new BufferedWriter(osw);  
  
        bw.write("Testando a escrita em arquivo");  
        bw.newLine();  
    }  
}
```

```
        bw.write("Conteúdo na próxima linha");  
        bw.close();  
    }  
}
```

Execute o código para ver o resultado. Um ponto importante é que a IDE normalmente não atualiza o seu projeto automaticamente se você criar um arquivo por fora dela, portanto, não se esqueça de dar um `F5` (*refresh*) no projeto após executar a classe para que o arquivo `saida.txt` fique visível.

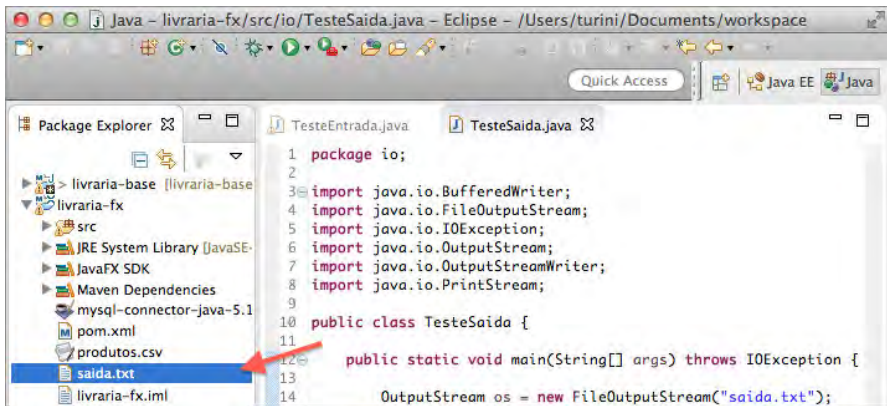


Fig. 3.9: Arquivo saida.txt no Eclipse

Só por curiosidade, se quiser você pode testar mudar a implementação de `OutputStream`, assim como fizemos no exemplo de escrita. Experimente usar o `System.out`:

```
public class TesteSaida {  
  
    public static void main(String[] args)  
        throws IOException {  
  
        OutputStream os = System.out;  
        OutputStreamWriter osw = new OutputStreamWriter(os);  
        BufferedWriter bw = new BufferedWriter(osw);
```

```
        bw.write("Testando a escrita em arquivo");
        bw.newLine();
        bw.write("Conteúdo na próxima linha");
        bw.close();
    }
}
```

Simples, não é? A mudança foi mínima e já temos a capacidade de escrever em outro local, além de um arquivo. A saída será:

```
Testando a escrita em arquivo
Conteúdo na próxima linha
```

Mas, claro, nesse caso específico um `System.out.println` seria muito mais simples. Por falar em simplicidade, veremos agora como simplificar esse processo de escrita.

3.6 ESCRITA MAIS SIMPLES COM PRINTSTREAM

Assim como o `java.util.Scanner` facilita o processo de leitura de bytes, desde o Java 5 temos uma forma mais interessante de fazer esse processo de escrita, com a classe `java.io.PrintStream`. Repare como fica nosso código utilizando-a:

```
PrintStream out = new PrintStream("saida.txt");
out.println("Testando a escrita em arquivo");
out.println("Conteúdo na próxima linha");
out.close();
```

Muito mais simples, não acha? Achou essa API familiar? Na verdade, você provavelmente já vem usando indiretamente o `PrintStream` desde o seu primeiro *hello world* em Java, pois este é o tipo do atributo `out` da classe `System`. Repare:

```
PrintStream out = System.out;
out.println("agora a saída é no console");
out.println("Conteúdo na próxima linha");
```

Ambas as classes `Scanner` e `PrintStream` facilitam bastante o processo de entrada e saída de dados. Quer ler mais sobre essas duas classes? Não deixe de conferir sua documentação:

- <http://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html>
- <http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html>

3.7 GERANDO UM CSV DE PRODUTOS

Agora que já conhecemos um pouco da API de `io`, vamos adicionar um novo comportamento em nossa listagem de produtos. Queremos exportar os produtos da listagem para um arquivo `CSV`. O termo `CSV` vem de *comma separated value*, ou seja, valores separados por vírgula. Esse formato é bastante comum e utilizado no dia a dia.

Queremos ao final ter um arquivo parecido com:

```
Nome, Descricao, Valor, ISBN
Java 8 Prático, Novos recursos da linguagem, 59.9,
978-85-66250-46-6
```

Vamos começar criando a classe `Exportador`, que será responsável por esse trabalho. Ela terá um único método, chamado `paraCSV`, que receberá um `List` de `Produto` como parâmetro:

```
public class Exportador {

    public void paraCSV(List<Produto> produtos) {

    }

}
```

Implementando esse método, nosso primeiro passo será criar um arquivo `produtos.csv` e já colocar um cabeçalho com as informações que vamos escrever. Podemos usar um `PrintStream` para fazer o trabalho, repare:

```
public void paraCSV(List<Produto> produtos)
    throws IOException {
```

```
PrintStream ps = new PrintStream("produtos.csv");
ps.println("Nome, Descricao, Valor, ISBN");

ps.close();
}
```

Já execute o código para ver o resultado. O arquivo deve ser criado na raiz do seu projeto e por enquanto apenas com o conteúdo:

```
Nome, Descricao, Valor, ISBN
```

Excelente. Agora tudo o que precisamos fazer é iterar nessa lista de produtos adicionando uma linha para cada elemento, como por exemplo:

```
public void paraCSV(List<Produto> produtos)
    throws IOException {

    PrintStream ps = new PrintStream("produtos.csv");
    ps.println("Nome, Descricao, Valor, ISBN");

    for (Produto produto : produtos) {
        ps.println(produto.getNome() + ", "
            + produto.getDescricao() + ", "
            + produto.getValor() + ", "
            + produto.getIsbn());
    }
    ps.close();
}
```

Isso já trará o resultado que esperamos. Mas o código esté bem feio, não acha? Essa concatenação de `Strings` pode prejudicar bastante a legibilidade e dificultar a manutenção de nosso código.

USANDO O PROJETO DO OUTRO LIVRO?

Se, no lugar de baixar o projeto base, você estiver utilizando o projeto do livro *Desbravando Java e Orientação a Objetos*, você precisará modificar sua interface `Produto` para que esse código compile.

Ela deve ficar assim:

```
public interface Produto
    extends Comparable<Produto> {

    double getValor();
    String getNome();
    String getDescricao();
    String getIsbn();
}
```

Uma opção talvez mais interessante seria usar o `String.format`, como a seguir:

```
public void paraCSV(List<Produto> produtos)
    throws IOException {

    PrintStream ps = new PrintStream("produtos.csv");
    ps.println("Nome, Descricao, Valor, ISBN");

    for (Produto produto : produtos) {

        ps.println(String.format("%s, %s, %s, %s",
            produto.getNome(),
            produto.getDescricao(),
            produto.getValor(),
            produto.getIsbn()));
    }
    ps.close();
}
```

Ufa! Um pouco melhor. Claro, poderíamos ter usado um

`StringBuilder`, `StringJoiner` etc., mas essa opção é bem simples e já resolve o problema. Vamos ver se está funcionando? Crie um método `main` dentro dessa mesma classe para testar, ele pode ficar assim:

```
public static void main(String[] args)
    throws IOException {

    Livro livro = new LivroFisico(new Autor());
    livro.setNome("Java 8 Prático");
    livro.setDescricao("Novos recursos da linguagem");
    livro.setValor(59.90);
    livro.setIsbn("978-85-66250-46-6");

    Livro maisUmlivro = new LivroFisico(new Autor());
    maisUmlivro.setNome("Desbravando a 0.0.");
    maisUmlivro.setDescricao("Livro de Java e 0.0");
    maisUmlivro.setValor(59.90);
    maisUmlivro.setIsbn("321-54-67890-11-2");

    new Exportador().paraCSV(Arrays.asList(livro, maisUmlivro));
}
```

O resultado será o arquivo `produtos.csv`, com o seguinte conteúdo:

```
Nome, Descricao, Valor, ISBN
Java 8 Prático, Novos recursos da linguagem, 59.9,
978-85-66250-46-6
Desbravando a 00, Livro de Java e 0.0, 59.9, 321-54-67890-11-2
```

Não encontrou o arquivo? Lembre-se de dar um `F5` (*refresh*) em seu projeto para ele aparecer na IDE. Ou, se preferir, abra a pasta do seu projeto direto pelo seu sistema operacional.

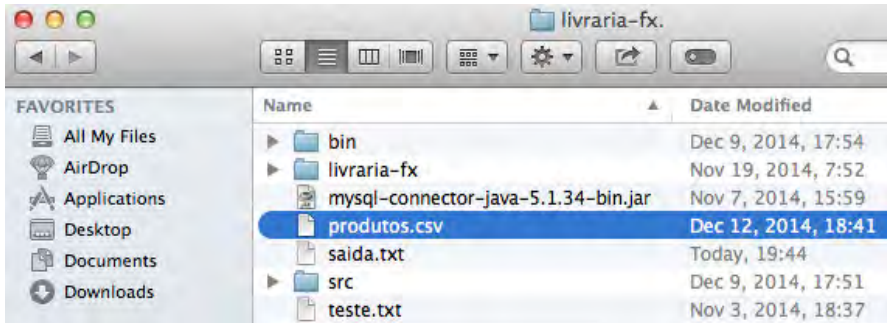


Fig. 3.10: Arquivo produtos.csv visto pelo sistema operacional

Nossa classe `Exportador` ficou assim:

```
import java.io.IOException;
import java.io.PrintStream;
import java.util.List;

import br.com.casadocodigo.livraria.produtos.Produto;

public class Exportador {

    public void paraCSV(List<Produto> produtos)
        throws IOException {

        PrintStream ps = new PrintStream("produtos.csv");
        ps.println("Nome, Descricao, Valor, ISBN");

        for (Produto produto : produtos) {

            ps.println(String.format("%s, %s, %s, %s",
                produto.getNome(),
                produto.getDescricao(),
                produto.getValor(),
                produto.getIsbn()));
        }
        ps.close();
    }
}
```

```
}
```

3.8 BOTÃO DE EXPORTAR PRODUTOS

Agora que já temos o código para exportar uma lista de produtos como um CSV, podemos adicionar um botão em nossa listagem feita em Java FX para disparar essa lógica. Isso é bem simples.

O primeiro passo é criar um `Button`, do pacote `javafx.scene.control`. Opcionalmente, em seu construtor você já pode passar o texto que quer exibir. Podemos fazer:

```
Button button = new Button("Exportar CSV");
```

Da mesma forma que fizemos com a `label` e `tableView`, precisamos adicionar esse botão no cenário da página. Basta adicioná-lo como argumento do método `addAll`:

```
group.getChildren().addAll(label, vbox, button);
```

Execute o código e você perceberá que ele já está na página, mas está desalinhado, cobrindo o texto da nossa `label`:

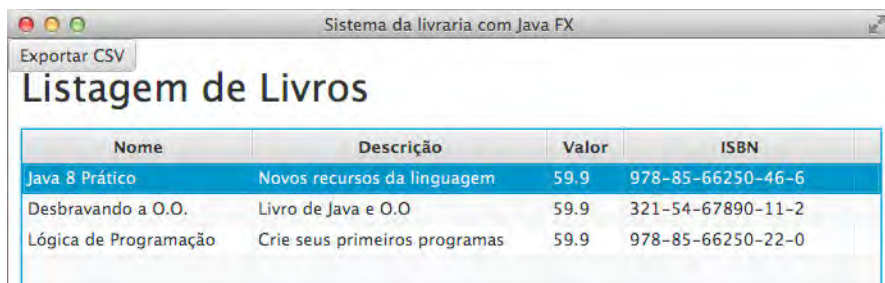


Fig. 3.11: Button desalinhado na tela

Bem, para alinhá-lo podemos envolver esse `button` em um `VBox` com um *padding* definido, assim como fizemos com a tabela. Outra opção mais simples é utilizar seus métodos `setLayoutX` e `setLayoutY` para definir sua posição, algo como:

```
Button button = new Button("Exportar CSV");
button.setLayoutX(575);
button.setLayoutY(25);
```

Agora sim, o resultado será:

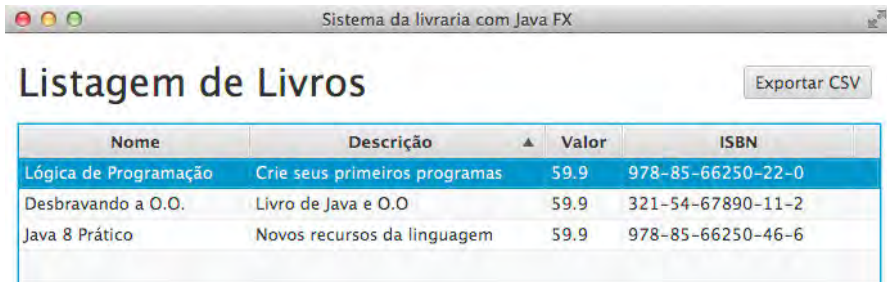


Fig. 3.12: Button alinhado com `setLayoutX` e `setLayoutY`

Mas note que ao clicar no botão nenhuma ação será executada. Faz sentido, ainda não implementamos isso.

3.9 ADICIONANDO AÇÕES COM `SETONACTION`

O método `setOnAction` é o responsável por definir as ações que serão executadas no momento do *click* de nosso `Button`. Podemos fazer:

```
button.setOnAction(???); // oq?
```

Esse método espera receber uma instância do tipo `EventHandler` (interface) como parâmetro. Sim, poderíamos criar uma classe que implementasse essa interface e passá-la como parâmetro, mas comumente fazemos isso com uma *classe anônima*:

```
button.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        // ação a ser executada
    }
});
```

Classe anônima

Se você não está acostumado ou nunca viu classes anônimas, não se preocupe. Apesar de um tanto feias, elas são bem simples de entender. No lugar de usar uma classe anônima, poderíamos ter criado uma classe que implementasse `EventHandler<ActionEvent>`:

```
public class AcaoDoBotao
    implements EventHandler<ActionEvent> {

    @Override
    public void handle(ActionEvent event) {
        // ação a ser executada
    }
}
```

Poderíamos usá-la assim:

```
button.setOnAction(new AcaoDoBotao());
```

O problema desse código é que cada botão pode ter uma ação que não vai se repetir em nenhum outro momento do código, só será usada aqui. Além disso, podemos ter vários botões; teríamos que criar uma classe nova para fazer a ação de cada um deles?

Aqui que entra a classe anônima! No lugar de criar uma classe, digamos que convencional, podemos instanciar diretamente a interface e já instanciar seus métodos ali mesmo. Essa instância criada é fruto de uma classe anônima, uma classe que nem tem um nome.

Agora, voltando ao botão. Queremos portanto adicionar a ação do *click* dentro do método `handle`, que estamos implementando na classe anônima. Vamos começar com um exemplo simples, apenas mostrando uma mensagem no *console*:

```
button.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Click!");
    }
});
```

Para testar, execute a aplicação e pressione o botão `Exportar CSV` algumas vezes, você verá que a cada *click* o texto será impresso.

O que queremos na verdade é exportar todos os nossos produtos como um CSV, portanto, fazemos:

```
button.setOnAction(new EventHandler<ActionEvent>() {  
    @Override  
    public void handle(ActionEvent event) {  
        new Exportador().paraCSV(produtos);  
    }  
});
```

Mas, espera! Esse código ainda não compila. Como o método `paraCSV` lança uma `IOException`, precisamos tratá-la ou declará-la. Vamos tratar:

```
button.setOnAction(new EventHandler<ActionEvent>() {  
    @Override  
    public void handle(ActionEvent event) {  
        try {  
            new Exportador().paraCSV(produtos);  
        } catch (IOException e) {  
            System.out.println("Erro ao exportar: " + e);  
        }  
    }  
});
```

Pronto! Coloque esse código em ação, basta clicar no botão `Exportar CSV` que o arquivo `produtos.csv` será criado na raiz de seu projeto.

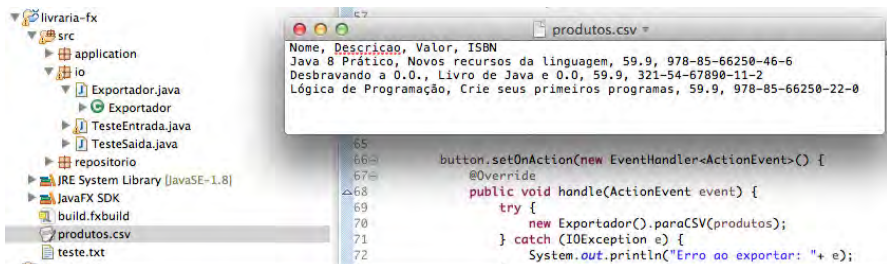


Fig. 3.13: Projeto com arquivo `produtos.csv` criado

3.10 JAVA FX E JAVA 8

Que tal dar um toque de Java 8 nesse código? Os novos recursos dessa versão deixam o código do Java FX bem mais enxuto e interessante. Quer um exemplo? Dê uma boa olhada novamente na classe anônima que criamos no `setOnAction` do botão de exportar:

```
button.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        try {
            new Exportador().paraCSV(produtos);
        } catch (IOException e) {
            System.out.println("Erro ao exportar: " + e);
        }
    }
});
```

Que tal escrevê-lo como a seguir, utilizando uma *expressão lambda*?

```
button.setOnAction(event -> {
    try {
        new Exportador().paraCSV(produtos);
    } catch (IOException e) {
        System.out.println("Erro ao exportar: " + e);
    }
});
```

Cortamos o código quase pela metade! Mas nem sempre menos linhas de código significa melhora, temos que considerar que isso pode diminuir um pouco a legibilidade. Para deixar esse código um pouco mais limpo, vamos extrair um método. Podemos chamá-lo de `exportaEmCSV`:

```
private void exportaEmCSV(ObservableList<Produto> produtos) {
    try {
        new Exportador().paraCSV(produtos);
    } catch (IOException e) {
        System.out.println("Erro ao exportar: " + e);
    }
}
```

A expressão lambda ficará assim:

```
button.setOnAction(event -> exportaEmCSV(produtos));
```

Temos agora uma linha de código! E ela é bastante declarativa> temos um `evento` como parâmetro e queremos chamar o método `exportaEmCSV`, passando a lista de produtos. Muito mais enxuto e legível, não acha?

Claro, o Java 8 vai muito além disso. Há diversas outras novidades. Está curioso para conhecer algumas delas? Então com certeza você vai gostar do livro que escrevi junto com o Paulo Silveira, o *Java 8 Prático Lambdas, Streams e os novos recursos da linguagem*.

<http://www.casadocodigo.com.br/products/livro-java8>

Apesar de estarmos usando algumas coisas do Java 8 nos exemplos desse livro, esse não é nosso foco. Você que não tem o Java 8 instalado pode manter a sintaxe das classes anônimas. Eu sempre procuro mostrar das duas formas para que você consiga acompanhar independente da sua versão do Java. Ok?

Nesse ponto, o código final da classe `Main` está assim:

```
@SuppressWarnings({ "unchecked", "rawtypes" })
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) {

        Group group = new Group();
        Scene scene = new Scene(group, 690, 510);

        ObservableList<Produto> produtos =
            new RepositorioDeProdutos().lista();

        TableView<Produto> tableView = new TableView<>(produtos);

        TableColumn nomeColumn = new TableColumn("Nome");
        nomeColumn.setMinWidth(180);
        nomeColumn.setCellValueFactory(
            new PropertyValueFactory("nome"));

        TableColumn descColumn = new TableColumn("Descrição");
```

```
descColumn.setMinWidth(230);
descColumn.setCellValueFactory(
    new PropertyValueFactory("descricao"));

TableColumn valorColumn = new TableColumn("Valor");
valorColumn.setMinWidth(60);
valorColumn.setCellValueFactory(
    new PropertyValueFactory("valor"));

TableColumn isbnColumn = new TableColumn("ISBN");
isbnColumn.setMinWidth(180);
isbnColumn.setCellValueFactory(
    new PropertyValueFactory("isbn"));

tableView.getColumns().addAll(nomeColumn, descColumn,
    valorColumn, isbnColumn);

final VBox vbox = new VBox(tableView);
vbox.setPadding(new Insets(70, 0, 0 ,10));

Label label = new Label("Listagem de Livros");

label.setFont(Font
    .font("Lucida Grande", FontPosture.REGULAR, 30));

label.setPadding(new Insets(20, 0, 10, 10));

Button button = new Button("Exportar CSV");
button.setLayoutX(575);
button.setLayoutY(25);

button.setOnAction(event -> exportaEmCSV(produtos));

group.getChildren().addAll(label, vbox, button);

primaryStage.setTitle("Sistema da livraria com Java FX");
primaryStage.setScene(scene);
primaryStage.show();
}
```



```
private void exportaEmCSV(ObservableList<Produto> produtos) {
    try {
        new Exportador().paraCSV(produtos);
    } catch (IOException e) {
        System.out.println("Erro ao exportar: " + e);
    }
}

public static void main(String[] args) {
    launch(args);
}
}
```

A única diferença para você, que está usando Java 7, será no `setOnAction` do `button`. Agora que extraímos o método, ele ficará assim com classe anônima:

```
button.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        exportaEmCSV(produtos);
    }
});
```


CAPÍTULO 4

Banco de Dados e JDBC

A aplicação está ficando mais completa, mas ainda estamos trabalhando com valores fixos na listagem de produtos que criamos manualmente na classe `RepositorioDeProdutos`. Nosso objetivo agora será buscar essas informações de um banco de dados, como é natural nas aplicações do dia a dia.

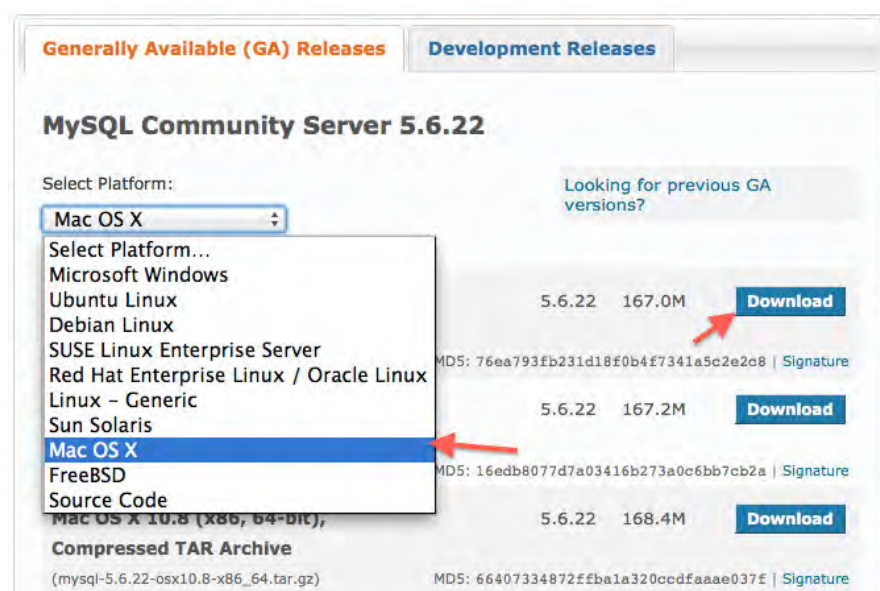
4.1 INICIANDO COM MySQL

Usaremos o banco de dados MySQL no projeto de nossa livraria, já que é um dos bancos de dados relacionais mais utilizados no mercado, é gratuito e bastante simples de instalar.

Se você ainda não tem o MySQL instalado, pode fazer download pelo link:
<http://dev.mysql.com/downloads/mysql/>



Ao final da página, você encontrará um campo para escolher seu sistema operacional. Selecione e depois clique em download, na opção que preferir:



No mesmo site, você encontra um tutorial de instalação de acordo com o

seu sistema operacional. Ou você pode acessar o tutorial diretamente em:

<http://dev.mysql.com/doc/refman/5.7/en/installing.html>

Após baixar e instalar o MySQL, vamos nos logar no MySQL e criar a base de dados (*database*) de nosso projeto. O primeiro passo é bem simples, tudo que você precisa fazer para se logar é abrir seu terminal e digitar:

```
mysql -u SEU_USUARIO -p SUA_SENHA
```

Como em meu caso o usuário é `root` e eu não tenho uma senha, só preciso fazer:

```
mysql -u root
```

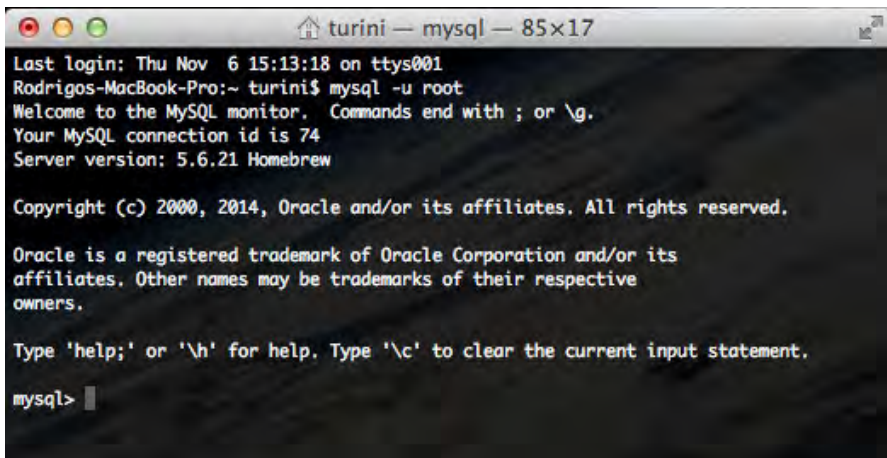


Fig. 4.3: Tela inicial do MySQL Command Line

MYSQL COMMAND LINE OU WORKBENCH?

Há quem prefira utilizar o *MySQL Workbench*, que é uma ferramenta visual. Pela simplicidade, nos exemplos do livro vamos utilizar o *MySQL Command Line*. Nele, nós executamos as instruções diretamente pela linha de comando, no terminal.

Se tiver qualquer dificuldade para instalar ou executar os comandos, não pense duas vezes antes de mandar um e-mail para lista desse livro.

Agora que já estamos logados, podemos criar uma nova *database* chamada `livraria`. Isso pode ser feito com o comando:

```
create database livraria;
```

Após executá-lo, a saída deve ser parecida com:

```
Query OK, 1 row affected (0.00 sec)
```

Perfeito, vamos usar agora o comando `use livraria` para acessar essa base de dados que criamos.

```
use livraria;
```

A mensagem `Database changed` deve ser exibida.

4.2 CRIANDO A TABELA DE PRODUTOS

Podemos agora criar uma tabela onde serão **persistidas** as informações de nossos produtos. Faremos isso com o comando `create table` a seguir:

```
CREATE TABLE produtos (  
    id BIGINT NOT NULL AUTO_INCREMENT,  
    nome VARCHAR(255),  
    descricao VARCHAR(255),  
    valor VARCHAR(255),  
    isbn VARCHAR(255),  
    PRIMARY KEY (id)  
);
```

Se você não está acostumado com SQL, não se preocupe. Nosso foco aqui será a integração do código Java com um banco de dados, e não o SQL em si. Em poucas palavras, o comando criou uma tabela de produtos com as colunas `id`, `nome`, `descricao` e `isbn`, que é o que precisamos por agora. Note também que dissemos que o `id` do produto não pode ser nulo e deve ser autoincrementado, assim o próprio banco de dados cuidará de incrementar o valor do `id` a cada novo registro.

4.3 O PACOTE JAVA.SQL E O JDBC

A biblioteca padrão do Java para persistência em banco de dados é conhecida como *JDBC*, de *Java Data Base Connective*. O *JDBC* é, na verdade, um conjunto de interfaces bem elaborado que deve ser implementado de forma diferente para cada banco de dados. Dessa forma, evitamos que cada banco tenha seu próprio conjunto de classes e métodos. Qual a vantagem disso? Manutenibilidade é uma das muitas. Migrar de um banco para outro é um processo fácil, já que todos implementam as mesmas interfaces, portanto possuem métodos com a mesma assinatura.

Esse conjunto de classes é conhecido como `driver`; elas fazem a ponte entre a API de *JDBC* e o banco de dados. O nome é análogo aos drivers utilizados na instalação de impressoras. Da mesma forma que os drivers possibilitam a comunicação entre uma determinada impressora com os diferentes sistemas operacionais existentes, os drivers que implementam as interfaces do *JDBC* possibilitam a comunicação entre nosso código Java com os diferentes bancos de dados existentes.

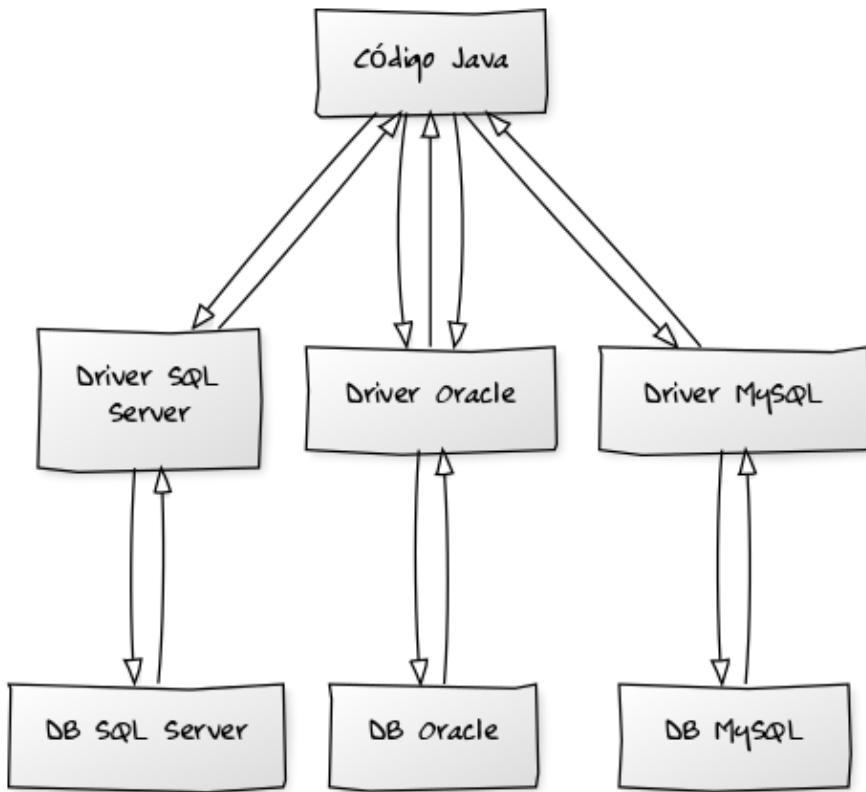


Fig. 4.4: Comunicação entre código Java e Banco de Dados

BOA PRÁTICA: PACOTE `JAVA.SQL`

No momento de fazer os *imports* das classes, tome o cuidado de sempre escolher o pacote `java.sql`. Nunca escolha opções de classes equivalentes do pacote `com.mysql`, ou qualquer outro banco de dados que você estiver utilizando. Dessa forma, você não se acopla com a API do banco de dados que está utilizando.

4.4 ABRINDO CONEXÃO COM MySQL EM JAVA

A classe `DriverManager` é a responsável por gerenciar o processo de comunicação com os drivers dos bancos de dados. Para abrir uma conexão, tudo que precisamos fazer é invocar seu método `getConnection` passando como parâmetro uma *String de conexão do JDBC*, seu login e senha. Essa *String de conexão*, para o MySQL, tem a seguinte estrutura:

```
jdbc:mysql://IP/DATABASE
```

O `IP` deve ser substituído pelo IP do servidor onde se encontra o banco de dados. Em nosso caso, como será na mesma máquina, podemos deixar como `localhost`. A `DATABASE` deve ser substituída pelo nome da base de dados que criamos para o projeto, a `livraria`. Portanto:

```
jdbc:mysql://localhost/livraria
```

Para utilizar o `DriverManager`, precisaremos adicionar o driver do MySQL em nosso projeto. Esse arquivo também está disponível no zip que você baixou com os arquivos necessários para este livro. Basta copiar o arquivo `mysql-connector-java-5.1.34-bin.jar` para dentro do seu projeto no Eclipse. Com o botão direito do mouse, clique no driver e depois em `Build Path` e `Add to Build Path`.

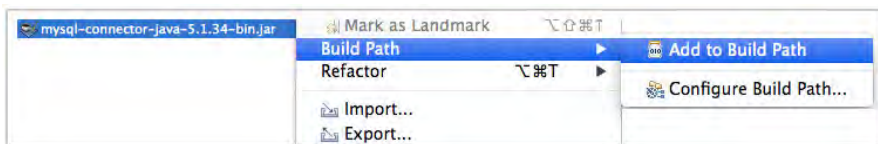


Fig. 4.5: Adicionando driver do MySQL no Build Path.

Se você ainda não está familiarizado com o termo `JAR`, não se preocupe. Temos um capítulo que detalha bem esse tipo de arquivo e seu uso.

Agora que temos o driver, vamos criar uma nova classe, que chamaremos de `ConnectionFactory` para colocar isso tudo em prática. Seu código deverá se parecer com:

```
public class ConnectionFactory {  
  
    public Connection getConnection() {  
        String url = "jdbc:mysql://localhost/livraria";  
        return DriverManager.getConnection(url, "root", "");  
    }  
}
```

DESIGN PATTERNS E O FACTORY METHOD

Não foi por acaso que chamamos a classe `ConnectionFactory` dessa forma. Essa é uma prática de projeto muito comum e bastante recomendada. Muitas vezes queremos controlar o processo de criação de um objeto que pode ser repetido diversas vezes em nosso código, como o de abrir uma conexão. O que aconteceria se esse código estivesse espalhado por nossa aplicação e alguma informação mudasse? Uma troca de *login*, por exemplo, faria você mudar diversas partes do seu código.

Note que, agora que temos a classe `ConnectionFactory`, não estamos deixando as informações do banco de dados se replicarem em toda nossa aplicação. O método `getConnection` não recebe nenhum parâmetro e retorna uma conexão pronta para ser utilizada. Alguma informação mudou? Não tem problema, só preciso mudar um único ponto do meu código (a implementação do método `getConnection`), que isso será replicado pra todos que o usam. Interessante, não acha?

Essa boa prática de projeto é conhecida como *factory method* e compõe um dos diversos *Design Patterns* existentes.

Mas esse código ainda não compila, o método `getConnection` lança uma `SQLException`. Precisamos tratar ou declarar essa exceção de alguma forma. uma maneira bastante recomendada de tratar seria:

```
public class ConnectionFactory {  
  
    public Connection getConnection() {  
        String url = "jdbc:mysql://localhost/livraria";
```

```
        try {
            return DriverManager.getConnection(url, "root", "");
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Qual a vantagem em fazer isso? Afinal só estamos “embrulhando” e re-lançando a `SQLException` como uma `RuntimeException`. A grande ideia dessa prática é que assim não deixamos essa *exception* da API de JDBC se espalhar pelo nosso código, diminuindo **acoplamento** com essa API.

Já podemos testar nosso código para garantir que a conexão com o banco de dados está acontecendo sem nenhum problema. Podemos criar um método `main`, na própria classe `ConnectionFactory`, com o seguinte código:

```
public static void main(String[] args) {
    Connection conn = new ConnectionFactory().getConnection();
    System.out.println("Conexao aberta, e agora?");
    conn.close();
}
```

Execute para ver se tudo está ok. Alguns dos erros comuns são:

- Errar a digitação da `String` de conexões, como por exemplo esquecer de uma barra ou algo desse tipo. Neste caso, uma *exception* parecida com a seguinte deve acontecer:

```
Caused by: java.sql.SQLException: No suitable
    driver found for jdbc:mysql://localhost/livraria
```

- Referenciar uma database que não existe. Por exemplo, se eu não criar a database `livraria` no MySQL, o seguinte erro vai acontecer:

```
Caused by: com.mysql.jdbc.exceptions.jdbc4
    .MySQLSyntaxErrorException: Unknown database 'livraria'
```

Se tudo correu bem, a mensagem *Conexao aberta, e agora?* deve ser exibida. Podemos dar mais um passo, adicionar e listar elementos com a API do JDBC.

4.5 LISTANDO TODOS OS PRODUTOS DO BANCO

Vamos começar pelo nosso objetivo, que é listar todos os produtos do banco de dados. Para fazer isso, limpe o conteúdo do método `lista` da classe `RepositorioDeProdutos`, deixe apenas:

```
public class RepositorioDeProdutos {  
  
    public ObservableList<Produto> lista() {  
  
        return FXCollections.observableArrayList();  
    }  
}
```

O primeiro passo será escrever a SQL que queremos executar no banco, que será algo como:

```
String sql = "select * from produtos";
```

A cláusula `select` com `*` está indicando que queremos todas as colunas de todos os produtos. Para executar esta, ou qualquer outra *query*, precisaremos antes abrir uma conexão com o banco e pra isso podemos utilizar a `ConnectionFactory` que criamos. O código fica assim:

```
Connection conn = new ConnectionFactory().getConnection();  
String sql = "select * from produtos";
```

Como enviar e executar essa *query* em um banco de dados? É bem simples, há uma interface chamada `PreparedStatement` para nos ajudar neste trabalho. Basta chamar o método `prepareStatement` da `Connection` que criamos, passando a SQL como argumento:

```
Connection conn = new ConnectionFactory().getConnection();  
String sql = "select * from produtos";  
PreparedStatement ps = conn.prepareStatement(sql);  
ps.executeQuery();
```

Esse código lança uma `SQLException`, vamos relançar essa exceção como uma `Runtime`, da mesma forma que fizemos na `ConnectionFactory` para diminuir o acoplamento com a API de JDBC. O código ficará assim:

```
Connection conn = new ConnectionFactory().getConnection();
PreparedStatement ps;
try {
    ps = conn.prepareStatement("select * from produtos");
    ps.executeQuery();
} catch (SQLException e) {
    throw new RuntimeException(e);
}
```

O método `executeQuery` fará seu trabalho e retornará um objeto com todos os registros do resultado da consulta. Esse objeto tem o tipo `ResultSet`. Cada chamada ao seu método `next` retornará `true` enquanto houver próximos registros, portanto, é natural iterar nessa estrutura utilizando um `while`, como faremos a seguir:

```
while(resultSet.next()) {
    // como pegar o nome, valor, etc?
}
```

Seus métodos `get` são os responsáveis por recuperar o valor de uma determinada coluna da tabela. Se for uma coluna de texto, utilizamos o `getString`, `getInt` para um inteiro e assim por diante. Em nosso caso, precisaremos apenas dos métodos `getString` para *nome*, *descricao*, *isbn* e do `getDouble` para o *valor* do produto. O código deve ficar assim:

```
while(resultSet.next()) {
    String nome = resultSet.getString("nome");
    String descricao = resultSet.getString("descricao");
    double valor = resultSet.getDouble("valor");
    String isbn = resultSet.getString("isbn");
}
```

Para concluir, nosso método precisa retornar a `ObservableList` de `Produtos` que será consumida pela nossa listagem em Java FX. Por enquanto, vamos representar os produtos apenas como `LivroFisico`:

```
while(resultSet.next()) {  
    LivroFisico livro = new LivroFisico(new Autor());  
    livro.setNome(resultSet.getString("nome"));  
    livro.setDescricao(resultSet.getString("descricao"));  
    livro.setValor(resultSet.getDouble("valor"));  
    livro.setIsbn(resultSet.getString("isbn"));  
    produtos.add(livro);  
}
```

Ao final, nossa classe `RepositorioDeProdutos` deve ficar assim:

```
public class RepositorioDeProdutos {  
  
    public ObservableList<Produto> lista() {  
  
        ObservableList<Produto> produtos = observableArrayList();  
        Connection conn = new ConnectionFactory().getConnection();  
        PreparedStatement ps;  
        try {  
            ps = conn.prepareStatement("select * from produtos");  
            ResultSet resultSet = ps.executeQuery();  
  
            while(resultSet.next()) {  
                LivroFisico livro = new LivroFisico(new Autor());  
                livro.setNome(resultSet.getString("nome"));  
                livro.setDescricao(resultSet.getString("descricao"));  
                livro.setValor(resultSet.getDouble("valor"));  
                livro.setIsbn(resultSet.getString("isbn"));  
                produtos.add(livro);  
            }  
            resultSet.close();  
            ps.close();  
            conn.close();  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
        return produtos;  
    }  
}
```

Mas, espera! Ao executar a classe `Main` agora, nossa listagem de produtos está vazia! Faz sentido, não é? Afinal, não adicionamos nenhum produto ao banco de dados, a tabela `produtos` está vazia. Esse será nosso próximo passo.

INSERINDO MANUALMENTE NO MySQL

Se você quiser, pode inserir um livro manualmente na tabela de produtos para confirmar desde já que seu código está funcionando como deveria. Você pode fazer isso executando os seguintes comandos em seu terminal:

```
$ mysql -u root livraria
$ insert into produtos (nome, descricao, valor, isbn)
  values ('Java 8 Prático', 'Novos recursos da linguagem',
    '59.90', '978-85-66250-46-6');
```

4.6 IMPORTANDO PRODUTOS DE UM DUMP

Para não precisar cadastrar manualmente, vamos importar o arquivo `dump.sql` com alguns livros já cadastrados para nossa tabela de produtos. Um arquivo de *dumb* é um arquivo de texto com instruções SQL, com os *inserts* de dados ou até mesmo com as instruções de criação de tabelas.

CRIANDO SEU PRÓPRIO DUMP

Se quiser saber mais sobre esse arquivo de *dump*, ou mesmo criar o seu próprio, você pode dar uma olhada no link:

<http://dev.mysql.com/doc/refman/5.0/en/mysqldump-sql-format.html>

Isso pode ser bem útil quando queremos fazer *backups* de segurança de nossas bases de dados.

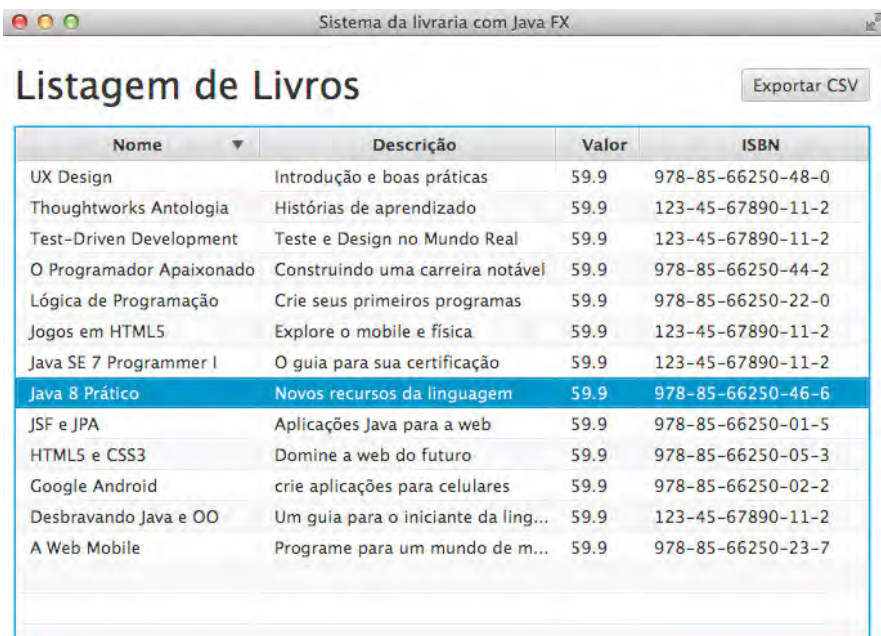
Esse arquivo também está presente naquele `zip` com o conteúdo inicial do projeto, em `arquivos-do-livro/dump.sql`. Tudo que precisamos fazer para importar esse *dump* em nosso banco de dados é executar a seguinte instrução no terminal:

```
mysql -uroot < CAMINHO_COMPLETO_PARA_O_DUMP
```

Para facilitar o processo, você pode copiar o arquivo `dump.sql` para a pasta raiz de seu usuário, e executar apenas:

```
mysql -uroot < dump.sql
```

Ótimo! Depois que concluir, tente acessar novamente a listagem de produtos. Se tudo correu bem, diversos livros devem ser listados.



Nome	Descrição	Valor	ISBN
UX Design	Introdução e boas práticas	59.9	978-85-66250-48-0
Thoughtworks Antologia	Histórias de aprendizado	59.9	123-45-67890-11-2
Test-Driven Development	Teste e Design no Mundo Real	59.9	123-45-67890-11-2
O Programador Apaixonado	Construindo uma carreira notável	59.9	978-85-66250-44-2
Lógica de Programação	Crie seus primeiros programas	59.9	978-85-66250-22-0
Jogos em HTML5	Explore o mobile e física	59.9	123-45-67890-11-2
Java SE 7 Programmer I	O guia para sua certificação	59.9	123-45-67890-11-2
Java 8 Prático	Novos recursos da linguagem	59.9	978-85-66250-46-6
JSF e JPA	Aplicações Java para a web	59.9	978-85-66250-01-5
HTML5 e CSS3	Domine a web do futuro	59.9	978-85-66250-05-3
Google Android	crie aplicações para celulares	59.9	978-85-66250-02-2
Desbravando Java e OO	Um guia para o iniciante da ling...	59.9	123-45-67890-11-2
A Web Mobile	Programa para um mundo de m...	59.9	978-85-66250-23-7

FRAMEWORKS ORM

Existem diversas ferramentas que facilitam e muito o uso do JDBC, que são conhecidas como ferramentas *ORM*, de *Object Relational Mapping*. A ideia é que você trabalhe pensando exclusivamente em orientação a objetos, enquanto essas bibliotecas (ou *frameworks*, como são comumente chamadas) o ajudam a traduzir isso para o mundo relacional dos bancos de dados. O *Hibernate* é um dos frameworks de *ORM* mais conhecidos e utilizados do mercado. Aconselho bastante que você pesquise a respeito desses e outros após conhecer a API de JDBC do Java.

Se quiser saber mais sobre o hibernate você pode querer dar uma olhada no site:

<http://hibernate.org/orm/>

4.7 PARA SABER MAIS: ADICIONANDO PROGRAMATICAMENTE

Para conhecer um pouquinho mais sobre a API do JDBC, vamos criar mais um método na classe `RepositorioDeProdutos`, que exemplificará como é feito um `INSERT` no banco de dados programaticamente. Vamos chamá-lo de `adiciona`, afinal esse será seu propósito.

```
public class RepositorioDeProdutos {  
  
    public ObservableList<Produto> lista() {  
        // código omitido  
    }  
  
    public void adiciona(Produto produto) {  
        // código de adicionar com JDBC  
    }  
}
```

Note que esse método já está recebendo o `Produto` que deverá ser adicionado no banco.

Da mesma forma como fizemos com o método `lista`, utilizaremos a interface `PreparedStatement` para executar as instruções SQL no banco de dados.

```
public void adiciona(Produto produto) {
    Connection conn = new ConnectionFactory().getConnection();
    PreparedStatement ps = conn.prepareStatement("???");

    // código de executar o PreparedStatement

    ps.close();
    conn.close();
}
```

Vale lembrar que esse código não compilará se não tratarmos a `SQLException` que o método `prepareStatement` lança.

```
public void adiciona(Produto produto) {
    Connection conn = new ConnectionFactory().getConnection();
    PreparedStatement ps;
    try {
        ps = conn.prepareStatement("???");

        // código de executar o PreparedStatement

        ps.close();
        conn.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

Por enquanto, o código está muito parecido com o de listagem, mas a grande diferença começa aqui. Uma forma de escrever a cláusula `INSERT` que será executada no banco de dados seria:

```
ps = conn.prepareStatement("insert into produtos (nome,
    descricao, valor, isbn) values ('Livro de PHP',
    'Um livro sobre PHP', '59.90', '123-45-67890-12-3')");
```

Mas, claro, não podemos deixar esses valores fixos em nosso código. Uma alternativa é concatenar os atributos do `Produto`, como faremos a seguir:

```
ps = conn.prepareStatement("insert into produtos (nome, "
    + "descricao, valor, isbn) values ('"+produto.getNome()+"', "
    + "'"+produto.getDescricao()+"', '"+produto.getValor()+"', "
    + "'"+produto.getIsbn()+"");
```

Que tal? Nem um pouco legível, não acha? Imagine se no lugar de 4 campos existissem cerca de 20, com o que esse código se pareceria?

A prova de que é muito difícil manter um código como esse é que ao final da `String` eu deixei de fora uma ``` (aspas simples). Você identificou isso ao olhar pro código? Sem dúvida poderia passar despercebido.

Mas esse não é o único problema desse código cheio de concatenações, há ainda o clássico problema de nomes que contêm aspas simples (como *Joana D'arc*). O que aconteceria ao tentar concatenar essa `String` na SQL anterior? A instrução SQL vai quebrar toda! Pior ainda, fazer concatenação dessa forma possibilita ao usuário final modificar seu SQL para executar o que ele bem entender, o clássico problema de **SQL Injection**.

PARA SABER MAIS: SQL INJECTION

Para entender melhor o que é uma injeção de SQL, vamos considerar a seguinte instrução que faz um `SELECT` verificando se o usuário existe. Ela comumente seria executada por um formulário de login.

```
ps = conn.prepareStatement("select * from usuarios where "  
    + "login = '"+login+"' and senha ='"+senha+"'");
```

O que acontece se o valor da variável `login` for um texto vazio e o valor da variável `senha` for `' or 1=1`? Vamos substituir a `String` com esses valores para verificar. O resultado será:

```
select * from usuarios where login ='' and senha ='' or 1=1
```

Esse comando trará todos os usuários de nosso banco de dados! Ou seja, o usuário final da aplicação pode enviar qualquer informação, que pode ser maliciosa, e resultar em um retorno inesperado como esse. Nunca devemos concatenar variáveis em comandos SQL dessa forma.

Bem, já vimos que essa é uma péssima prática. No lugar de concatenar dessa forma, a interface `PreparedStatement` nos oferece uma série de métodos `set` para popularmos os valores variáveis da instrução SQL. Nosso comando SQL ficará como a seguir, deixando pontos de interrogação no lugar desses valores:

```
ps = conn.prepareStatement("insert into produtos (nome,"  
    + " descricao, valor, isbn) values (?, ?, ?, ?)");
```

Para popular as variáveis, podemos utilizar os métodos `setString` para os valores de texto e `setDouble` para o campo `valor`, que é do tipo `double`, sempre passando a posição (que curiosamente começa em 1, e não 0), seguida do valor que deve ser preenchido:

```
ps = conn.prepareStatement("insert into produtos (nome,"  
    + " descricao, valor, isbn) values (?, ?, ?, ?)");
```

```
ps.setString(1, produto.getNome());
ps.setString(2, produto.getDescricao());
ps.setDouble(3, produto.getValor());
ps.setString(4, produto.getIsbn());
```

4.8 QUAL A MELHOR FORMA DE FECHAR A CONEXÃO?

Um ponto fundamental que precisa ser cuidadosamente verificado sempre que estamos trabalhando com banco de dados é a forma como fechamos nossas conexões. O nosso código parece estar fazendo isso adequadamente, não acha? Dê uma boa olhada:

```
public class RepositorioDeProdutos {

    public ObservableList<Produto> lista() {

        // código omitido
        try {
            // código omitido
            conn.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
        return produtos;
    }
}
```

Tudo parece estar certo, afinal estamos fechando a conexão. Isso é feito dentro do bloco `try`, logo após executar o código no banco de dados. Mas o que acontecerá quando esse código lançar uma `Exception`? Bem, você já deve ter percebido o problema. O código do bloco `try` será interrompido, o `catch` será executado e a conexão não será fechada.

Uma forma de resolver o problema seria adicionando a chamada ao método `close` também dentro do `catch`, como no exemplo:

```
public class RepositorioDeProdutos {
```

```
public ObservableList<Produto> lista() {  
  
    // código omitido  
    try {  
        // código omitido  
        conn.close();  
    } catch (SQLException e) {  
        conn.close();  
        throw new RuntimeException(e);  
    }  
    return produtos;  
}
```

Isso resolve o problema, mas o que você acha da solução? Nunca é muito interessante replicar código dessa forma; o que aconteceria se esse método tivesse múltiplos blocos `catch`? Copiamos e colamos em cada um deles? E se esquecermos de algum?

Por nossa sorte não precisamos fazer dessa forma, que não parece ser ideal. Podemos utilizar o `finally`:

```
public class RepositorioDeProdutos {  
  
    public ObservableList<Produto> lista() {  
  
        // código omitido  
        try {  
            // código omitido  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        } finally {  
            conn.close();  
        }  
        return produtos;  
    }  
}
```

As instruções de dentro do bloco `finally` sempre são executadas, em caso de `exception` ou não. É sempre muito interessante fazer oper-

ações como esta, de fechar uma conexão, dentro desse bloco. Colocamos no `finally` operações que devem ser executadas a qualquer custo.

Mas ainda há um problema. Se você fez essa mudança, já deve ter percebido que o código parou de compilar. O problema é que o método `close` da classe `Connection` lança uma *checked exception*, ou seja, uma exceção que precisa ser tratada ou declarada. Adicionar um novo `try catch` dentro do bloco `finally` teria um custo bem grande na legibilidade desse código. Repare como ele ficaria:

```
public class RepositorioDeProdutos {

    public ObservableList<Produto> lista() {

        // código omitido
        try {
            // código omitido
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            try {
                conn.close();
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
        }
        return produtos;
    }
}
```

O código ficou bem ruim, não acha? Por nossa sorte, o Java 7 trouxe uma novidade muito interessante, conhecida como `try-with-resources`. Você pode declarar e também inicializar a `Connection` dentro do `try`, como a seguir:

```
public class RepositorioDeProdutos {

    public ObservableList<Produto> lista() {
```

```
// código omitido
try (Connection conn = new ConnectionFactory()
    .getConnection()) {
    // código omitido
} catch (SQLException e) {
    throw new RuntimeException(e);
}
return produtos;
}
```

Pronto, a conexão será fechada para você, independente de se o código funcionou ou lançou uma `exception`. Mas em que momento chamamos o método `close`? A resposta é: em nenhum. O segredo desse recurso está na interface `AutoCloseable`, que interfaces como a `Connection` herdam. Você só pode usar o `try-with-resources` em objetos do tipo `AutoCloseable`, assim o Java sabe exatamente qual o método que deve ser chamado após a execução do seu `try catch`. Ele funciona como um `finally` implícito.

4.9 O PADRÃO DE PROJETO DAO

Uma última alteração que podemos fazer nessa classe, a `RepositorioDeProdutos`, é em seu nome. O que estamos fazendo é uma prática bem comum: isolar a lógica de acesso ao banco de dados em uma classe especialista neste trabalho. O fluxo atual está assim:



Fig. 4.7: Fluxo atual de acesso a dados.

O benefício de **encapsular** essa regra de negócios (acesso ao banco) em uma classe já é facilmente notado quando pensamos em evoluções futuras. Por exemplo, se no lugar de persistirmos as informações em

um banco de dados resolvermos utilizar uma planilha de *Excel*, quantas classes precisarão ser alteradas em nosso projeto? Apenas uma, nosso `RepositorioDeProdutos`.



Fig. 4.8: Acesso a dados do banco ou planilha.

Como o comportamento está isolado, encapsulado, fica fácil evoluir sem causar efeitos colaterais indesejados. Para a classe `Main`, pouco importa se quando ela chama `repositorio.lista()` a listagem é carregada de uma planilha ou base de dados do *MySQL*, *PostgreSQL*, *Oracle* ou qualquer outro. Pouco importa se estamos usando *JDBC* ou alguma ferramenta que simplifique o trabalho, como o *hibernate*. O que importa para quem chama o método `lista` é que ele retorne a listagem de produtos, e ponto.

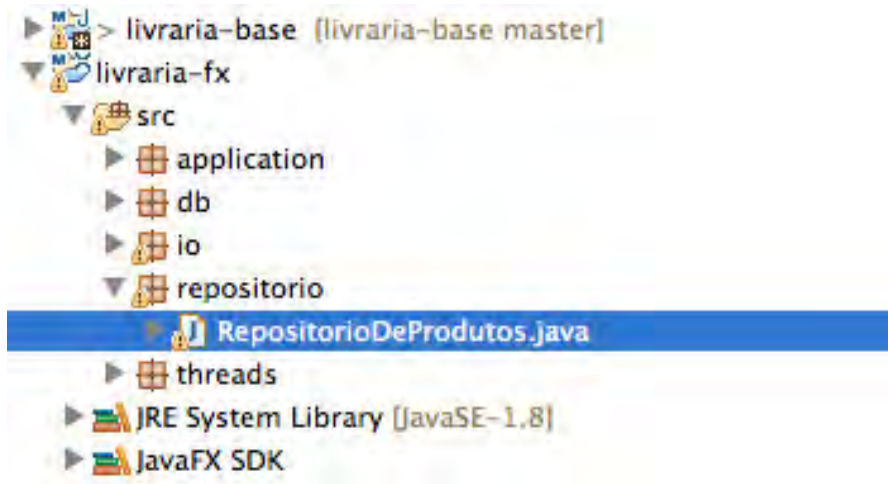
Essa boa pratica é tão conhecida e utilizada no dia a dia que acabou virando um dos mais importantes *Design Patterns*; ele é usado em todos os projetos que conheço! Talvez você já tenha ouvido falar do termo **DAO**, eis seu significado: é um acrônimo para **Data Access Object**.

Mesmo sem saber, já estamos usando o *design pattern DAO*, mas é muito comum e recomendado seguirmos o padrão de nome desse tipo de classe. Como ela encapsula acesso a dados da classe `Produto`, por padrão normalmente a chamamos de `ProdutoDAO`.

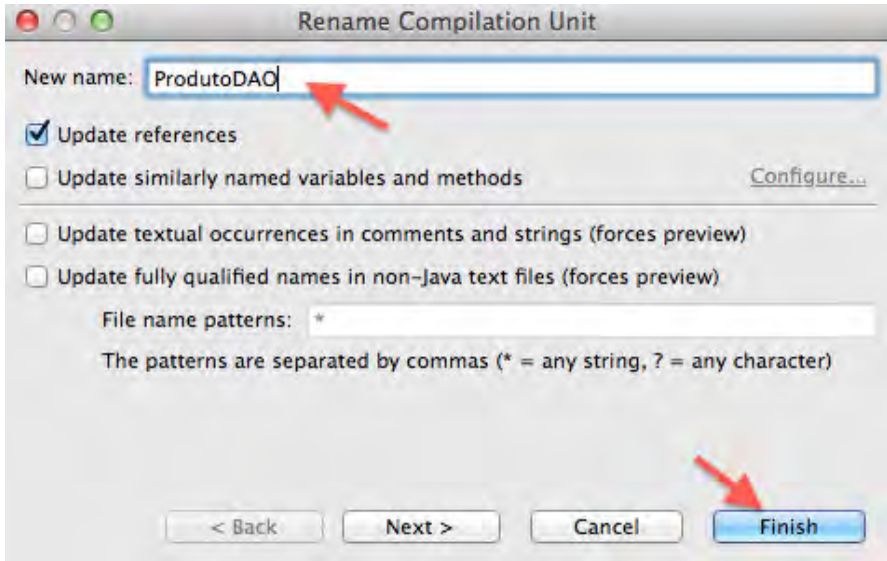
Note que é muito similar ao que fizemos com a `ConnectionFactory`, que é outro *design pattern* que conhecemos. Como ela é uma fábrica de `Connection`, recebe como prefixo o nome da classe e como sufixo a palavra `Factory`.

Agora que sabemos a convenção, vamos fazer a alteração em nosso código. Para renomear uma classe pelo Eclipse, basta selecionar essa classe

pelo *package explorer* (view lateral) e pressionar F2.



Uma janela chamada *Rename Compilation Unit* deve ser exibida. Para concluir, mudamos o nome para `ProdutoDAO` e clicamos em `Finish`.



Nosso DAO completo deve ficar assim:

```
public class ProdutoDAO {  
  
    public ObservableList<Produto> lista() {  
  
        ObservableList<Produto> produtos =  
            observableArrayList();  
        PreparedStatement ps;  
        try (Connection conn =  
            new ConnectionFactory().getConnection()) {  
            ps = conn.prepareStatement(  
                "select * from produtos");  
            ResultSet resultSet = ps.executeQuery();  
  
            while(resultSet.next()) {  
                LivroFisico livro =  
                    new LivroFisico(new Autor());  
                livro.setNome(resultSet.getString("nome"));  
            }  
        }  
    }  
}
```

```

        livro.setDescricao(resultSet.getString(
                                "descricao"));
        livro.setValor(resultSet.getDouble("valor"));
        livro.setIsbn(resultSet.getString("isbn"));
        produtos.add(livro);
    }
    resultSet.close();
    ps.close();
} catch (SQLException e) {
    throw new RuntimeException(e);
}
return produtos;
}

public void adiciona(Produto produto) {
    PreparedStatement ps;

    try (Connection conn =
        new ConnectionFactory().getConnection()) {
        ps = conn.prepareStatement("insert into produtos
            (nome," + " descricao, valor, isbn)
            values (?, ?, ?, ?)");

        ps.setString(1, produto.getNome());
        ps.setString(2, produto.getDescricao());
        ps.setDouble(3, produto.getValor());
        ps.setString(4, produto.getIsbn());

        ps.execute();
        ps.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
}

```

O único impacto em nossa classe `Main` será na linha em que usávamos o `RepositorioDeProdutos`:

```
ObservableList<Produto> produtos =
```

```
new RepositorioDeProdutos().lista();
```

Agora que ele chama `ProdutoDAO`, a linha ficará assim:

```
ObservableList<Produto> produtos = new ProdutoDAO().lista();
```

Mas não se preocupe em alterar manualmente, o Eclipse já fez isso para você.

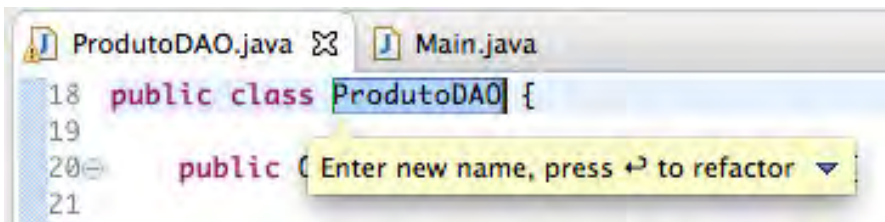
O fluxo agora é:



Fig. 4.11: Acesso a dados encapsulado em um DAO.

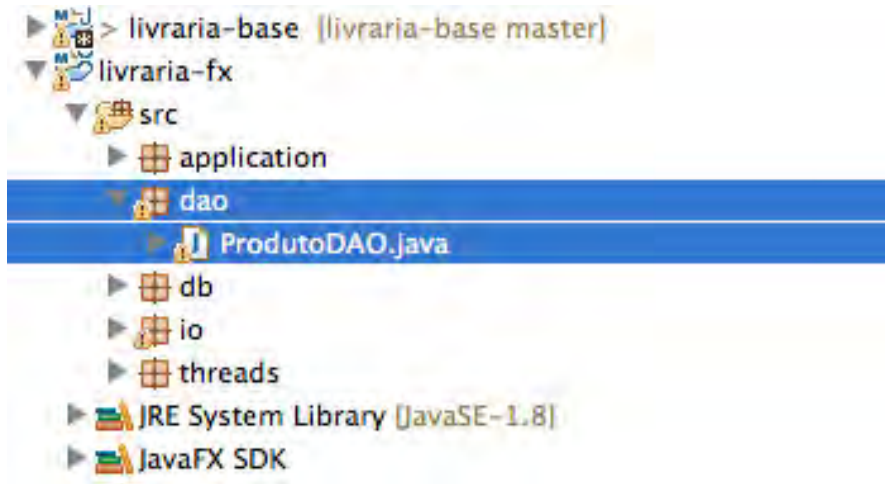
OUTRAS ALTERNATIVAS PARA RENOMEAR CLASSES

A forma que vimos é uma das muitas de se renomear uma classe pela IDE. Outra opção bastante comum pelo Eclipse é utilizando o atalho `Control + Shift + R` (*rename*). A diferença é que o utilizamos diretamente na classe, com o cursor sob seu nome:



Basta mudar seu nome e depois pressionar `Enter`. Todos que fazem referência para essa classe também serão atualizados automaticamente.

Também podemos renomear o pacote do nosso `ProdutoDAO`, que atualmente se chama `repositorio`. Vamos alterar para `dao`. O processo será o mesmo, clicamos no pacote, pressionamos `F2`, escolhemos um novo nome e depois `Finish`. Ele ficará assim:



A vantagem de utilizar essas convenções de nomes é que é muito mais fácil para quem *cair de paraquedas* no projeto saiba se contextualizar. Se você entrar hoje em um projeto e precisar alterar a forma que um `Usuario` é persistido, qual classe você vai abrir? O `UsuarioDAO`. Não tem erro, você não precisa saber nada sobre o projeto para saber disso.

CAPÍTULO 5

Threads e Paralelismo

A consulta ao banco de dados que alimenta nossa listagem de produtos é suficientemente rápida, afinal estamos trabalhando com poucos valores. Mas já parou para pensar o que aconteceria se ela demorasse para responder? Outro ponto que poderia causar um problema é a exportação dessa lista para o arquivo CSV. Quanto mais produtos, mais ela deve demorar. Como nossa aplicação reage enquanto uma tarefa está demorando para ser executada?

Neste capítulo, veremos esse problema em prática, assim como uma alternativa para resolvê-lo. Vamos conhecer um pouco da forma como o Java faz para executar ações em paralelo, com uso de `Threads`, classes anônimas e muitos lambdas do Java 8. Se você ainda não está acostumado com esse recurso, com certeza vai aproveitar bastante os exemplos deste capítulo para praticar.

5.1 PROCESSAMENTO DEMORADO, E AGORA?

Uma forma simples de experienciar isso em prática seria forçando nossa aplicação a dormir por alguns instantes, no momento em que geramos o arquivo CSV. Assim conseguimos simular que a função demora cerca de 20 segundos para ser executada.

```
button.setOnAction(event -> {  
    Thread.sleep(20000);  
    exportaEmCSV(produtos);  
});
```

Veja que quem cuidou desse trabalho foi o método estático `sleep`, presente na classe `Thread` (*que muito em breve será desmistificada*). Essa classe está presente no pacote `java.lang`, portanto não precisa ser importada. Ele recebe como parâmetro o tempo em milissegundos em que deve dormir (temporariamente pausar) a linha de execução atual, e o força a tratar ou declarar uma `InterruptedException`.

```
button.setOnAction(event -> {  
    try {  
        Thread.sleep(20000);  
    } catch (InterruptedException e) {  
        System.out.println("Ops, ocorreu um erro: " + e);  
    }  
    exportaEmCSV(produtos);  
});
```

Depois dessa alteração, a ação de exportar o arquivo `produtos.csv` vai levar pelo menos 20 segundos para ser executada. Execute a aplicação para ver isso acontecendo.

O que acontece se você tentar redimensionar a janela, clicar nas linhas da tabela, ou fazer qualquer coisa similar enquanto a ação do *button* está sendo executada?



Fig. 5.1: Aplicação não respondendo, totalmente travada.

A aplicação fica totalmente travada!

A necessidade de executar ações em paralelo

Podem ser muitas as situações em que precisamos executar ações em paralelo. Em um sistema operacional, por exemplo, fazemos isso o tempo todo! Podemos fazer um download e ao mesmo tempo navegar na internet. Já pensou como seria ter que esperar o download terminar para só depois fazer qualquer outra coisa? O próprio sistema operacional já gerencia todos os programas executados em vários processos paralelos, não precisamos nos preocupar com isso.

Em nossa aplicação Java também podemos executar ações em paralelo, como exportar os dados da listagem sem ter que travar as demais funções e, enquanto isso, mostrar o avanço da função de exportar para dar um retorno (*feedback*) ao usuário.

5.2 TRABALHANDO COM THREADS EM JAVA

Em Java podemos criar linhas de execuções paralelas utilizando a classe `Thread`, do pacote `java.lang`, a mesma que utilizamos para fazer a apli-

cação dormir por 20 segundos.

Para criar e iniciar a execução de uma *thread*, só precisamos instanciar um objeto desse tipo e chamar seu método `start`:

```
Thread thread = new Thread();  
thread.start();
```

Mas há uma questão, o que essa *thread* faz? Criamos uma nova linha de execução, mas em nenhum momento dissemos o que ela deve fazer. Ela morrerá assim que for criada.

A classe `Thread` tem um construtor que recebe como argumento um objeto com o código que queremos executar, tudo que precisamos fazer é criar essa classe. Vamos chamá-la de `ExportadorDeCSV`, já que é isso o que queremos fazer em paralelo.

```
package threads;  
  
public class ExportadorDeCSV {  
  
    public void exporta() {  
        // código que deve ser executado em paralelo  
    }  
}
```

Vamos, agora, no momento de criar uma `Thread`, passar o exportador como argumento:

```
ExportadorDeCSV exportador = new ExportadorDeCSV();  
Thread thread = new Thread(exportador);  
thread.start();
```

Tudo parece certo, mas esse código não vai compilar. O erro será:

```
constructor java.lang.Thread.Thread(java.lang.Runnable) is  
not applicable. (argument mismatch; threads.ExportadorDeCSV  
cannot be converted to java.lang.Runnable)
```

Faz sentido, afinal, como a `Thread` saberá qual o método que deve ser executado? E se a classe `ExportadorDeCSV` tivesse dez métodos, como ela escolheria um?

5.3 O CONTRATO RUNNABLE

Para ensinar à classe `Thread` qual método deve ser executado e dar uma garantia ao compilador de que esse método está declarado em nossa classe, usamos um **contrato**, uma interface Java. Não precisamos criar uma nova interface; se você olhar com atenção na mensagem gerada pelo erro de compilação, você perceberá que esse contrato já existe e é esperado. Estamos falando da interface `java.lang.Runnable`.

Uma classe que assina o contrato `Runnable` assume a responsabilidade de ser um executável. Essa interface tem um único método, chamado `run`. Por esse motivo, a `Thread` sabe exatamente como executá-la.

Vamos modificar nosso `ExportadorDeCSV` para implementar essa interface. Agora que sabemos como o método deve se chamar, trocaremos o nome do método de `exporta` para `run`:

```
package threads;

public class ExportadorDeCSV implements Runnable {

    @Override
    public void run() {
        // código que deve ser executado em paralelo
    }
}
```

Ótimo, nosso código passou a compilar.

Para testar, coloque um `System.out.println` no método `run`. Algo simples, como:

```
package threads;

public class ExportadorDeCSV implements Runnable {

    @Override
    public void run() {
        System.out.println("Rodando em paralelo!");
    }
}
```

Agora, crie a classe `TestandoThread`, com o seguinte método `main`:

```
package threads;

public class TestandoThread {

    public static void main (String... args) {

        ExportadorDeCSV exportador = new ExportadorDeCSV();
        Thread thread = new Thread(exportador);
        thread.start();
        System.out.println("Terminei de rodar o main");
    }
}
```

Execute o código para ver o resultado. Deverá ser algo como:

```
Rodando em paralelo!
Terminei de rodar o main
```

A ordem pode variar, já que são duas linhas de execuções em paralelo. Mas não se preocupe com isso agora. O importante é as duas mensagens terem sido impressas no console. Note que em nenhum momento chamamos o método `run` do nosso `ExportadorDeCSV`, mas ao chamar o método `start` da classe `Thread` ela fez isso por nós.

PARA SABER MAIS: ESCALONADOR DE THREADS

Como executamos ações em paralelo quando existe um único processador na máquina? O segredo está no **scheduler** de *threads*. Ele é um escalonador que pega todas as *threads* que precisam ser executadas e faz seu processador alternar entre elas. Isso acontece tão rápido que parece que as duas (ou mais) *threads* estão rodando ao mesmo tempo. Essa alternância que ele faz é conhecida como **context switch**, ou troca de contexto.

Quando existem dois ou mais processadores, a máquina virtual do Java e grande parte (se não todos) os sistemas operacionais conseguem tirar proveito disso. Nesse caso, verdadeiramente teremos execuções paralelas, ainda escalonadas pelo **scheduler** do Java. Isso porque podemos ter 20 *threads* sendo executadas em dois processadores, por exemplo. A diferença é que a troca de contexto acontecerá entre eles e não em um único processador.

5.4 THREADS COM CLASSES ANÔNIMAS E LAMBDAS

No lugar de criarmos a classe `ExportadorDeCSV` implementando um `Runnable`, poderíamos fazer isso diretamente utilizando uma *classe anônima*. Isso é extremamente comum quando estamos trabalhando com `Threads`. O código ficaria assim:

```
Runnable exportador = new Runnable() {
    @Override
    public void run() {
        System.out.println("Rodando em paralelo!");
    }
};

Thread thread = new Thread(exportador);
thread.start();
```

No lugar de extrair as variáveis intermediárias, é natural fazer isso diretamente:

```
new Thread(new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("Rodando em paralelo!");  
    }  
}).start();
```

O código completo da classe `TestandoThread` fica:

```
package threads;  
  
public class TestandoThread {  
  
    public static void main (String... args) {  
  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                System.out.println("Rodando em paralelo!");  
            }  
        }).start();  
        System.out.println("Terminei de rodar o main");  
    }  
}
```

Vamos executar o código para ver o resultado? A mesma coisa. As duas mensagens serão impressas, podendo variar um pouco a ordem, já que são executadas em paralelo.

O problema desse código está tanto na verbosidade quanto na sintaxe, que é bem poluída. Chegam a chamar esse tipo de código de *“problema vertical”*, por causa da quantidade de linhas.

Essa foi uma das motivações para introdução das *expressões lambdas*, do Java 8. Assim como já estamos usando esse recurso com a ação do `button`, podemos usar com o `Runnable`. O segredo está no conceito de **interface funcional**, que é como uma interface de um único método abstrato é chamada no Java 8.

Em outras palavras, como um `Runnable` só tem um método abstrato, podemos usar a sintaxe do `lambda`, já que o Java saberá exatamente qual

método está sendo implementado. O código ficará assim:

```
package threads;

public class TestandoThread {

    public static void main (String... args) {

        new Thread(() -> {
            System.out.println("Rodando em paralelo!");
        }).start();
        System.out.println("Terminei de rodar o main");
    }
}
```

O que acha, mais simples? A sintaxe pode parecer estranha no começo, mas sem dúvida o código está mais enxuto. Como existe apenas uma única instrução dentro da expressão lambda, podemos tirar as chaves, ponto e vírgula e colocar tudo em um único *statement*:

```
package threads;

public class TestandoThread {

    public static void main (String... args) {

        new Thread(() -> System.out
            .println("Rodando em paralelo!")).start();

        System.out.println("Terminei de rodar o main");
    }
}
```

O foco do livro não é Java 8, mas se você ainda não está acostumado com as expressões lambdas e outros recursos, não deixe de praticar. Ficou com alguma dúvida? Lembre-se de mandar um e-mail na lista do grupo.

5.5 EXPORTANDO EM UMA THREAD SEPARADA

Pronto, agora que conhecemos um pouco mais sobre `Threads` em Java, podemos atacar o problema da lógica de exportar em `CSV`. Para não travar a aplicação, podemos rodar toda a lógica de exportar a lista de produtos em paralelo!

Vamos aplicar essa mudança à classe `Main`. A *action* do `button` ficará assim:

```
Button button = new Button("Exportar CSV");

button.setOnAction(event -> {

    new Thread(() -> {
        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            System.out.println("Ops, ocorreu um erro: "+ e);
        }
        exportaEmCSV(produtos);
    }).start();
});
```


SE VOCÊ ESTÁ COM JAVA 7...

Em Java 7, com uso das classes anônimas, esse código ficaria assim:

```
Button button = new Button("Exportar CSV");

button.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {

        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(20000);
                } catch (InterruptedException e) {
                    System.out.println("Ops,
                        ocorreu um erro: "+ e);
                }
                exportaEmCSV(produtos);
            }
        }).start();
    }
});
```

Vale lembrar que você não precisa ter o Java 8 instalado para rodar esses exemplos, no final do capítulo vou mostrar o código completo com Java 7 também. Aproveite para fazer uma comparação entre as duas sintaxes.

Para melhorar um pouco a legibilidade, vamos extrair o código do `Thread.sleep`, que faz a aplicação dormir por 20 segundos, para dentro de um método:

```
private void dormePorVinteSegundos() {
    try {
        Thread.sleep(20000);
    }
```

```

    } catch (InterruptedException e) {
        System.out.println("Ops, ocorreu um erro: "+ e);
    }
}

```

Agora o código da *action* de exportar em CSV fica assim:

```

button.setOnAction(event -> {
    new Thread(() -> {
        dormePorVinteSegundos();
        exportaEmCSV(produtos);
    }).start();
});

```

Com essas alterações, o código completo da classe `Main` deve ficar da seguinte forma:

```

package application;

// imports omitidos

@SuppressWarnings({ "unchecked", "rawtypes" })
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) {

        Group group = new Group();
        Scene scene = new Scene(group, 690, 510);

        ObservableList<Produto> produtos =
            new ProdutoDAO().lista();

        TableView<Produto> tableView =
            new TableView<>(produtos);

        TableColumn nomeColumn = new TableColumn("Nome");
        nomeColumn.setMinWidth(180);
        nomeColumn.setCellValueFactory(
            new PropertyValueFactory("nome"));
    }
}

```

```
TableColumn descColumn = new TableColumn("Descrição");
descColumn.setMinWidth(230);
descColumn.setCellValueFactory(
    new PropertyValueFactory("descricao"));

TableColumn valorColumn = new TableColumn("Valor");
valorColumn.setMinWidth(60);
valorColumn.setCellValueFactory(
    new PropertyValueFactory("valor"));

TableColumn isbnColumn = new TableColumn("ISBN");
isbnColumn.setMinWidth(180);
isbnColumn.setCellValueFactory(
    new PropertyValueFactory("isbn"));

tableView.getColumns().addAll(nomeColumn, descColumn,
    valorColumn, isbnColumn);

final VBox vbox = new VBox(tableView);
vbox.setPadding(new Insets(70, 0, 0 ,10));

Label label = new Label("Listagem de Livros");

label.setFont(Font
    .font("Lucida Grande", FontPosture.REGULAR, 30));

label.setPadding(new Insets(20, 0, 10, 10));

Button button = new Button("Exportar CSV");
button.setLayoutX(575);
button.setLayoutY(25);

button.setOnAction(event -> {
    new Thread(() -> {
        dormePorVinteSegundos();
        exportaEmCSV(produtos);
    }).start();
});
```

```
group.getChildren().addAll(label, vbox, button);

primaryStage.setTitle(
    "Sistema da livraria com Java FX");
primaryStage.setScene(scene);
primaryStage.show();
}

private void exportaEmCSV(ObservableList<Produto> produtos){
    try {
        new Exportador().paraCSV(produtos);
    } catch (IOException e) {
        System.out.println("Erro ao exportar: "+ e);
    }
}

private void dormePorVinteSegundos() {
    try {
        Thread.sleep(20000);
    } catch (InterruptedException e) {
        System.out.println("Ops, ocorreu um erro: "+ e);
    }
}

public static void main(String[] args) {
    launch(args);
}
}
```

Pronto para testar mais uma vez? Execute a aplicação e mande exportar a lista de produtos como CSV. Você vai perceber que, mesmo que esse código demore 20 segundos para ser executado, o restante da aplicação continua funcionando sem travar. Experimente redimensionar a janela, clicar nas linhas da tabela etc. Tudo continua funcionando.

5.6 UM POUCO MAIS SOBRE AS THREADS

Trabalhar com `Threads` nem sempre é um sonho. Quando estamos trabalhando com linhas de execuções diferentes, a complexidade de nosso código aumenta. Há ainda uma preocupação constante, que são os temíveis problemas de concorrência. O que poderia dar errado?

Dê uma boa olhada no seguinte código:

```
public class LivroFisico extends Livro
    implements Promocional {

    private boolean jaTeveDesconto = false;
    // parte do código omitido

    public boolean aplicaDescontoDe(double porcentagem) {
        if (jaTeveDesconto) {
            return false;
        }
        jaTeveDesconto = true;
        // restante do código do método
    }
}
```

Grande parte do código está omitido, mas só precisamos focar nessa parte. Essa classe possui um método chamado `aplicaDescontoDe`, que da forma que está só aplica o desconto uma única vez por instância. Temos um atributo boolean para nos ajudar, o `jaTeveDesconto`. O código é bem simples e tudo parece certo. De certa maneira está, o problema só vai aparecer quando duas ou mais `Thread` concorrerem pelo acesso desse método.

Imagine que queremos fazer um processamento em massa, em todos os nossos livros, chamando o método `aplicaDescontoDe`. Esse processamento será executado em paralelo, com diversas `Threads`. O que acontece se duas `Threads` passarem “ao mesmo tempo” pelo `if` que valida se ele já teve desconto?

Isso pode resultar em um desastre. Como a primeira `Thread` pode ainda não ter chegado à linha que atribui `true` ao valor do atributo `jaTeveDesconto`, as duas vão passar pelo `if`. Em outras palavras, vamos aplicar o desconto duas vezes para o mesmo objeto.

Claro, esse é um exemplo bem simples e o problema não aconteceria sempre. Esse código poderia ser executado por um mês sem nunca acontecer um acesso concorrente, até que um belo dia o *scheduler* de *threads* passasse duas para o mesmo objeto.

Como impedir isso? O Java nos oferece um recurso interessante, um meio de garantir que apenas uma *Thread* acessará determinado bloco ou método. Se outra *Thread* chegar, ela vai precisar esperar a primeira terminar todo o trabalho para depois começar o seu. O acesso é **sincronizado**.

```
public class LivroFisico extends Livro
    implements Promocional {

    private boolean jaTeveDesconto = false;
    // parte do código omitido

    public synchronized boolean
    aplicaDescontoDe(double porcentagem) {
        if (jaTeveDesconto) {
            return false;
        }
        jaTeveDesconto = true;
        // restante do código do método
    }
}
```

Note no uso da palavra chave `synchronized`. É ela quem dá essa característica ao método `aplicaDescontoDe`. Com isso, já estamos prevenindo acesso concorrente a essa lógica, garantimos que não existirá concorrência nela. Normalmente chamamos de **thread safe** uma classe que está protegida contra acesso concorrente.

BLOCO SINCRONIZADO

Utilizando o modificador `synchronized` em um método, estamos dizendo que todo o código daquele método é sincronizado. Faz sentido. Mas como dizer que apenas um bloco de código deve ser sincronizado?

Há uma outra forma de utilizar o `synchronized`, definindo um bloco de código que recebe qual é o objeto que será bloqueado:

```
public boolean aplicaDescontoDe(double porcentagem) {  
    synchronized (this) {  
        if (jaTeveDesconto) {  
            return false;  
        }  
        jaTeveDesconto = true;  
        // restante do código do método  
    }  
}
```

Agora apenas o `if` está no bloco do `synchronized`, no lugar de todo o método.

5.7 GARBAGE COLLECTOR

Não podemos falar de *threads* sem comentar sobre uma `Thread` bastante conhecida em Java, responsável por jogar fora os objetos que não estão mais sendo referenciados em nosso código. Estamos falando do famoso *Garbage Collector* (coletor de lixo).

Já não é uma novidade que quando criamos um novo objeto, suas informações serão mantidas em memória. Atribuímos esse objeto do tipo `Autor`, por exemplo, a uma variável para futuramente termos uma forma de nos referenciarmos a ele:

```
Autor autor = new Autor();
```

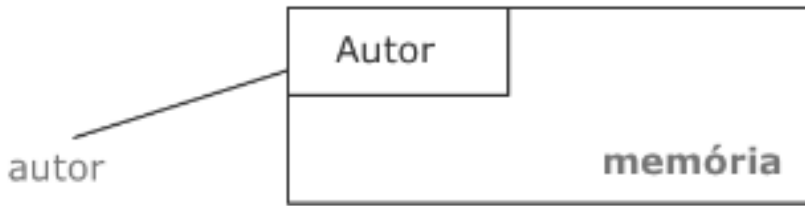


Fig. 5.2: Variável autor referenciando o objeto em memória.

Mas o que acontece com o objeto em memória quando deixamos de nos referenciar a ele? Por exemplo, ao atribuir um novo autor para essa variável:

```
Autor autor = new Autor();  
Autor outro = new Autor();  
autor = outro;
```

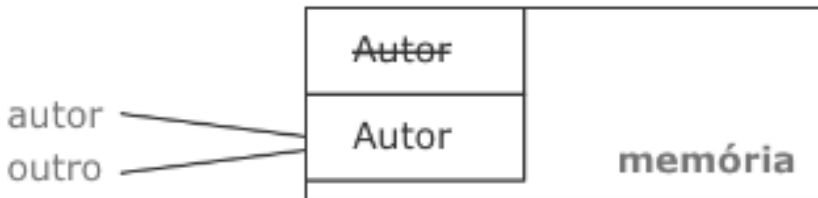


Fig. 5.3: Duas variáveis referenciando o mesmo objeto.

A partir desse momento, ninguém mais está referenciando esse objeto, direta ou indiretamente. Isso significa que ele não existe mais em memória? Não necessariamente. Significa com toda a certeza que ele não está mais acessível, mas não que ele foi removido da memória.

Vale lembrar que quem cuida desse trabalho é o *Garbage Collector* (GC), que é uma `Thread`. Mas quando ela é executada? A todo momento? Provavelmente não, isso teria um custo bem grande e totalmente

desnecessário. Ela é executada de forma estratégica, quando for preciso e conveniente para a JVM, podendo depender de uma implementação para outra.

Não há uma forma de obrigar a *thread* do *Garbage Collector* a ser executada. A classe `System` até possui um método estático chamado `gc`, mas não se engane, chamá-lo não significa que o GC será executado. Isso apenas sugere para a JVM que rode o GC, ela pode ou não aceitar. O uso desse método normalmente é desencorajado, são pouquíssimas as situações em que usá-lo fará alguma diferença em sua aplicação, o ideal é não depender disso.

5.8 JAVA FX ASSÍNCRONO

Agora que já conhecemos um pouco mais sobre a API de `Threads` do Java, podemos voltar ao nosso código. Resolvemos o problema de a ação demorada travar a aplicação, mas não estamos dando uma resposta muito adequada para o cliente que pediu a lista como um `CSV`. Ele clica no botão, olha no diretório e o arquivo `produtos.csv` ainda não está lá. Só depois de alguns segundos o arquivo é gerado.

Para melhorar isso, podemos mostrar uma mensagem indicando que o arquivo está sendo exportado e logo em seguida atualizá-la para indicar que o trabalho foi concluído.

Vamos começar criando uma `Label` chamada `progresso`, para mostrar este texto.

```
Label progresso = new Label();
```

Note que não atribuímos um texto inicial para ela, seu texto será adicionado e atualizado dinamicamente pela ação de exportar produtos.

Lembre-se de que, para deixá-la visível, precisamos adicioná-la ao `group` de elementos dessa tela:

```
group.getChildren().addAll(  
    label, vbox, button, progresso);
```

Ótimo, agora podemos tentar adicionar e atualizar seu texto dentro da `Thread` que acontece no `setOnAction` do botão de exportar. Uma possibilidade seria:

```
button.setOnAction(event -> {  
    new Thread(() -> {  
        progresso.setText("Exportando...");  
        dormePorVinteSegundos();  
        exportaEmCSV(produtos);  
        progresso.setText("Concluído!");  
    }).start();  
});
```

Com isso, queremos que, ao iniciar, nossa `Thread` adicione o texto `Exportando...` no `Label`, que chamamos de `progresso`. Quando concluir, mude seu texto para `Concluído!`. Parece fazer sentido, não acha? Mas ao executar:

```
Exception in thread "Thread-5"  
java.lang.IllegalStateException:  
Not on FX application thread;
```

Isso aconteceu porque não podemos adicionar ou atualizar o texto de um elemento do Java FX em uma `Thread` que não seja do Java FX, ou sua própria `Thread` principal.

5.9 TRABALHANDO COM A CLASSE TASK

Para nos ajudar nesse trabalho, o Java FX oferece uma classe chamada `Task`. Essa classe é bem útil quando estamos trabalhando de forma assíncrona, como já veremos em prática. Ela tem um único método abstrato, chamado `call`.

Comumente instanciamos uma `Task` utilizando classes anônimas:

```
Task<Void> task = new Task<Void>() {  
    @Override  
    protected Void call() throws Exception {  
        // algum trabalho  
        return null;  
    }  
};
```

PERGUNTA: POR QUE NÃO UM LAMBDA?

Talvez você se pergunte por que criamos a `Task` usando uma classe anônima e não com uma expressão lambda? Porque funciona com o `Runnable`, mas não com a `Task`. Não vou falar a resposta agora, tente descobrir. Ok? Não se preocupe, em breve eu lhe conto.

A `Task` pode ter um retorno, mas note que, como não queremos retornar nada, utilizamos o tipo `Void` (classe *wrapper* do já tão conhecido `void`).

Vamos modificar nosso código para que, no lugar de executar o método de dormir por 20 segundos e exportar dentro de um `Runnable`, ela faça isso dentro da `Task`.

No lugar de fazer:

```
button.setOnAction(event -> {  
    new Thread(() -> {  
        dormePorVinteSegundos();  
        exportaEmCSV(produtos);  
    }).start();  
});
```

Vamos quebrar esse trabalho em duas partes, primeiro criando uma instância de `Task` cujo método `call` fará esse trabalho. Ela deve ficar dentro do `setOnAction`, no botão de exportar:

```
button.setOnAction(event -> {  
  
    Task<Void> task = new Task<Void>() {  
        @Override  
        protected Void call() throws Exception {  
            dormePorVinteSegundos();  
            exportaEmCSV(produtos);  
            return null;  
        }  
    };  
  
});
```

Agora, ainda dentro do `setOnAction`, vamos criar e iniciar uma nova `Thread` passando essa `Task` como argumento.

```
button.setOnAction(event -> {  
  
    Task<Void> task = new Task<Void>() {  
        @Override  
        protected Void call() throws Exception {  
            dormePorVinteSegundos();  
            exportaEmCSV(produtos);  
            return null;  
        }  
    };  
  
    new Thread(task).start();  
});
```

O trecho de código do botão de exportar ficará assim:

```
Button button = new Button("Exportar CSV");  
  
Label progresso = new Label();  
  
button.setOnAction(event -> {  
  
    Task<Void> task = new Task<Void>() {  
        @Override  
        protected Void call() throws Exception {  
            dormePorVinteSegundos();  
            exportaEmCSV(produtos);  
            return null;  
        }  
    };  
  
    new Thread(task).start();  
});
```

Podemos executar nossa aplicação para confirmar que tudo está funcionando.

RESPOSTA: POR QUE NÃO UM LAMBDA?

Você provavelmente já descobriu o motivo, mas não podemos implementar a `Task` como um lambda porque ela não é uma **interface funcional**. Ela não é nem mesmo uma **interface**: se você olhar seu código perceberá que se trata de uma classe anônima. Nós só podemos usar a sintaxe do lambda com interfaces funcionais, sem exceções.

Mas e quanto ao texto da Label?

Mudamos para `Task` e tudo continua funcionando, mas e quanto ao texto da `Label` de progresso? O problema ainda não foi resolvido.

Agora que estamos usando a API do Java FX para executar código assíncrono, podemos tirar bastante proveito de alguns recursos. Por exemplo, há um método chamado `setOnRunning` em `Task` que nos permite executar um evento quando a *task* está prestes a ser executada.

```
task.setOnRunning(new EventHandler<WorkerStateEvent>() {  
    @Override  
    public void handle(WorkerStateEvent e) {  
        // alguma ação aqui  
    }  
});
```

O `EventHandler` sim é uma interface funcional, portanto, podemos tirar proveito do lambda. O código fica assim:

```
task.setOnRunning( e -> {  
    // alguma ação aqui  
});
```

Nesse momento, podemos atualizar o texto da `Label`, dizendo que o produto está sendo exportado:

```
task.setOnRunning(e -> {  
    progresso.setText("exportando...");  
});
```

Como é uma única expressão, não precisamos das chaves nem do ponto e vírgula:

```
task.setOnRunning(e -> progresso.setText("exportando..."));
```

O código do evento de `setOnAction` ficará assim:

```
button.setOnAction(event -> {

    Task<Void> task = new Task<Void>() {
        @Override
        protected Void call() throws Exception {
            dormePorVinteSegundos();
            exportaEmCSV(produtos);
            return null;
        }
    };

    task.setOnRunning(e -> progresso.setText("exportando..."));

    new Thread(task).start();
});
```

Vamos testar. Após modificar o código, podemos executá-lo e ao clicar em `exportar` o resultado será o esperado:

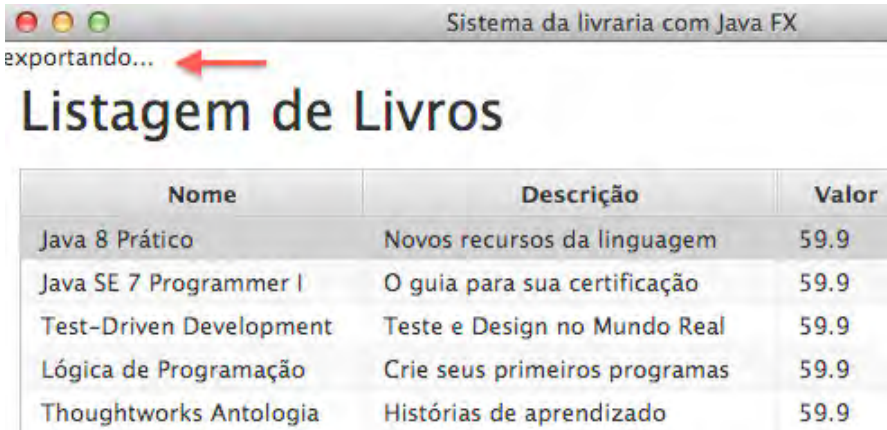


Fig. 5.4: Texto 'exportando..' sendo exibido na tela.

O texto está desalinhado, claro. Mas logo cuidaremos disso. Ainda precisamos fazer com que esse texto seja atualizado no momento em que a *Task* seja concluída. Podemos fazer isso utilizando um outro método de *callback*, o `setOnSucceeded`.

Ele também recebe um `EventHandler` como parâmetro, portanto, vamos de lambda:

```
task.setOnSucceeded(e -> {  
    progresso.setText("concluído!");  
});
```

Agora nosso código está assim:

```
button.setOnAction(event -> {  
  
    Task<Void> task = new Task<Void>() {  
        @Override  
        protected Void call() throws Exception {  
            dormePorVinteSegundos();  
            exportaEmCSV(produtos);  
            return null;  
        }  
    };  
});
```

```
task.setRunning(e -> progresso.setText("exportando..."));

task.setOnSucceeded(e -> progresso.setText("concluído!"));

new Thread(task).start();
});
```

OUTROS MÉTODOS INTERESSANTES

Além do `setRunning` e `setOnSucceeded`, que acabamos de conhecer, a classe `Task` tem outros métodos de *callback* que podem ser muito úteis em seu dia a dia. São eles:

- `setOnCancelled`
- `setOnFailed`
- `setOnScheduled`

Não deixe de conferir a documentação da classe `Task` para saber mais detalhes desses e outros métodos. Além de pesquisar, tente testar alguns deles em seu projeto.

<https://docs.oracle.com/javafx/2/api/javafx/concurrent/Task.html>

Pronto, vamos ver o resultado. Ao clicar em `Exportar CSV` a mensagem `exportando...` é exibida. Alguns segundos depois, o texto é atualizado para `concluído!`. Sucesso.



Fig. 5.5: Texto concluído sendo exibido na tela.

Por fim, podemos alinhar essas mensagens. Faremos isso da mesma forma como foi feito no `button`:

```
Label progresso = new Label();  
progresso.setLayoutX(485);  
progresso.setLayoutY(30);
```

Ao executar a aplicação, clicando em `Exportar CSV` teremos:



Após alguns segundos...



Nome	Descrição	Valor	ISBN
Java 8 Prático	Novos recursos da linguagem	59.9	978-85-66250-46-6
Java SE 7 Programmer I	O guia para sua certificação	59.9	123-45-67890-11-2
Test-Driven Development	Teste e Design no Mundo Real	59.9	123-45-67890-11-2
Lógica de Programação	Crie seus primeiros programas	59.9	978-85-66250-22-0
Thoughtworks Antologia	Histórias de aprendizado	59.9	123-45-67890-11-2
Jogos em HTML5	Explore o mobile e física	59.9	123-45-67890-11-2
UX Design	Introdução e boas práticas	59.9	978-85-66250-48-0

Perfeito. Vimos que trabalhar com `Threads` não é fácil, mas também não é um bicho de sete cabeças. Temos que tomar alguns cuidados extras, claro. Em Java FX, a classe `Task` ajuda bastante nesse trabalho. Ela oferece uma forma flexível e desacoplada de encapsular e executar código paralelo.

5.10 CÓDIGO FINAL COM E SEM LAMBDA

Nosso código final, com Java 8, ficou assim:

```
@SuppressWarnings({ "unchecked", "rawtypes" })
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) {

        Group group = new Group();
        Scene scene = new Scene(group, 690, 510);

        ObservableList<Produto> produtos =
            new ProdutoDAO().lista();

        TableView<Produto> tableView =
```

```
        new TableView<>(produtos);

    TableColumn nomeColumn = new TableColumn("Nome");
    nomeColumn.setMinWidth(180);
    nomeColumn.setCellValueFactory(
        new PropertyValueFactory("nome"));

    TableColumn descColumn = new TableColumn("Descrição");
    descColumn.setMinWidth(230);
    descColumn.setCellValueFactory(
        new PropertyValueFactory("descricao"));

    TableColumn valorColumn = new TableColumn("Valor");
    valorColumn.setMinWidth(60);
    valorColumn.setCellValueFactory(
        new PropertyValueFactory("valor"));

    TableColumn isbnColumn = new TableColumn("ISBN");
    isbnColumn.setMinWidth(180);
    isbnColumn.setCellValueFactory(
        new PropertyValueFactory("isbn"));

    tableView.getColumns().addAll(nomeColumn, descColumn,
        valorColumn, isbnColumn);

    final VBox vbox = new VBox(tableView);
    vbox.setPadding(new Insets(70, 0, 0 ,10));

    Label label = new Label("Listagem de Livros");

    label.setFont(Font
        .font("Lucida Grande", FontPosture.REGULAR, 30));

    label.setPadding(new Insets(20, 0, 10, 10));

    Label progresso = new Label();
    progresso.setLayoutX(485);
    progresso.setLayoutY(30);
```

```

Button button = new Button("Exportar CSV");
button.setLayoutX(575);
button.setLayoutY(25);

button.setOnAction(event -> {

    Task<Void> task = new Task<Void>() {
        @Override
        protected Void call() throws Exception {
            dormePorVinteSegundos();
            exportaEmCSV(produtos);
            return null;
        }
    };

    task.setOnRunning(
        e -> progresso.setText("exportando..."));

    task.setOnSucceeded(
        e -> progresso.setText("concluído!"));

    new Thread(task).start();
});

group.getChildren().addAll(label, vbox, button,
                             progresso);

primaryStage.setTitle(
    "Sistema da livraria com Java FX");
primaryStage.setScene(scene);
primaryStage.show();
}

private void exportaEmCSV(ObservableList<Produto> produtos){
    try {
        new Exportador().paraCSV(produtos);
    } catch (IOException e) {
        System.out.println("Erro ao exportar: "+ e);
    }
}

```

```

    }

    private void dormePorVinteSegundos() {
        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            System.out.println("Ops, ocorreu um erro: " + e);
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}

```

Conforme prometido, a parte que muda do `setOnAction` em Java 7:

```

button.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {

        Task<Void> task = new Task<Void>() {
            @Override
            protected Void call() throws Exception {
                dormePorVinteSegundos();
                exportaEmCSV(produtos);
                return null;
            }
        };

        task.setRunning(new EventHandler<WorkerStateEvent>() {
            @Override
            public void handle(WorkerStateEvent e) {
                progresso.setText("exportando...");
            }
        });

        task.setOnSucceeded(
            new EventHandler<WorkerStateEvent>() {
                @Override

```

```
        public void handle(WorkerStateEvent e) {  
            progresso.setText("concluído!");  
        }  
    });  
  
    new Thread(task).start();  
}  
});
```

Curiosamente, o Eclipse nos oferece uma forma bem simples de converter classes anônimas para expressões lambda e vice-versa. Basta você selecionar o código e utilizar o atalho `Control + 1`. Quando temos um lambda, a opção *Convert to anonymous classe creation* estará disponível.

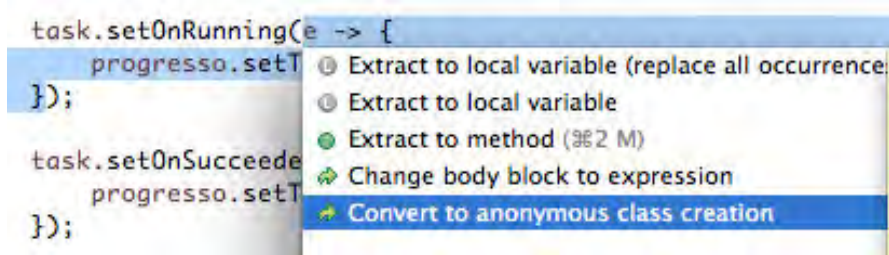


Fig. 5.8: Convertendo lambda para classe anônima no Eclipse.

O mesmo vale para o caminho contrário, quando temos uma classe anônima e queremos transformar em uma expressão lambda, mas nesse caso a opção será *Convert to lambda expression*. Esses recursos da IDE nos ajudam muito, migrar classes anônimas para lambdas nunca foi tão divertido!

CAPÍTULO 6

CSS no Java FX

Nossa aplicação está cada vez mais completa, mas antes de continuar veremos como melhorar ainda mais não só a aparência de nossa listagem de produtos, mas também a qualidade e manutenibilidade de nosso código.

Se você perceber, o estilo padrão do Java FX já é bastante atraente. Sem adicionar cores ou configurar nada, nossa aplicação já está com uma aparência bem aceitável. Repare em detalhes como a borda e a linha selecionada na tabela de produtos. Ao passar o mouse sob o botão `Exportar CSV`, você também pode perceber que sua cor ficará mais suave.

Mas ao utilizar estilos personalizados do Java FX vamos além da aparência. Você perceberá ao longo do capítulo que isso possibilita deixar nosso código ainda mais limpo, enxuto e fácil de manter. Vamos começar?

6.1 SEU PRIMEIRO CSS NO JAVA FX

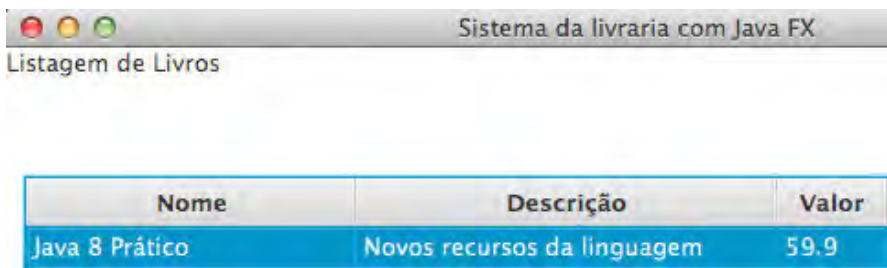
Antes de sair aplicando novos estilos em nosso código, vamos olhar mais de perto a forma como fizemos até agora. Repare como fizemos para personalizar a fonte e adicionar um *padding* em nossa `Label`, por exemplo.

```
Label label = new Label("Listagem de Livros");
label.setFont(Font.font(
    "Lucida Grande", FontPosture.REGULAR, 30));
label.setPadding(new Insets(20, 0, 10, 10));
```

Isso funciona muito bem, mas adiciona uma certa verbosidade a nosso código. O que realmente importa aqui é a primeira linha, onde criamos a `Label`. Queremos um código limpo e simples de entender, pois, quanto mais próximo disso nosso código for, melhor será sua manutenibilidade. Repare no que acontece se tirarmos o estilo desse código:

```
Label label = new Label("Listagem de Livros");
```

Ficou ótimo! Bem simples de entender. Mas o resultado visual...



O ponto é: precisamos atacar os dois pontos. O código precisa ser limpo, mas ao mesmo tempo nossas *views* precisam ser estilizadas para que a aplicação fique mais atraente e tenha uma boa usabilidade.

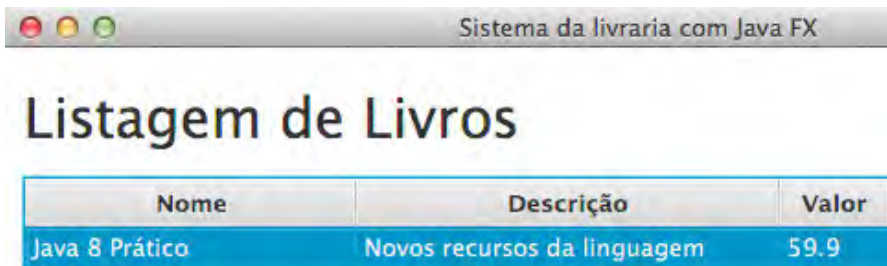
Nossa solução inicial será extrair estes estilos e aplicá-los utilizando o método `setStyle`, como a seguir:


```
Label label = new Label("Listagem de Livros");  
label.setStyle(???);
```

Esse método espera receber como parâmetro as propriedades CSS que devem ser aplicadas no elemento. Um exemplo seria:

```
Label label = new Label("Listagem de Livros");  
  
label.setStyle("-fx-font-size:  
              30px; -fx-padding: 20 0 10 10;");
```

Se você já está habituado com CSS, deve ter percebido que as propriedades aqui são muito parecidas com CSS que utilizamos na web, tendo o prefixo `-fx` sempre presente como uma diferença. Neste caso, o código está fazendo o mesmo que o anterior, onde declarávamos o tamanho da fonte e *padding* programaticamente. Podemos executá-lo para verificar o resultado:



Excelente! Ele está com a aparência esperada, mas agora utilizando as propriedades do CSS.

JAVA FX CSS REFERENCE GUIDE

Na documentação do método `setStyle`, você encontrará um link direto para a página *CSS Reference Guide*, onde os possíveis estilos são explicados detalhadamente. Essa página pode ajudar bastante no dia a dia; consulte-a sempre que quiser conhecer mais detalhes sobre as propriedades e possibilidades dos parâmetros CSS.

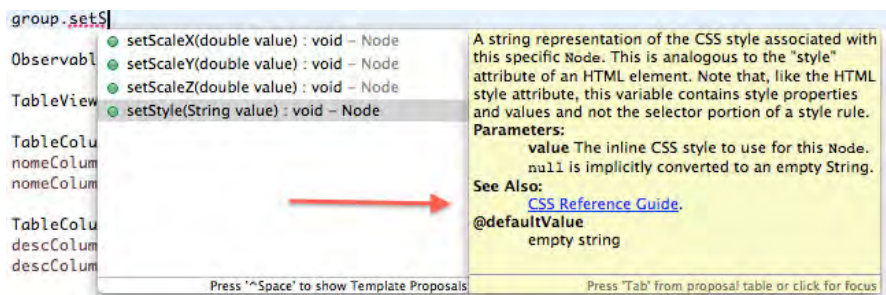


Fig. 6.3: Documentação do método `setStyle` pelo Eclipse

6.2 EXTRAINDO ESTILOS PRA UM ARQUIVO .CSS

A mudança que fizemos já pode ser interessante, mas ainda não resolve nosso problema. Ainda estamos misturando nosso código de estilos com o restante do código! No lugar de utilizar o `setStyle` dessa forma, podemos (e devemos, sempre que possível) isolar nossos estilos no arquivo `application.css`.

Se você criou o projeto utilizando uma versão mais atual do Eclipse, ou qualquer outra IDE moderna, o arquivo `application.css` deve ter sido criado junto com ele. Não se preocupe se isso não aconteceu, esse é um arquivo normal. Se necessário, você pode criá-lo clicando em `New...`, sub-menu `File` e digitando o nome `application.css` ou qualquer outro nome que preferir.

Dentro dele colocaremos o seguinte conteúdo:

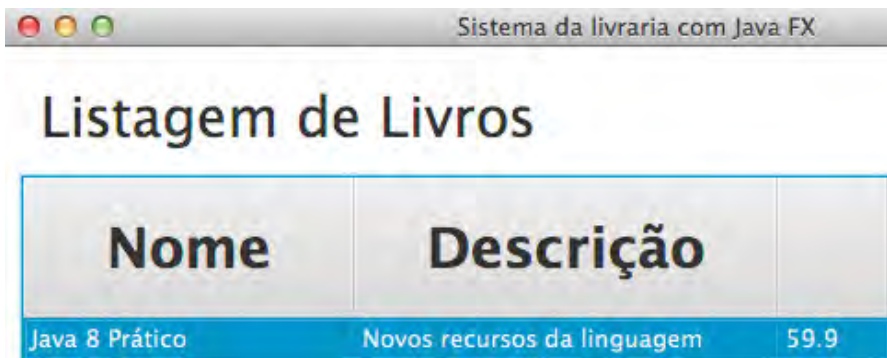
```
.label {  
    -fx-font-size: 30px;  
    -fx-padding: 20 0 10 10;  
}
```

Note que se parece muito com a estrutura do CSS padrão. Aplicamos o efeito em `.label`, pois é a classe padrão (*default*) do elemento `Label`. A maior parte dos elementos visuais do Java FX já tem uma classe *default*, que normalmente é seu próprio nome.

Agora tudo que precisamos fazer é adicionar o arquivo `application.css` como um estilo de nosso cenário. Em nossa classe `Main`, faremos algo como:

```
scene.getStylesheets().add(getClass()  
    .getResource("application.css").toExternalForm());
```

Pronto! Ao rodar esse código, vemos que o resultado não foi bem o que estávamos esperando:



O efeito foi aplicado, mas para todos os *labels* da *view*. Para nossa sorte (ou o oposto disso) o componente `TableView` também utiliza *labels* em suas colunas.

Identificando elementos únicos

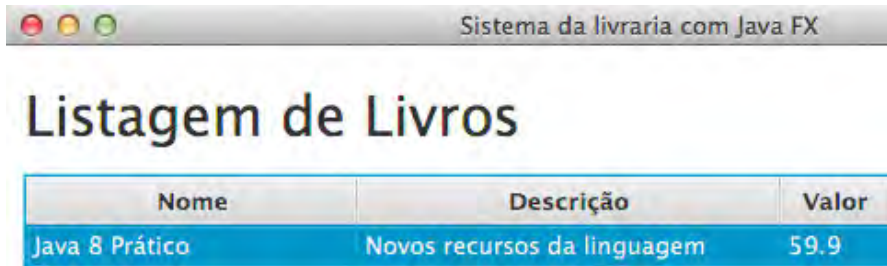
Precisamos de uma forma única de identificar o `Label` que contém nosso texto da listagem, sem que os efeitos dele interfiram nos demais `Labels` que existem, ou possam existir em algum momento futuro, na *view*. Podemos fazer isso adicionando um **identificador único** ao elemento, como a seguir:

```
Label label = new Label("Listagem de Livros");  
label.setId("label-listagem");
```

Como você pode ver, o método `setId` nos ajuda com esse trabalho. Agora que temos uma maneira única de nos referir ao `Label`, basta modificar o arquivo `application.css` para ficar assim:

```
#label-listagem {  
    -fx-font-size: 30px;  
    -fx-padding: 20 0 10 10;  
}
```

O *seletor* de *id* tem a estrutura idêntica, mas recebe um `#` como prefixo, no lugar do `.` utilizado para as classes dos elementos. Execute o código e dessa vez o resultado será o esperado:



Extraindo mais estilos pro `application.css`

Da mesma forma como fizemos para tirar o *padding* na `Label` principal da nossa listagem, podemos tirar do `VBox` que utilizamos para alinhar a tabela. Veja como ele está agora:

```
VBox vbox = new VBox(tableView);  
vbox.setPadding(new Insets(70, 0, 0 ,10));
```

Poderíamos simplesmente remover a chamada ao método `setPadding` e no arquivo `application.css` adicionar o estilo:

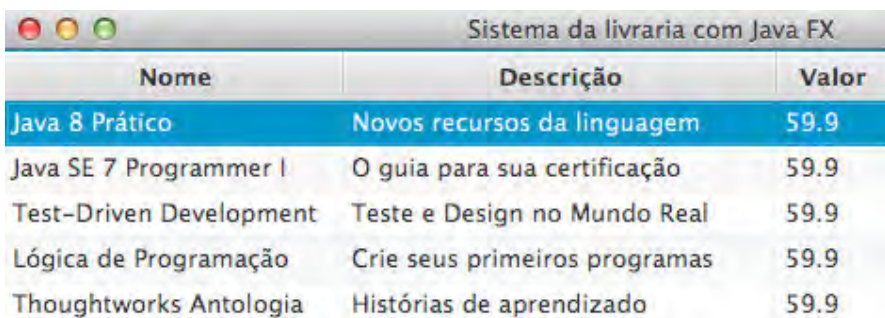
```
.vbox {  
    -fx-padding: 70 0 0 10;  
}
```

Mas há um detalhe importante, nem todo elemento em Java FX tem uma classe seletora já definida.

Em outras palavras, para aplicar um estilo no `Label`, fizemos:

```
.label {  
    # estilos aqui  
}
```

Ou seja, por padrão o `Label` já tem uma classe seletora, chamada `.label`. Já o `VBox` não tem uma classe padrão, ou seja, o CSS que escrevemos com o seletor `.vbox` não vai funcionar. Ao executar a app dessa forma, percebemos que o *padding* não foi aplicado:



Nome	Descrição	Valor
Java 8 Prático	Novos recursos da linguagem	59.9
Java SE 7 Programmer I	O guia para sua certificação	59.9
Test-Driven Development	Teste e Design no Mundo Real	59.9
Lógica de Programação	Crie seus primeiros programas	59.9
Thoughtworks Antologia	Histórias de aprendizado	59.9

Já sabemos como resolver esse problema, certo? Da mesma forma como demos um **identificador único** para o `Label` (*label-listagem*), podemos dar um `ID` para o `VBox`, aplicando um estilo para ele:

```
VBox vbox = new VBox(tableView);  
vbox.setId("vbox");
```

Em nosso arquivo `application.css`, podemos fazer:

```
#vbox {  
    -fx-padding: 70 0 0 10;  
}
```

Ótimo, ao executar o código vemos que o *padding* foi aplicado. Note o `#` do seletor, para identificá-lo como um `ID`.

Podemos agora modificar um pouco o estilo do `Button` de exportar produtos como `CSV`. A primeira mudança pode ser extrair os métodos `setLayouts` para um estilo `CSS`. O código agora está assim:

```
Button button = new Button("Exportar CSV");  
button.setLayoutX(575);  
button.setLayoutY(25);
```

No lugar disso, removemos as chamadas aos dois métodos e adicionamos o mesmo efeito no `application.css`, utilizando as propriedades `-fx-translate-x` e `-fx-translate-y`:

```
.button {  
    -fx-translate-x: 575;  
    -fx-translate-y: 25;  
}
```

Também podemos aplicar uma cor nesse botão, utilizando o `-fx-background-color`. Uma forma de fazer isso seria:

```
.button {  
    -fx-translate-x: 575;  
    -fx-translate-y: 25;  
    -fx-background-color: rgb(31, 149, 206);  
    -fx-text-fill: white;  
}
```

Assim, o `button` ficará com o azul padrão do `Java FX`. Se preferir, você pode aplicar qualquer outra cor. O mesmo para os outros elementos,

claro. Note que também utilizamos a propriedade `-fx-text-fill` para deixar o texto branco. Além desses, existem diversas outras propriedades. Aproveite para explorar o *Java FX CSS Reference Guide* durante seus testes para conhecer mais algumas delas.

Outro detalhe bem interessante é que, da mesma forma que no CSS convencional de navegadores, também podemos adicionar um `:hover` em nossos seletores para definir o estilo para o momento em que o mouse estiver sob o elemento. Para ver isso em prática, vamos dizer que queremos uma transparência maior no `button`, quando estiver em *hover*:

```
.button:hover {  
    -fx-background-color: rgba(31, 149, 206, 0.71);  
}
```

Atualize e execute seu código. Você perceberá a transparência ao passar o cursor do mouse sob o botão.



Fig. 6.7: Button com e sem efeito de hover

O mesmo pode ser feito com o `Label` de progresso, que está assim:

```
Label progresso = new Label();  
progresso.setLayoutX(485);  
progresso.setLayoutY(30);
```

Removemos essas duas linhas e damos um `ID` para esse campo:

```
Label progresso = new Label();  
progresso.setId("label-progresso");
```

Agora basta adicionar o estilo no arquivo de CSS:

```
#label-progresso {  
    -fx-translate-x: 485;  
    -fx-translate-y: 30;  
}
```

EXPLORANDO MAIS SELETORES E EFEITOS

Você pode utilizar diversas outras funções e efeitos, como um `linear-gradient` na cor de seu `button` ou fundo do `scene`. E que tal um efeito de sombra?

```
.button {  
    -fx-background-color:  
        linear-gradient(#61a2b1, #2A5058);  
    -fx-effect: dropshadow(three-pass-box,  
        rgba(0,0,0,0.6), 5, 0.0, 0, 1);  
}
```

Não deixe de testar. Você pode ver esses e diversos outros efeitos no tutorial de Java FX da Oracle:

<https://docs.oracle.com/javase/8/javafx/get-started-tutorial>

Quantidade de produtos e valor em estoque

Para deixar nossa listagem um pouco mais completa, vamos adicionar um novo `Label` com o valor total que temos em estoque e também a quantidade de produtos. Criar a `label` já sabemos. Podemos também adicionar um identificador único, com o método `setID`, para aplicar um estilo no novo elemento. O código ficará assim:

```
Label labelFooter = new Label("Texto vai aqui");  
labelFooter.setId("label-footer");
```

No CSS, podemos alinhar a `label-footer` e também modificar o tamanho de sua fonte, para que tenha um destaque maior.

```
#label-footer {  
    -fx-font-size: 15px;
```



```
-fx-translate-x: 210;  
-fx-translate-y: 480;  
}
```

Precisamos agora adicionar o texto e valores que devem aparecer nesse `Label`. Vamos utilizar o método `format` da classe `String` mais uma vez, para deixar o resultado formatado como a seguir:

```
Label labelFooter = new Label(  
    String.format("Você tem R$%.2f em estoque, " +  
        "com um total de %d produtos.", 100.00, 10));  
  
labelFooter.setId("label-footer");
```

Dessa forma, o resultado será parecido com:

Você tem R\$100.00 em estoque, com um total de 10 produtos.

PARA SABER MAIS: `STRING#FORMAT`

Note que utilizamos `%.2f` para formatar um número de ponto flutuante em duas casas decimais, além do `%d` para exibir o número inteiro. Existem diversas outras opções, como `%s` para `Strings`, `%c` para `chars` e `%t` para datas. Se você não está familiarizado com as convenções do `Formatter` e quiser conhecer um pouco mais, pode dar uma olhada em:

<https://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html>

Não queremos trabalhar com um valor fixo, como acabamos de fazer. No lugar disso, nos interessa recuperar o valor total de todos os produtos de nossa lista. Podemos iterar na lista acumulando todos os valores em uma variável temporária:

```
double valorTotal = 0.0;  
  
for (Produto produto: produtos) {  
    valorTotal += produto.getValor();  
}
```

Não tem muito segredo nesse código, usamos o `+=` para incrementar o valor de cada produto.

CÓDIGO MAIS DECLARATIVO COM JAVA 8

No lugar de escrever esse código imperativo, onde declaramos uma variável intermediária (`valorTotal`) e incrementamos seu valor em um `enhanced-for`, que tal fazer esse mesmo trabalho de uma forma mais declarativa, com um toque de programação funcional do Java 8?

Utilizando a nova API de `Stream`, o código ficaria assim:

```
double valorTotal = produtos.stream()
    .mapToDouble(Produto::getValor).sum();
```

Se você está utilizando Java 8, atualize seu código para testar. O resultado será o mesmo.

Para concluir, podemos recuperar a quantidade de produtos em estoque utilizando o método `size` da lista de `produtos`:

```
Label labelFooter = new Label(
    String.format("Você tem R$%.2f em estoque, " +
        "com um total de %d produtos.",
        valorTotal, produtos.size()));

labelFooter.setId("label-footer");
```

Só precisamos adicionar o `labelFooter` na *view*, e pronto:

```
group.getChildren().addAll(label,
    vbox, button, progresso, labelFooter);
```

Perfeito, execute o código para ver o resultado!

Sistema da livraria com Java FX

Listagem de Livros

Exportar CSV

Nome	Descrição	Valor	ISBN
A Web Mobile	Programa para um mundo de m...	59.9	978-85-66250-23-7
Desbravando Java e OO	Um guia para o iniciante da ling...	59.9	123-45-67890-11-2
Google Android	crie aplicações para celulares	59.9	978-85-66250-02-2
HTML5 e CSS3	Domine a web do futuro	59.9	978-85-66250-05-3
JSF e JPA	Aplicações Java para a web	59.9	978-85-66250-01-5
Java 8 Prático	Novos recursos da linguagem	59.9	978-85-66250-46-6
Java SE 7 Programmer I	O guia para sua certificação	59.9	123-45-67890-11-2
Jogos em HTML5	Explore o mobile e física	59.9	123-45-67890-11-2
Lógica de Programação	Crie seus primeiros programas	59.9	978-85-66250-22-0
O Programador Apaixonado	Construindo uma carreira notável	59.9	978-85-66250-44-2
Test-Driven Development	Teste e Design no Mundo Real	59.9	123-45-67890-11-2
Thoughtworks Antologia	Histórias de aprendizado	59.9	123-45-67890-11-2
UX Design	Introdução e boas práticas	59.9	978-85-66250-48-0

Você tem R\$778.70 em estoque, com um total de 13 produtos.

O código completo da classe `Main` ficou assim:

```
package application;

import io.Exportador;

import java.io.IOException;

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.concurrent.Task;
import javafx.geometry.Insets;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.VBox;
```

```
import javafx.scene.text.*;
import javafx.stage.Stage;
import br.com.casadocodigo.livraria.produtos.Produto;
import dao.ProdutoDAO;

@SuppressWarnings({ "unchecked", "rawtypes" })
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) {

        Group group = new Group();
        Scene scene = new Scene(group, 690, 510);

        scene.getStylesheets().add(getClass()
            .getResource("application.css").toExternalForm());

        ObservableList<Produto> produtos =
            new ProdutoDAO().lista();

        TableView<Produto> tableView =
            new TableView<>(produtos);

        TableColumn nomeColumn = new TableColumn("Nome");
        nomeColumn.setMinWidth(180);
        nomeColumn.setCellValueFactory(
            new PropertyValueFactory("nome"));

        TableColumn descColumn = new TableColumn("Descrição");
        descColumn.setMinWidth(230);
        descColumn.setCellValueFactory(
            new PropertyValueFactory("descricao"));

        TableColumn valorColumn = new TableColumn("Valor");
        valorColumn.setMinWidth(60);
        valorColumn.setCellValueFactory(
            new PropertyValueFactory("valor"));

        TableColumn isbnColumn = new TableColumn("ISBN");
```

```
isbnColumn.setMinWidth(180);
isbnColumn.setCellValueFactory(
    new PropertyValueFactory("isbn"));

tableView.getColumns().addAll(nomeColumn, descColumn,
    valorColumn, isbnColumn);

final VBox vbox = new VBox(tableView);
vbox.setId("vbox");

Label label = new Label("Listagem de Livros");
label.setId("label-listagem");

Label progresso = new Label();
progresso.setId("label-progresso");

Button button = new Button("Exportar CSV");

button.setOnAction(event -> {

    Task<Void> task = new Task<Void>() {
        @Override
        protected Void call() throws Exception {
            dormePorVinteSegundos();
            exportaEmCSV(produtos);
            return null;
        }
    };

    task.setOnRunning(e -> progresso.setText(
        "exportando..."));

    task.setOnSucceeded(e -> progresso.setText(
        "concluído!"));

    new Thread(task).start();
});

double valorTotal = produtos.stream()
```

```
        .mapToDouble(Produto::getValor).sum());

Label labelFooter = new Label(
    String.format("Você tem R$%.2f em estoque, " +
        "com um total de %d produtos.",
        valorTotal, produtos.size()));

labelFooter.setId("label-footer");

group.getChildren().addAll(label,
    vbox, button, progresso, labelFooter);

primaryStage.setTitle(
    "Sistema da livraria com Java FX");
primaryStage.setScene(scene);
primaryStage.show();
}

private void exportaEmCSV(ObservableList<Produto> produtos){
    try {
        new Exportador().paraCSV(produtos);
    } catch (IOException e) {
        System.out.println("Erro ao exportar: "+ e);
    }
}

private void dormePorVinteSegundos() {
    try {
        Thread.sleep(20000);
    } catch (InterruptedException e) {
        System.out.println("Ops, ocorreu um erro: "+ e);
    }
}

public static void main(String[] args) {
    launch(args);
}
}
```

E nosso arquivo `application.css`:

```
#label-listagem {
    -fx-font-size: 30px;
    -fx-padding: 20 0 10 10;
}

#label-progresso {
    -fx-translate-x: 485;
    -fx-translate-y: 30;
}

.table-view {
    -fx-min-width: 665;
    -fx-max-width: 665;
}

.table-column {
    -fx-padding: 0 0 0 10;
}

#vbox {
    -fx-padding: 70 0 0 10;
}

.button {
    -fx-translate-x: 575;
    -fx-translate-y: 25;
    -fx-background-color: rgb(31, 149, 206);
    -fx-text-fill: white;
}

.button:hover {
    -fx-background-color: rgba(31, 149, 206, 0.71);
}

#label-footer {
    -fx-font-size: 15px;
    -fx-translate-x: 210;
```

```
-fx-translate-y: 480;  
}
```

Antes de seguir para os próximos capítulos, você pode personalizar ainda mais a sua listagem. Aplique novos estilos, cores, fontes, efeitos. Use e abuse da documentação para absorver esse conteúdo de forma prática. Se quiser, você pode compartilhar comigo e com os demais leitores o resultado final do projeto com as suas melhorias.

CAPÍTULO 7

JAR, bibliotecas e build

Queremos o quanto antes lançar nossa aplicação, distribuir nas diversas livrarias e *stands* de venda. Por onde podemos começar?

Neste capítulo, veremos não só como compactar e distribuir nossa aplicação final, mas também como gerar e utilizar bibliotecas para linguagem Java. Além disso, veremos uma forma interessante de fazer o *build* e gerenciamento de dependências de nosso projeto, utilizando *Maven*, que é uma das ferramentas mais essenciais do mercado de hoje em dia.

7.1 JAR

Não queremos mandar todas as nossas classes e arquivos de configurações soltos para o cliente final, isso seria uma bagunça. Perder um dos arquivos implicaria no não funcionamento da aplicação, por exemplo. Uma forma bem

mais interessante seria compactar tudo em um único arquivo, um ZIP com tudo que o usuário precisa para executar nosso código.

Esse tipo de ZIP recebe na verdade a extensão JAR, de Java ARchive (arquivo Java). Nele estarão presentes nosso código-fonte compilado (bytecode), arquivos de configurações e dependências.

Como criar um JAR executável? Isso é bem simples, você inclusive pode fazer com qualquer compactador ZIP atual que conheça. Veremos uma forma bem prática de fazer pela própria IDE, em nosso caso o Eclipse.

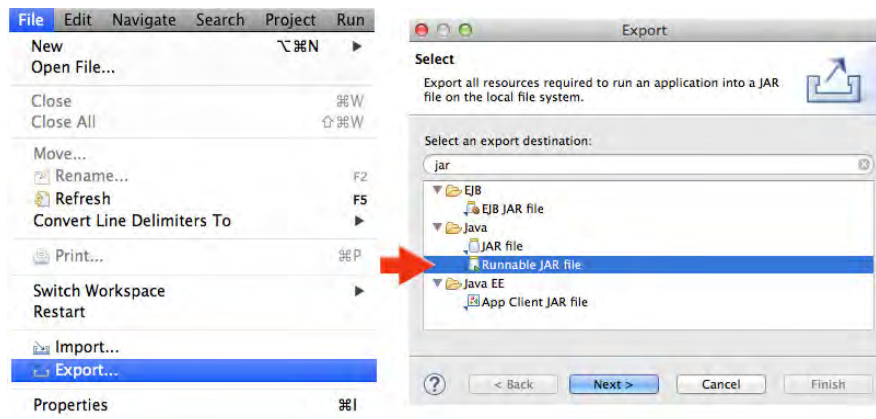
JAR NA LINHA DE COMANDO

Você também pode gerar um JAR diretamente pela linha de comando, utilizando o programa **jar** presente no JDK. Apesar de ser um pouco mais trabalhoso, você consegue gerar o seu executável com o comando:

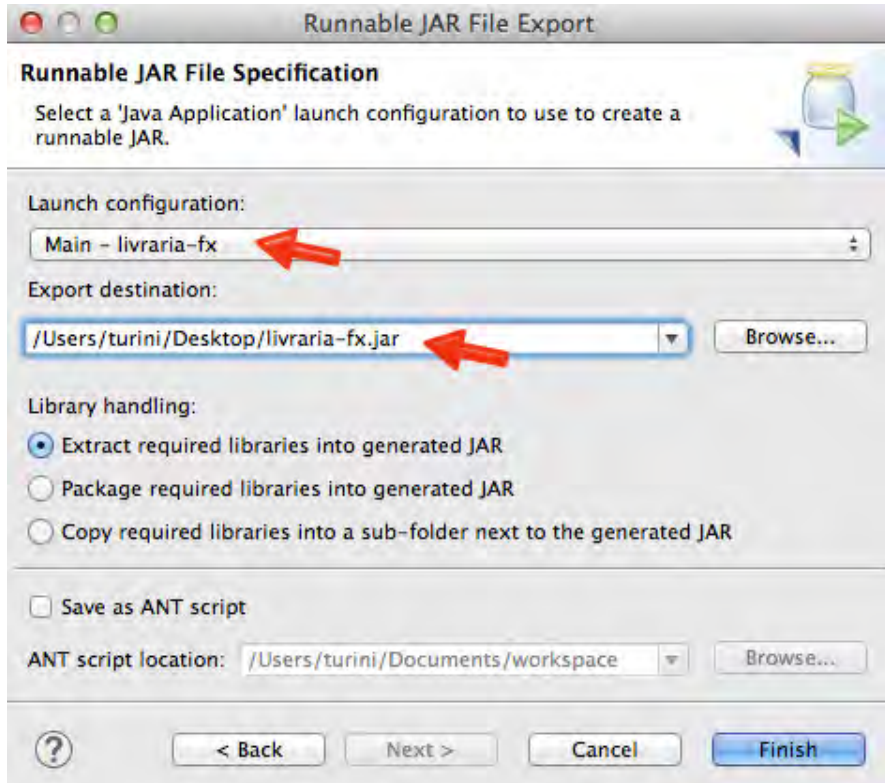
```
jar -cvf livraria.jar
  src/application/*.java
  src/db/*.java
  src/io/*.java
  src/repositorio/*.java;
```

7.2 GERANDO JAR EXECUTÁVEL PELA IDE

Fazer o processo pelo Eclipse, ou qualquer outra IDE moderna, é verdadeiramente mais simples. Para gerar o JAR executável de nossa aplicação Java FX, tudo o que precisamos fazer é clicar com o botão direito no projeto, selecionar **Export** e em seguida a opção **Runnable JAR File**.



Após selecioná-la, podemos escolher em *Launch configuration* qual a classe que deverá ser executada ao rodar esse JAR. Vamos escolher a classe `Main`. Além disso, no campo `Export destination`, precisamos preencher o local em que o arquivo executável Java deve ser criado, assim como seu nome e extensão (que será `.jar`):



Em meu caso, o caminho ficou `/Users/turini/Desktop/livreria-fx.jar`, em outras palavras, o arquivo se chamará `livreria-fx.jar` e será criado em minha área de trabalho.

Tudo pronto, basta clicar em `Finish`. Simples, não acha? Aproveite para gerar o JAR executável em sua casa, siga esses passos e confira se o arquivo `livreria-fx.jar` foi criado no caminho que você definiu.

7.3 EXECUTANDO A LIVRARIA-FX.JAR

Agora que temos o arquivo criado, podemos tentar executá-lo para conferir se tudo correu bem. Vamos fazer isso pela linha de comando, utilizando `java`

-jar SEU_EXECUTAVEL. Um exemplo seria:

```
java -jar Desktop/livraria-fx.jar
```

Excelente, a app está funcionando conforme esperado!



Note que, ao clicar em `Export CSV`, o arquivo `produtos.csv` com a relação de todos os livros será criado na área de trabalho, mesmo local onde se encontra o `JAR`.



PARA SABER MAIS: JAR LAUNCHER E OUTROS

Você pode utilizar programas como o *JAR Launcher* para rodar os seus executáveis Java com apenas um click, sem a necessidade de digitar o comando `java -jar` no terminal. No Mac OS, por exemplo, esse programa já vem instalado. Há diversas outras opções para os mais diferentes sistemas operacionais.

7.4 BIBLIOTECAS EM JAVA

Um `JAR` não precisa necessariamente ser um executável. É muito comum criarmos um `JAR` com um conjunto de classes que podem ser reutilizados em outros projetos. Em outras palavras, criar uma **biblioteca** Java.

Por sinal, o grande número de bibliotecas gratuitas em Java é um dos fatores que fazem a linguagem ser tão bem aceita no mercado. Precisa gerar

relatórios, manipular XMLs, conectar ao banco de dados, gerar códigos de barras etc.? Você já tem tudo isso pronto! O ecossistema do Java é enorme. E, o melhor, grande parte (talvez a maior parte) dos projetos (bibliotecas) em Java é gratuita e tem seu código aberto, os conhecidos *open sources*.

A Caelum, empresa onde trabalho, possui diversos projetos *open source*. VRaptor, Mamute e Stella são alguns deles. Contribuir com projetos de código aberto tem diversos benefícios, desde o conhecimento, que você com certeza vai adquirir e compartilhar, até o reconhecimento e divulgação de seu trabalho. Se você quiser, pode conhecer alguns deles no repositório público da Caelum:

<http://github.com/caelum/>

7.5 DOCUMENTANDO SEU PROJETO COM JAVADOC

É sempre muito interessante documentar as suas classes quando você cria um novo JAR. Afinal, como a pessoa que vai usá-lo saberá quais classes e métodos estão disponíveis? Outras informações como exceções que o código pode lançar e detalhes importantes de seu uso também são diferenciais significantes.

O próprio código do Java é todo documentado, você pode ver a documentação completa de sua API pelo site da Oracle:

<http://download.java.net/jdk8/docs/api/index.html>

The screenshot shows the official Java API documentation for the Standard Edition 8. The navigation bar at the top includes links for Overview, Package, Class, Use, Tree, Deprecated, Index, and Help. The left sidebar provides quick access to all classes and profiles. The main content area is titled 'Java™ Platform, Standard Edition 8 API Specification' and includes a description of the document. Below this, there are sections for 'Profiles' (listing compact1, compact2, and compact3) and a 'Packages' table. The table lists the following packages and their descriptions:

Package	Description
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet container.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interfaces.

Você pode (e deve) usar o `javadoc` para criar uma documentação equivalente para os seus projetos. Esse programa também já vem na JDK e pode ser usado diretamente pela linha de comando, ou com ajuda da IDE. Se ainda não conhece o `javadoc`, pode gostar de ler mais a respeito em:

<http://www.oracle.com/technetwork/articles/java/index-137868.html>

Um exemplo prático de documentação em Java

É muito simples documentar seus projetos em Java, o `javadoc` cuida de boa parte do trabalho pesado. Mas, claro, é sempre interessante oferecer algumas informações para enriquecer a documentação de nosso projeto. Fazemos isso no próprio código:


```
/**
 * Cuida do acesso aos dados da classe {@link Produto}.
 *
 * @author Rodrigo Turini <rodrigo.turini@caelum.com.br>
 * @since 1.0
 */
public class ProdutoDAO {

    /**
     * Lista todos os produtos do banco de dados
     *
     * @return {@link ObservableList} com todos os produtos
     * @throws RuntimeException em caso de erros
     */
    public ObservableList<Produto> lista() {
        // código omitido
    }

    /**
     * @param produto que deverá ser adicionado no banco.
     * @throws RuntimeException em caso de erros
     */
    public void adiciona(Produto produto) {
        // código omitido
    }
}
```

Observe que isso é feito com um tipo de bloco de comentário com significado especial. Ele sempre começa com `/**` e termina com `*/`, nas outras linhas, colocamos apenas um `*`.

Você pode adicionar uma descrição para as classes e métodos, mas prefira nada muito extenso e que conte detalhes de implementação. Por exemplo, imagine que no `javadoc` eu diga que estamos persistindo em um banco de dados MySQL. O que acontecerá se eu migrar de banco? A documentação ficará desatualizada ou eu precisarei lembrar (o que provavelmente não vai acontecer).

Também podemos definir outras informações na documentação, como fizemos com o autor, desde quando a classe existe (*since*), parâmetros, retorno,

exceptions etc. Tudo isso é feito de forma simples com as próprias anotações de marcação do `javadoc`.

Depois de devidamente documentado, pelo Eclipse, podemos ir em **Project** e depois escolher a opção **Generate Javadoc**.

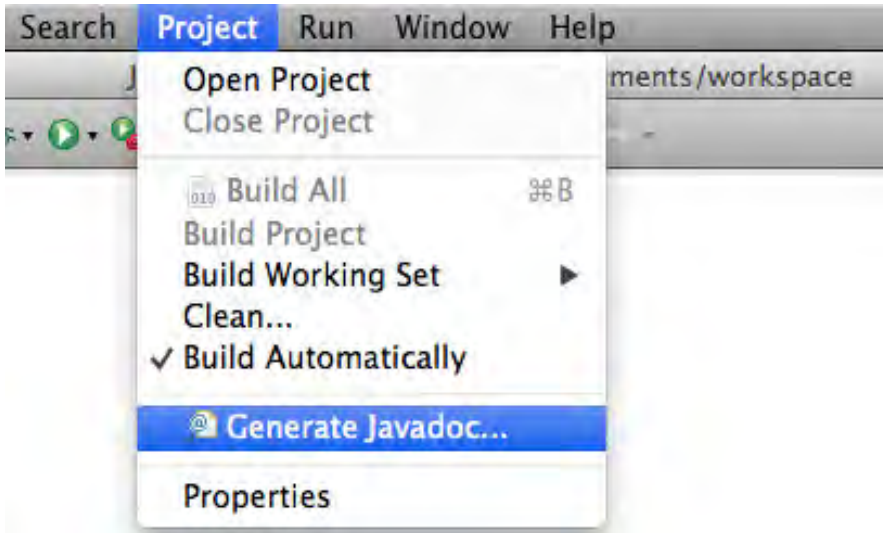


Fig. 7.6: Gerando javadoc pela interface do Eclipse.

Uma nova janela aparecerá. Tudo o que precisamos fazer é selecionar o projeto que queremos documentar e o diretório onde a documentação deverá ser gerada. Vamos colocar dentro de uma pasta chamada `doc` dentro do próprio projeto.

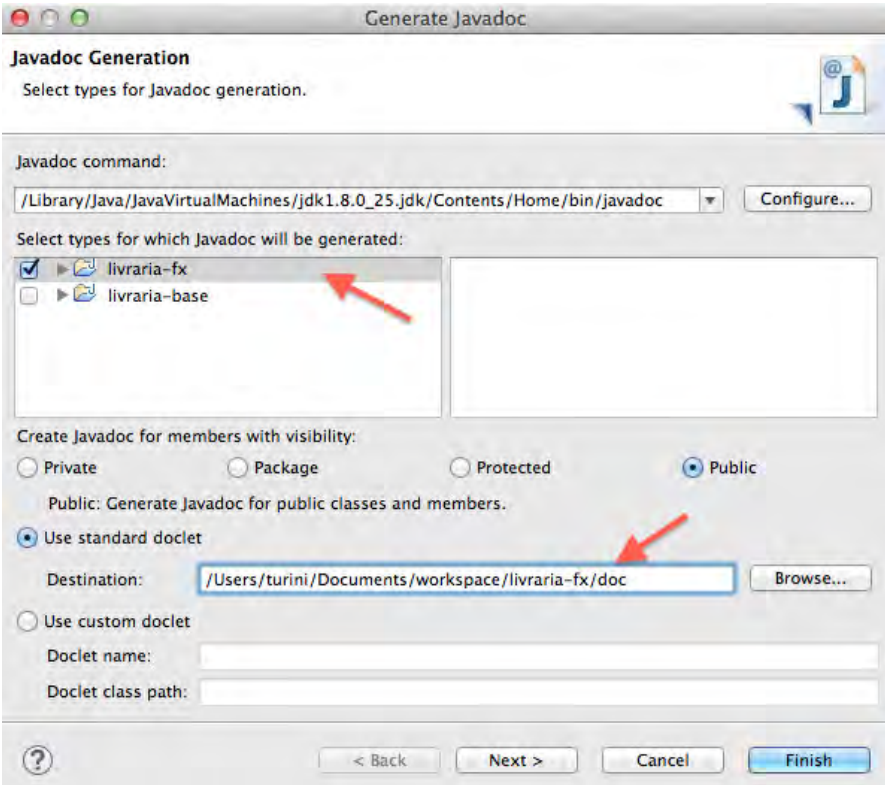


Fig. 7.7: Configurando o projeto e destino do javadoc.

Agora só precisamos clicar em `Finish` e pronto, confira que a pasta `doc` foi criada:

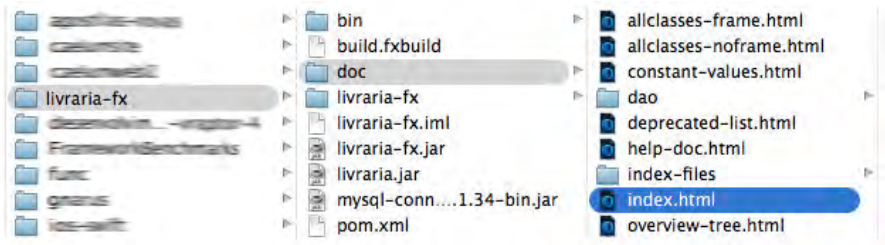


Fig. 7.8: Arquivos criados pelo javadoc na pasta doc.

Abra o arquivo `index.html` em seu navegador para ver o resultado, você vai gostar.

7.6 AUTOMATIZANDO BUILD COM MAVEN

O Eclipse facilitou bastante o processo de gerar o `JAR` executável, mas conforme a aplicação vai evoluindo o processo de *build* pode se tornar mais complexo. É necessário compilar o projeto, copiar suas dependências, configurações etc. Esse processo tem que ser repetido sempre.

Fazer isso manualmente, ainda que com ajuda da IDE, pode ser tanto trabalhoso quanto problemático. A cada execução, o humano que está fazendo a tarefa pode se esquecer de algum dos passos, cometer algum erro etc.

Qual a solução? Automatizar todo esse processo. Existem diversas ferramentas de *build* que nos auxiliam nesse trabalho, o *Maven* é uma das mais populares. Com ele e seus diversos plugins, o processo de *build* e gerenciamento de dependências de nossas aplicações ficam bem simples e sofisticados.

DOWNLOAD E INSTALAÇÃO DO MAVEN

Para saber se você tem o Maven instalado em seu sistema operacional, basta abrir o terminal e digitar o comando `mvn -version`. A saída deverá ser parecida com:

```
Rodrigos-MacBook-Pro:~ turini$ mvn -version
Apache Maven 3.2.3 (...; 2014-08-11T17:58:10-03:00)
Maven home: /Applications/apache-maven-3.2.3
Java version: 1.8.0_25, vendor: Oracle Corporation
Java home: /Library/Java/.../jdk1.8.0_25.jdk/Contents/Home/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.9.4", arch: "x86_64"
```

Se você ainda não tem o Maven instalado, pode fazer seu download direto pelo seu site:

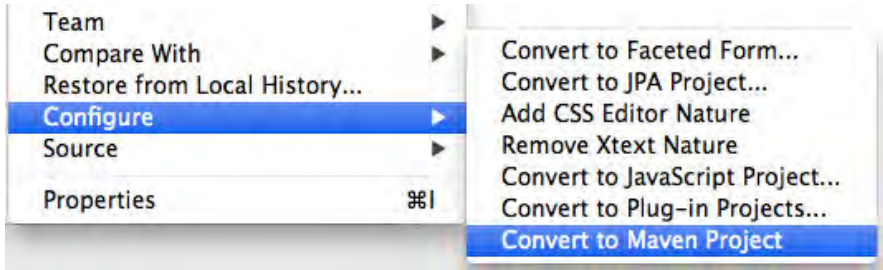
<http://maven.apache.org/download.cgi>

Lá você também encontra instruções de instalação para cada SO. Se encontrar qualquer dificuldade para fazer isso, não deixe de postar no GUY ou nos mandar um e-mail.

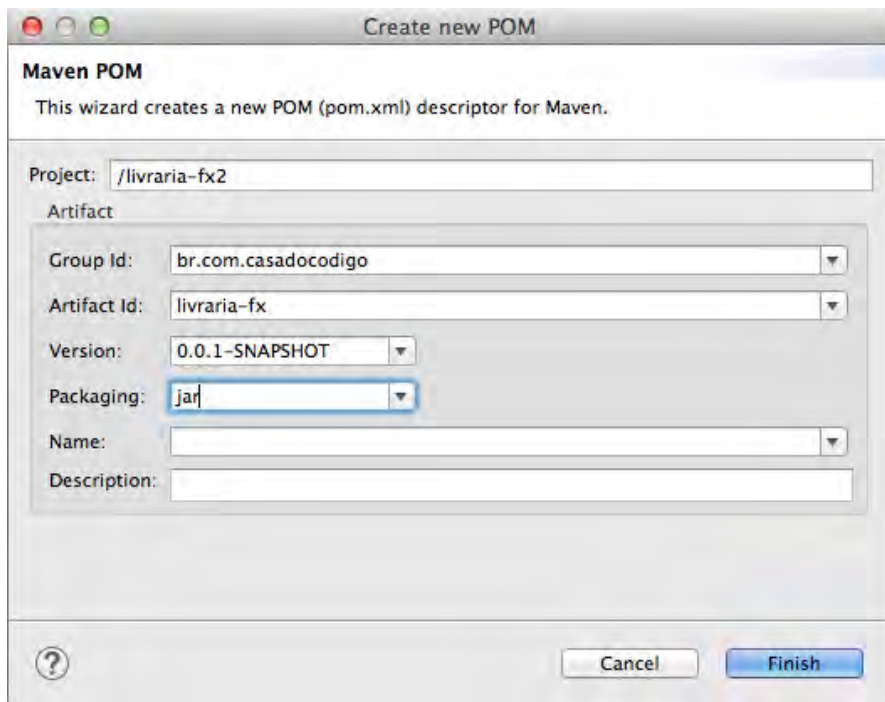
7.7 TRANSFORMANDO NOSSA APP EM UM PROJETO MAVEN

Agora que já temos tudo baixado e configurado, podemos converter nossa *app* para um projeto Maven. O Eclipse também simplifica muito esse trabalho, tudo o que precisamos fazer é clicar com o botão direito no projeto e selecionar:

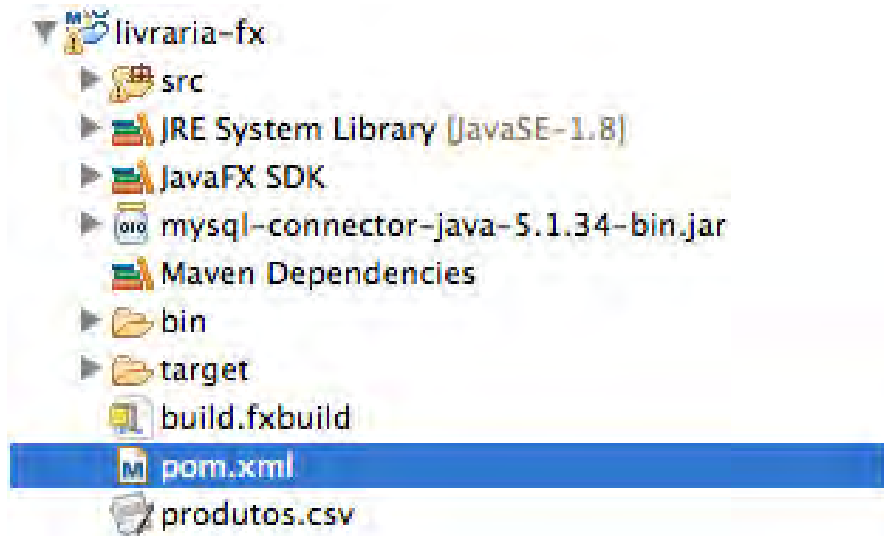
- Configure
- Convert to Maven Project



A janela *Create new POM* deve aparecer. Nela, vamos preencher o campo `Group Id` com `br.com.casadocodigo`, que é o pacote padrão do projeto, o `Artifact Id` com o nome `livraria-fx` e manter as outras opções com o valor padrão.



Clicamos em `Finish` e pronto, um arquivo chamado `pom.xml` foi criado em nosso projeto.



Esse arquivo possui o seguinte conteúdo:

```
<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.casadocodigo</groupId>
  <artifactId>livraria-fx</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <build>
    <sourceDirectory>src</sourceDirectory>

    <resources>
      <resource>
        <directory>src</directory>
        <excludes>
          <exclude>**/*.java</exclude>
        </excludes>
      </resource>
    </resources>
  </build>
</project>
```

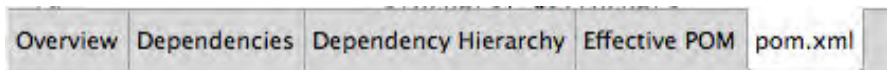


```
        </excludes>
      </resource>
    </resources>

    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>

  </build>
</project>
```

Ao abrir o arquivo pela primeira vez, o Eclipse provavelmente mostrará apenas a página de *Overview*. Para ver o conteúdo deste XML, você pode clicar na aba `pom.xml` que fica visível no rodapé do editor:



Não se preocupe em entender cada detalhe deste XML de configuração agora, mas note que ele já tem algumas informações essenciais de nosso *build*. Nossa app agora já é um projeto Maven. Bem simples, não acha?

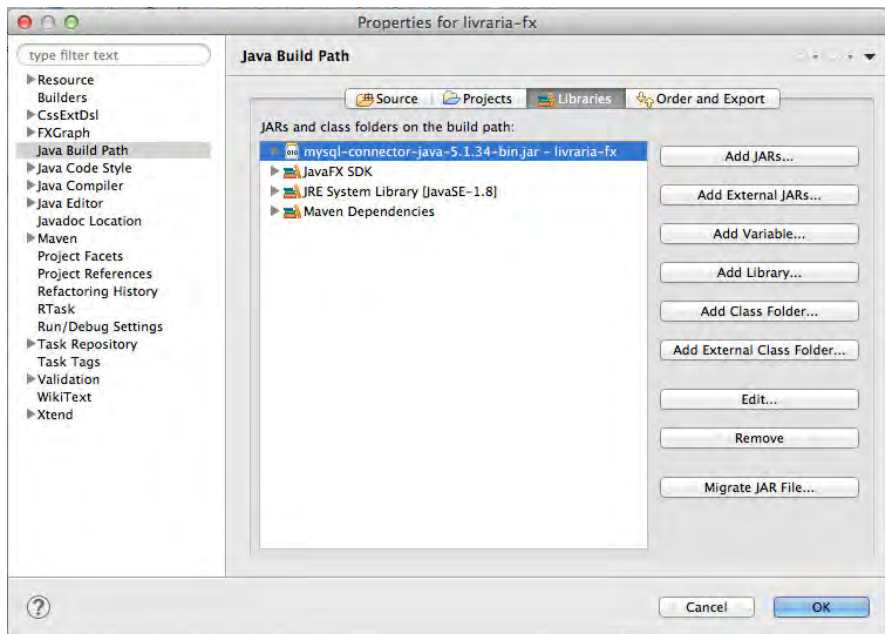
7.8 ADICIONANDO AS DEPENDÊNCIAS COM MAVEN

Agora que já estamos trabalhando com Maven, podemos deixar o controle das bibliotecas que temos em nosso projeto com ele. Ele será o responsável pelo gerenciamento de nossas dependências.

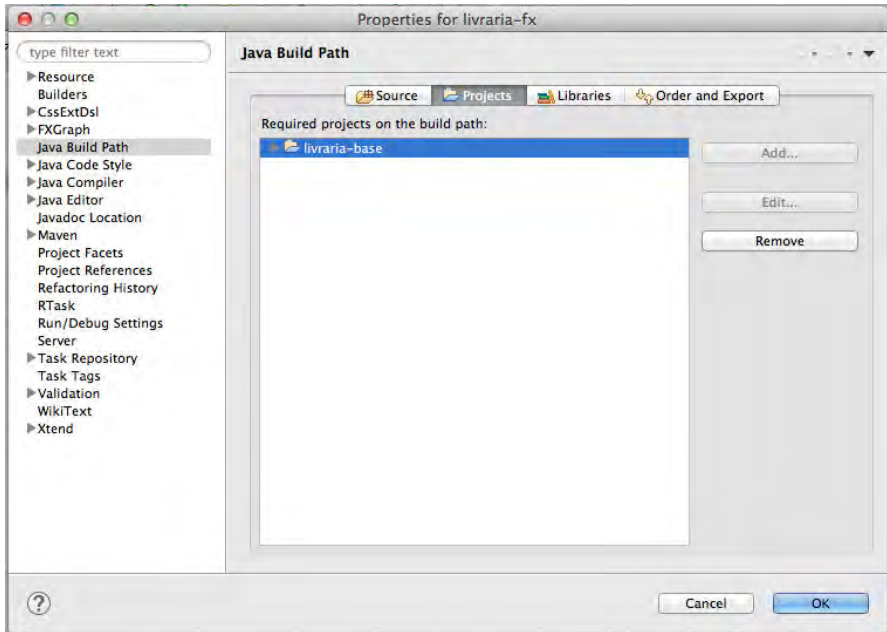
Atualmente nosso projeto tem apenas duas: o driver de conexão com MySQL e a base do projeto da livreria, onde se encontra a interface `Produto` e suas implementações.

Para começar, vamos remover esses dois elementos de nosso *classpath*. Faremos isso clicando com o botão direito no projeto, opção `Build Path` e `Configure Build Path`.

Na aba `Library`, clicamos no `mysql-connector-java.jar` e em `Remove`.



Em seguida, faremos o mesmo com o projeto `livreria-base`, na aba `Project`:



Ao clicar em **Ok**, o projeto já deve parar de compilar, afinal essas bibliotecas são necessárias. Vamos agora abrir o arquivo `pom.xml` e declarar essas duas dependências lá:

```
<!-- inicio do arquivo omitido -->
<dependencies>
  <dependency>
    <groupId>br.com.caelum.livraria</groupId>
    <artifactId>livraria-base</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.34</version>
  </dependency>
</dependencies>
```

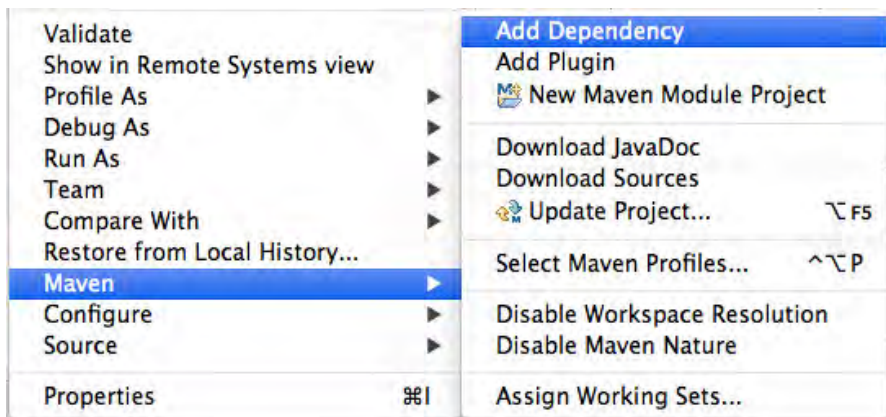
```
</project>
```

Note que para fazer isso só precisamos adicionar uma *tag* chamada *dependencies*, e para cada dependência uma nova *tag* *dependency* com a seguinte estrutura:

```
<dependency>
  <groupId> VALOR_AQUI </groupId>
  <artifactId> VALOR_AQUI </artifactId>
  <version> VALOR_AQUI </version>
</dependency>
```

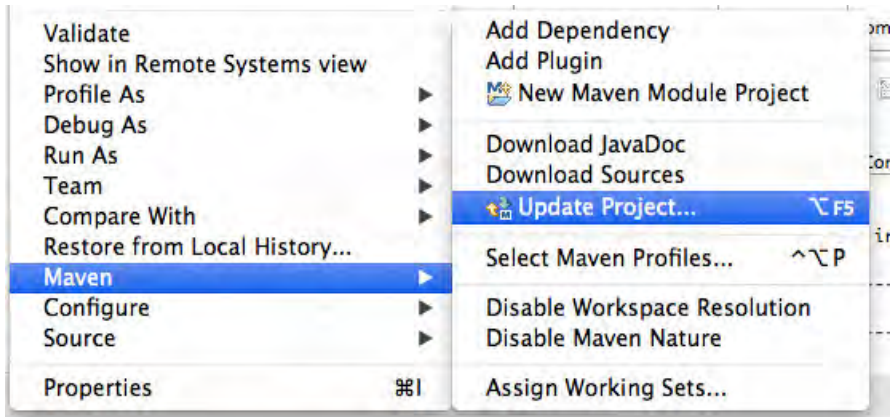
ADICIONANDO PELA INTERFACE DO ECLIPSE

Se preferir, você também pode adicionar novas dependências em seu projeto diretamente pela interface do Eclipse. Basta clicar em **Maven** e **Add Dependency**.

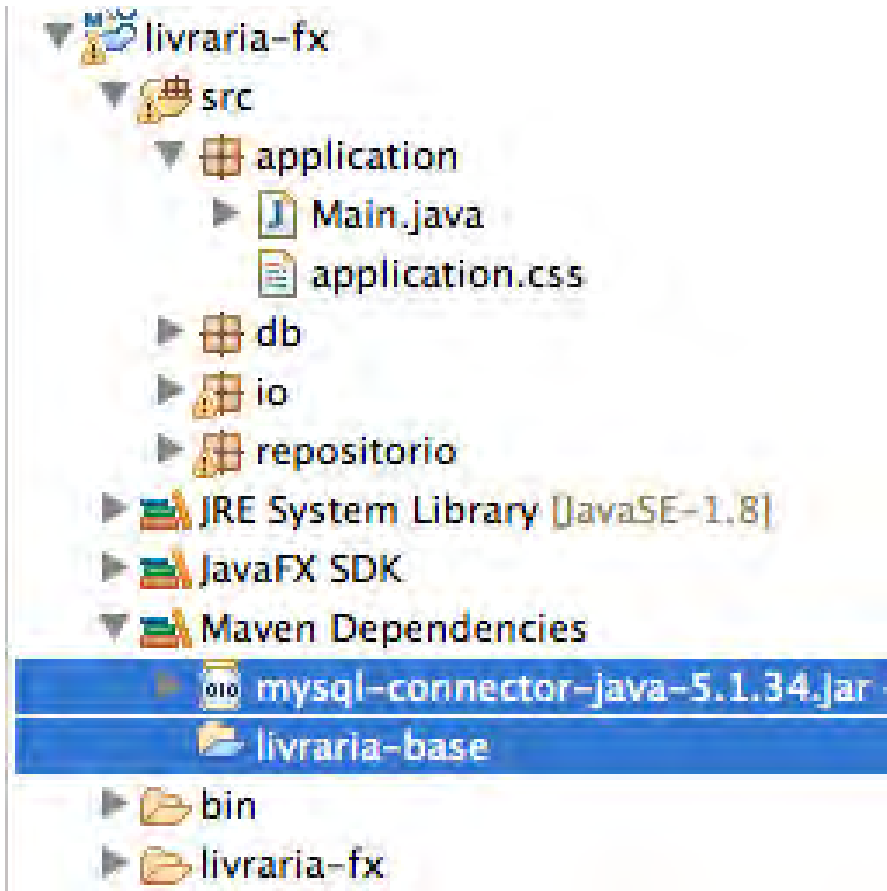


Dessa forma, você não precisa escrever nenhuma linha de XML, ele fará esse trabalho para você.

Vamos ver se tudo funcionou? Basta clicar com o botão direito no projeto, Maven e escolher `Update Project...` (ou pressionar o atalho `Alt + F5`).



Feito isso, a ferramenta fará seu trabalho e o projeto deve voltar a compilar. Note que as dependências ficam visíveis em `Maven Dependencies`, direto pelo `package explorer`:



Buscando por dependências no Maven

Assim como o `mysql-connector-java`, você pode encontrar qualquer outra dependência necessária pra seus projetos no site:

<http://search.maven.org/>

The Central Repository

mysql-connector-java

SEARCH | ADVANCED SEARCH | BROWSE | QUICK STATS

SEARCH

New: About Central | Advanced Search | API Guide | Help

Search Results

< 1 > displaying 1 to 3 of 3

GroupId	ArtifactId	Latest Version	Updated	Download
mysql	mysql-connector-java	5.1.34 all (48)	17-Oct-2014	pom jar bundle jar
org.wildon-framework	mysql-connector-java	5.1.11_1	06-Apr-2014	pom jar javadoc.jar sources.jar
org.ops4j.pax.tip	org.ops4j.pax.tip.mysql.connector.java	5.1.22.1	30-Dec-2012	pom jar

< 1 > displaying 1 to 3 of 3

Basta clicar na versão que quer utilizar e copiar a *tag* da dependência para dentro do arquivo `pom.xml` de seu projeto.

The Central Repository

mysql-connector-java

SEARCH | ADVANCED SEARCH | BROWSE | QUICK STATS

SEARCH

New: About Central | Advanced Search | API Guide | Help

Artifact Details For [mysql](#) : [mysql-connector-java](#) : [5.1.34](#)

Click on a link above to browse the repository.

Project Information

GroupId: [mysql](#)

ArtifactId: [mysql-connector-java](#)

Version: [5.1.34](#)

Dependency Information

Apache Maven

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifa
  <version>5.1.34</version>
</dependency>
```

Apache Buildr

Apache Ivy

Groovy Grape

Gradle/Grails

Scala SBT

Leiningen

Project Object Model (POM)

```
<project>

  <modelVersion>4.0.0</modelVersion>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.34</version>
  <packaging>jar</packaging>

  <name>MySQL Java Connector</name>
  <description>MySQL Java Connector</description>

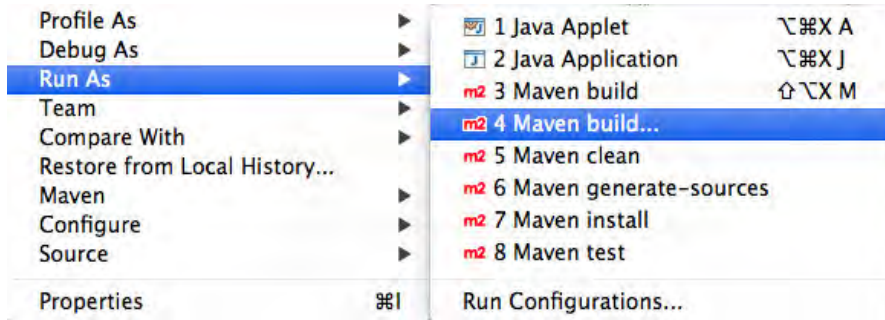
  <licenses>
    <license>
      <name>The GNU General Public License, Version 2</name>
      <url>http://www.gnu.org/licenses/gpl.txt</url>
      <distribution>repo</distribution>
      <comments>MySQL Connector/J contains exceptions to GPL requirements when it
that are licensed under OSI-approved open source licenses, see EXCEPTIONS-
in this distribution for more details.</comments>
    </license>
  </licenses>

  <url>http://dev.mysql.com/usingmysql/java/</url>

  <scm>
    <connection>
```

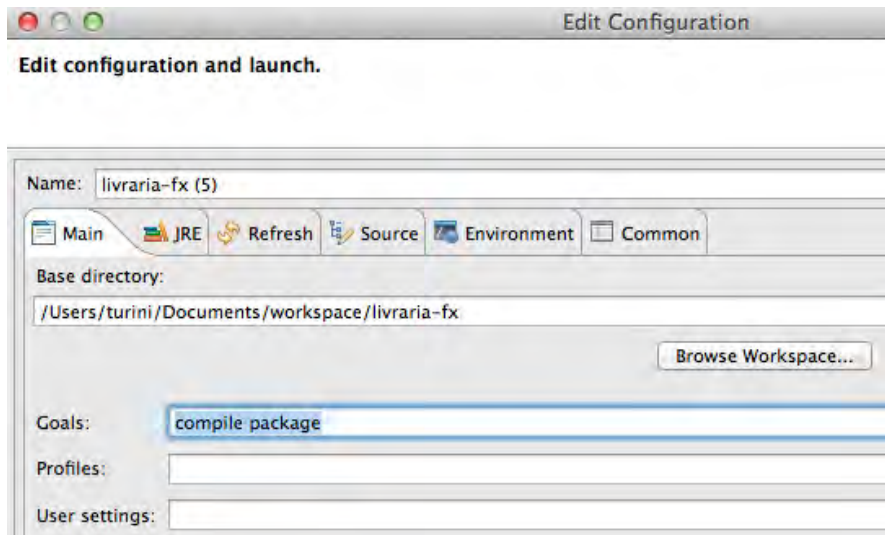
7.9 EXECUTANDO ALGUMAS TASKS DO MAVEN

Vamos agora ver o build do Maven em prática? Na opção `Run As...`, clicando com o botão direito no projeto, escolha `Maven Build`.



Adicione no campo `Goals` as opções:

- `compile package`



Essas são algumas das *tasks* padrões do Maven. Como o próprio nome já diz, estamos dizendo que queremos compilar e empacotar, gerar um `JAR` do projeto. Se você quiser, pode ler mais sobre esses *Goals* em:

<http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

Ao executar, a saída será parecida com:

```
[INFO] Scanning for projects...
[INFO]
...
[INFO] -----
[INFO] Building livraria-fx 0.0.1-SNAPSHOT
[INFO] -----
[INFO]
...
[INFO] Building jar: /Users/turini/Documents/workspace
    /livraria-fx/target/livraria-fx-0.0.1-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 7.129 s
[INFO] Finished at: 2014-11-22T20:14:47-03:00
[INFO] Final Memory: 8M/94M
[INFO] -----
```

Sucesso! O build foi concluído sem nenhuma falha. Note que o `JAR` foi criado dentro de uma pasta chamada `target`, esse é o destino padrão da ferramenta.

Mas esse `JAR` não é um executável! Precisamos deixar claro ao Maven que esse é um projeto Java FX. Como fazer isso?

7.10 ADICIONANDO PLUGIN DO JAVA FX

Como um extra, ensinaremos ao Maven como gerar um `JAR` executável do Java FX. Só de pensar parece trabalhoso, não é? Mas na verdade isso tudo já está pronto, existem diversos plugins que nos auxiliam nesse trabalho. Uma opção bastante utilizada é o `javafx-maven-plugin`.

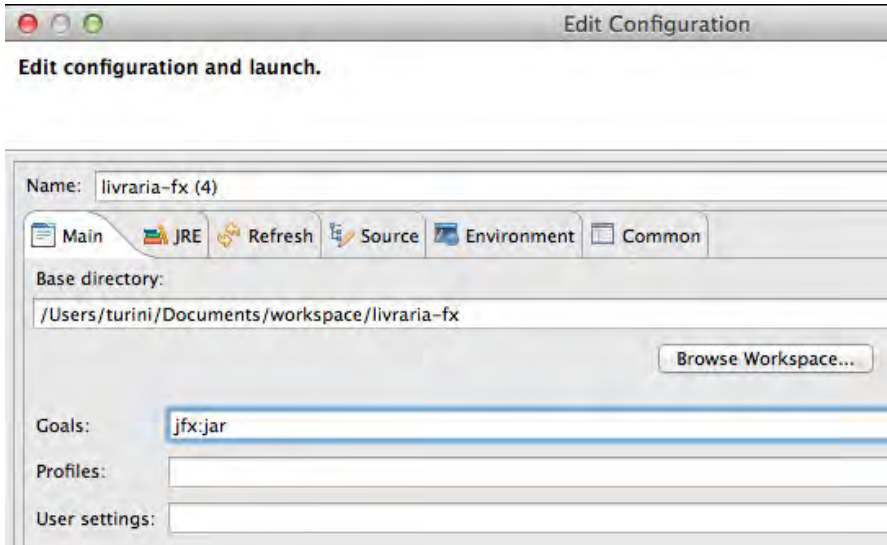
Da mesma forma como fizemos para adicionar uma dependência, podemos abrir o arquivo `pom.xml` e adicionar as seguintes linhas do plugin, logo após o `maven-compiler-plugin` que já está declarado:

```
<plugins>
  <!-- maven-compiler-plugin omitido -->
  <plugin>
    <groupId>com.zenjava</groupId>
    <artifactId>javafx-maven-plugin</artifactId>
    <version>8.1.2</version>
    <configuration>
      <mainClass>application.Main</mainClass>
    </configuration>
  </plugin>
</plugins>
```

Observe que estamos definindo que a classe `application.Main` é a classe principal de nossa aplicação. Além disso, precisamos adicionar a *tag* `organization` como a seguir:

```
<organization>
  <name>Casa do Código</name>
</organization>
```

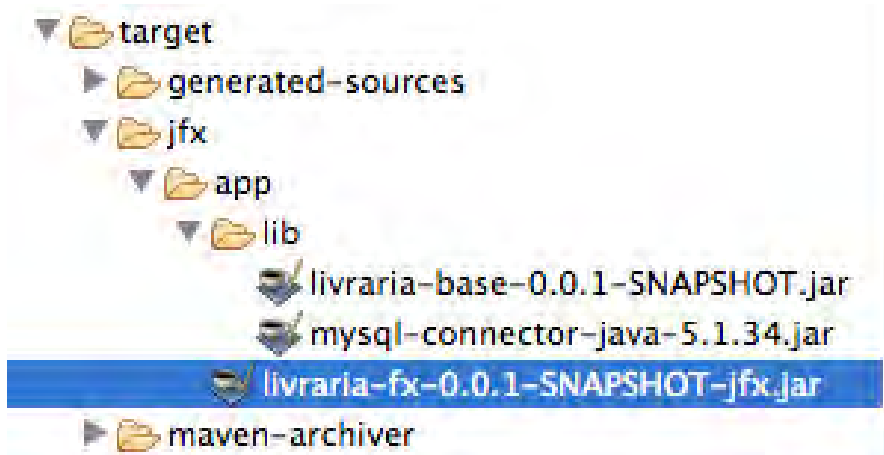
Tudo pronto! Já podemos usar uma de suas *tasks*. Para gerar um JAR executável, por exemplo, você pode clicar em `Run As...`, `Maven Build` e em `Goals` adicionar o valor `jfx:jar`.



Ao clicar em `Run`, o resultado será parecido com:

```
[INFO] Scanning for projects...
...
[INFO] Building Java FX JAR for application
[INFO] Adding 'deploy' directory to Mojo
        classpath: /Users/turini/Documents
                /workspace/livraria-fx/src/main/deploy
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 6.793 s
[INFO] Finished at: 2014-11-22T20:36:10-03:00
[INFO] Final Memory: 9M/93M
[INFO] -----
```

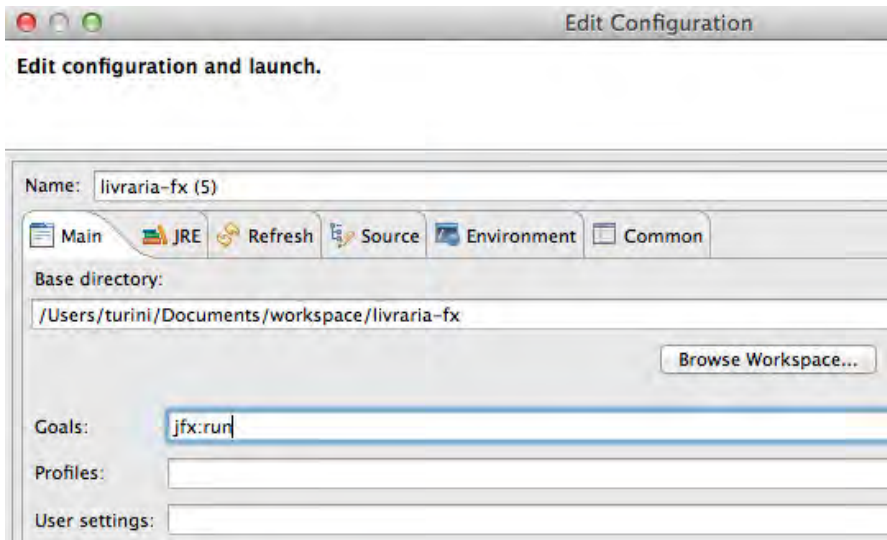
Note que o arquivo `JAR` foi gerado dentro da pasta `target`, no subdiretório `jfx/app`:



Se quiser, teste para verificar que ele foi criado corretamente. Para isso, basta digitar o comando:

```
java -jar target/jfx/app/livraria-fx-0.0.1-SNAPSHOT-jfx.jar
```

Perfeito, tudo está funcionando. Mas isso foi um pouco trabalhoso, não acha? Para nossa sorte, o plugin de Java FX já tem uma *task* que cria e executa o JAR de nossa aplicação automaticamente! Para utilizá-la, clique novamente em **Run As...**, **Maven Build...**. No campo **Goals** adicione `jfx:run`:



Agora basta clicar em `Run` e a app será executada a partir do `JAR` criado pelo plugin.

CONHECENDO MAIS ALGUNS PLUGINS

Além do plugin de Java FX que utilizamos, existem mais centenas de plugins que facilitam bastante o nosso trabalho. Você pode ver uma lista com alguns dos plugins existentes em:

<http://maven.apache.org/plugins/index.html>

7.11 MAVEN NA LINHA DE COMANDO

Vimos que é possível executar as metas do Maven diretamente pelo Eclipse, mas comumente esse trabalho é feito diretamente pela linha de comando. Se você quiser testar, basta abrir a pasta do seu projeto pelo terminal e executar a instrução `mvn` com as metas que você quiser. Por exemplo:

```
mvn jfx:run
```

Note que o resultado será o mesmo, um `JAR` executável de nossa app será criado e executado. Você pode executar uma série de comandos de uma só vez, basta separá-los por um espaço, como a seguir:

```
mvn clean jfx:run
```

A primeira meta executada, o `clean`, é utilizada para limpar os arquivos e diretórios gerados pelo Maven na pasta `target`. Esse comando, portanto, limpa a pasta `target` e logo em seguida executa o `jfx:run` responsável por gerar e executar o `JAR` do projeto.

Algumas das metas mais comuns do Maven são:

- `compile` compila o código-fonte do projeto
- `test` compila e executa seus testes de unidade existentes no projeto
- `package` empacota (gera o `JAR`, por exemplo) do código compilado de seu projeto
- `validate` valida se o projeto tem todas as informações necessárias para seu funcionamento
- `install` instala localmente seu projeto

Se você estiver interessado, pode conhecer um pouco mais sobre as possibilidades do Maven em:

<http://maven.apache.org/general.html>

OUTRA OPÇÃO: ANT E IVY

Outra opção bastante conhecida e utilizada no mercado é o *Ant*, também da *Apache*. Ele cuida apenas do processo de build. Para o gerenciamento de dependências você pode contar com outro framework bastante conhecido, o *Ivy*. Esses dois trabalham bem juntos e têm suas vantagens e desvantagens, assim como qualquer outra ferramenta. Há quem prefira usá-los no lugar do Maven. Interessado em saber mais sobre *Ant* e *Ivy*? A documentação deles pode ajudar bastante, elas estão disponíveis nos links:

- <http://ant.apache.org/manual/index.html>
- <http://ant.apache.org/ivy/>

7.12 COMO FICOU NOSSO POM.XML

Após as alterações, nosso arquivo `pom.xml` ficou 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>livraria-fx</groupId>
  <artifactId>livraria-fx</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <organization>
    <name>Casa do Código</name>
  </organization>

  <build>
    <sourceDirectory>src</sourceDirectory>
```

```
<resources>
  <resource>
    <directory>src</directory>
    <excludes>
      <exclude>**/*.java</exclude>
    </excludes>
  </resource>
</resources>
<plugins>
  <plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
      <source>1.8</source>
      <target>1.8</target>
    </configuration>
  </plugin>
  <plugin>
    <groupId>com.zenjava</groupId>
    <artifactId>javafx-maven-plugin</artifactId>
    <version>8.1.2</version>
    <configuration>
      <mainClass>application.Main</mainClass>
    </configuration>
  </plugin>
</plugins>
</build>
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-plugin-plugin</artifactId>
      <version>3.2</version>
    </plugin>
  </plugins>
</reporting>
<dependencies>
  <dependency>
    <groupId>br.com.caelum.livraria</groupId>
```



```
        <artifactId>livraria-base</artifactId>
        <version>1.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.34</version>
    </dependency>
</dependencies>
</project>
```

É natural não reagir tão bem ao ver um arquivo `xml` com tantas informações, mas após usar a ferramenta por alguns instantes você logo deve perceber que não é nada de tão complicado. Atualizar suas dependências nunca foi tão fácil, além de que podemos fazer o *build* completo de nossa aplicação com apenas um comando.

CAPÍTULO 8

Refatorações

“Refatoração é uma técnica controlada para reestruturar um trecho de código existente, alterando sua estrutura interna sem modificar seu comportamento externo. Consiste em uma série de pequenas transformações que preservam o comportamento inicial. Cada transformação (chamada de refatoração) reflete em uma pequena mudança, mas uma sequência de transformações pode produzir uma significativa reestruturação. Como cada refatoração é pequena, é menos provável que se introduza um erro. Além disso, o sistema continua em pleno funcionamento depois de cada pequena refatoração, reduzindo as chances de o sistema ser seriamente danificado durante a reestruturação.

– **Martin Fowler**

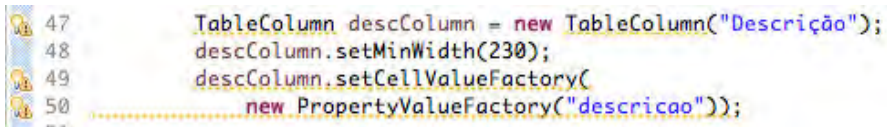
Essa ideia de utilizar pequenas transformações para melhorar a qualidade do nosso código recebe o nome de *baby steps* (passos de bebê). Essa é uma prática fundamental, utilizada para melhorar a legibilidade, clareza e design

de nosso código. Sempre que for fazer uma refatoração, evite mudar tudo de uma só vez. Caso algum erro apareça, será muito mais difícil identificar qual das mudanças foi a responsável pelo erro (ou *bug*, como normalmente chamamos).

Neste capítulo, vamos fazer pequenas refatorações em nosso código, buscando melhorar a legibilidade, evitar repetições e, com isso, aumentar a qualidade e manutenibilidade de nosso projeto. Além disso, relembremos algumas das boas práticas e *design patterns* que vimos ao decorrer do livro. Pronto para começar?

8.1 REFATORAÇÃO

Logo que começamos a trabalhar com a tabela de `Produtos`, você deve ter percebido que o Eclipse mostrou vários *warnings*. Inclusive, usamos a anotação `@SuppressWarnings` para silenciar esses alertas, mas por que eles estão acontecendo?



```
47     TableColumn descColumn = new TableColumn("Descrição");
48     descColumn.setMinWidth(230);
49     descColumn.setCellValueFactory(
50         new PropertyValueFactory("descricao"));
```

Olhando com cuidado para a mensagem, temos a informação:

```
TableColumn is a raw type. References to generic
type TableColumn<S,T> should be parameterized
```

Em outras palavras, não estamos informando os parâmetros genéricos da classe `TableColumn`. Idealmente, no lugar de fazer:

```
TableColumn nomeColumn = new TableColumn("Nome");

nomeColumn.setMinWidth(180);

nomeColumn.setCellValueFactory(
    new PropertyValueFactory("nome"));
```

Seria necessário fazer assim:

```
TableColumn<Produto, String> nomeColumn =  
    new TableColumn<Produto, String>("Nome");  
  
nomeColumn.setMinWidth(180);  
  
nomeColumn.setCellValueFactory(  
    new PropertyValueFactory<Produto, String>("nome"));
```

Veja que agora estamos definindo a classe `TableColumn` passando o `Produto`, que é o tipo do objeto que está sendo manipulado, mais a `String`, que é o tipo do conteúdo no campo da tabela. Mas o código ficou muito mais verboso, não acha? Por isso, desde o início optamos por não passar o parâmetro genérico, apesar das centenas de *warnings*.

Outro problema nessa parte do código está na quantidade de repetições. Observe que grande parte do código está idêntico, para cada coluna:

```
TableView<Produto> tableView = new TableView<>(produtos);  
  
TableColumn nomeColumn = new TableColumn("Nome");  
nomeColumn.setMinWidth(180);  
nomeColumn.setCellValueFactory(  
    new PropertyValueFactory("nome"));  
  
TableColumn descColumn = new TableColumn("Descrição");  
descColumn.setMinWidth(230);  
descColumn.setCellValueFactory(  
    new PropertyValueFactory("descricao"));  
  
TableColumn valorColumn = new TableColumn("Valor");  
valorColumn.setMinWidth(60);  
valorColumn.setCellValueFactory(  
    new PropertyValueFactory("valor"));  
  
TableColumn isbnColumn = new TableColumn("ISBN");  
isbnColumn.setMinWidth(180);  
isbnColumn.setCellValueFactory(  
    new PropertyValueFactory("isbn"));
```

```
tableView.getColumns().addAll(nomeColumn, descColumn,  
    valorColumn, isbnColumn);
```

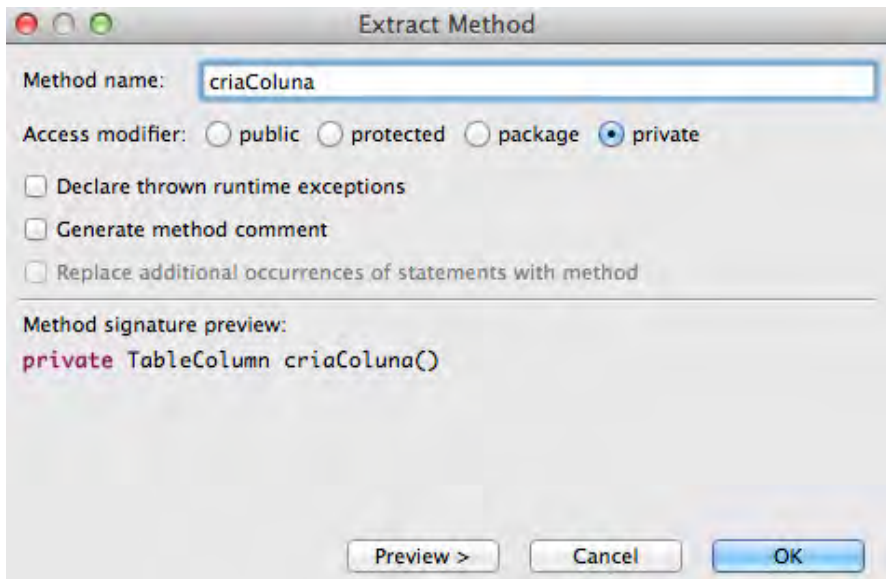
Seguindo as boas práticas da orientação a objetos, devemos isolar comportamentos repetidos em métodos. O ganho vai desde a reutilização, já que ele poderá ser chamado várias vezes em nosso código, até em legibilidade, já que podemos lhe dar um nome bastante significativo.

Vamos fazer essa pequena refatoração em nosso código. O método poderá ficar assim:

```
private TableColumn criaColuna(String titulo,  
    int largura, String atributo) {  
  
    TableColumn column = new TableColumn(titulo);  
    column.setMinWidth(largura);  
    column.setCellValueFactory(  
        new PropertyValueFactory(atributo));  
  
    return column;  
}
```

USANDO EXTRACT METHOD DO ECLIPSE

Experimente selecionar o trecho de código que você quer extrair e pressionar as teclas de atalho `Control + Shift + M`.



Basta digitar o nome do método que você quer criar e pronto!

Apesar de simples, essa pequena mudança já dá uma cara bem melhor para nosso código. Ele ficou assim:

```
TableView<Produto> tableView = new TableView<>(produtos);

TableColumn nomeColumn = criaColuna("Nome", 180, "nome");
TableColumn descColumn =
    criaColuna("Descrição", 230, "descricao");
TableColumn valorColumn = criaColuna("Valor", 60, "valor");
TableColumn isbnColumn = criaColuna("ISBN", 180, "isbn");
```

```
tableView.getColumns().addAll(nomeColumn, descColumn,  
    valorColumn, isbnColumn);
```

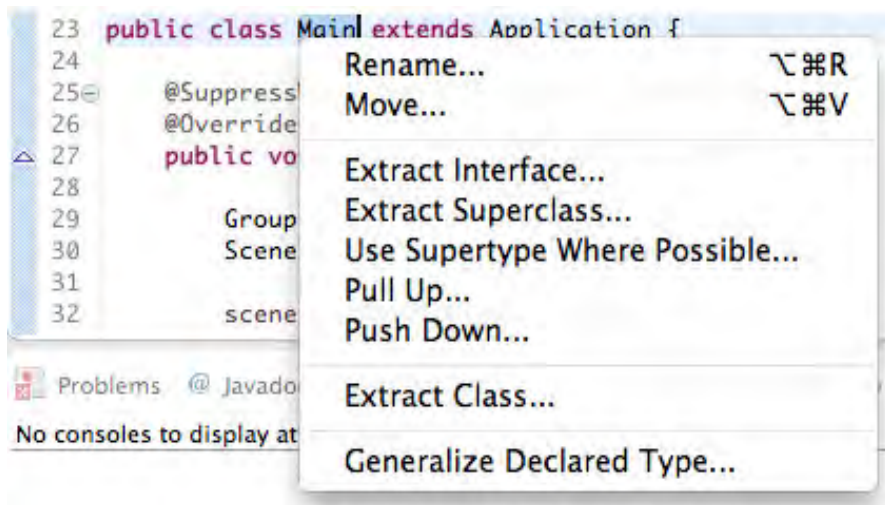
Além disso, agora que ele está isolado, podemos até adicionar os parâmetros genéricos sem sentir tanto o custo de legibilidade. O método ficará assim:

```
private TableColumn<Produto, String> criaColuna(String titulo,  
    int largura, String atributo) {  
  
    TableColumn<Produto, String> column =  
        new TableColumn<Produto, String>(titulo);  
    column.setMinWidth(largura);  
  
    column.setCellValueFactory(  
        new PropertyValueFactory<Produto, String>(atributo));  
  
    return column;  
}
```

Outras formas de refatoração

A extração de métodos é apenas um dos muitos tipos de refatorações. Renomear variáveis, classes e métodos também faz bastante diferença em nosso código, quanto mais claro for o nome, mais fácil será entender o algoritmo. Extrair variáveis é um outro tipo bastante comum e recomendado.

O Eclipse ajuda bastante nesse trabalho, há uma diversidade bem grande de atalhos para refatoração. A princípio, você não precisa decorar cada um desses atalhos, basta se lembrar do `Alt + Shift + T`. Ele mostra as possíveis refatorações para um determinado trecho de código. Por exemplo, ao usá-lo em uma classe:



Observe que algumas das sugestões são: renomear a classe, mover para outro pacote, extrair uma interface, entre diversas outras.

Desafio: mais e mais refatorações

As mudanças que fizemos aqui são bastante simples, meu objetivo foi mostrar o quanto isso pode significar em nosso código. Daqui para frente é com você. Que outras transformações poderiam melhorar esse projeto? Passe um tempo pesquisando a respeito, com toda certeza vai valer a pena. Vale lembrar que existem refatorações bem mais complexas, que afetam diversas partes de nossa aplicação e até mesmo a sua arquitetura.

8.2 OS TÃO POPULARES DESIGN PATTERNS

Não é à toa que eles são populares, vale muito a pena entender e aplicar esses conceitos sempre que for possível em nossos projetos. E isso **não muda dependendo da linguagem ou tecnologia que você estiver utilizando**, o que torna a ideia ainda mais interessante.

Design Patterns é um assunto muito extenso, precisaríamos de pelo menos

mais umas 200 páginas pra nos aprofundar nisso. Mas mesmo assim eu tentei mostrar aqui alguns dos mais fundamentais para nosso dia a dia desenvolvendo em Java. Relembrando alguns deles:

- **Factory Method:** você não quer deixar o código de criação de um objeto complicado espalhado por aí. Podemos estar falando de uma `Connection` do JDBC, como vimos no exemplo passado, ou qualquer outro. Ele não se limita a uma tecnologia específica. Nosso código ficou assim:

```
public class ConnectionFactory {  
  
    public Connection getConnection() {  
        String url = "jdbc:mysql://localhost/livraria";  
        try {  
            return DriverManager.getConnection(url, "root", "");  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

Quais as vantagens? Estamos **encapsulando** esse processo complicado que é criar uma conexão. Além disso, concentramos nossas informações de login, senha e nome do banco de dados em um único objeto. Se alguma dessas informações mudar, eu só preciso atualizar essa classe, **um único ponto da minha aplicação** e não os zilhares de lugares que acessam nosso banco.

- **DAO:** a ideia desse *pattern* é criar uma camada especialista em acessar dados de nossos objetos. Não importa se ele está usando JDBC, Hibernate ou qualquer outra tecnologia. As regras do MySQL, Oracle ou qualquer outro banco de dados ficam encapsuladas. Note que **encapsulamento** é um dos pilares de quase todos os *design patterns*. No lugar de sair acessando o banco de dados diretamente pelas nossas classes, passamos sempre pela camada do DAO:



Fig. 8.4: Acesso a dados encapsulado em um DAO.

- **Template Method:** apesar de não implementarmos esse *pattern* diretamente em nosso código, vimos ele em prática na API de IO do Java. `InputStream` ilustra esse exemplo, no fluxo de entrada. O fluxo é o mesmo, independente de onde vamos ler os bytes:

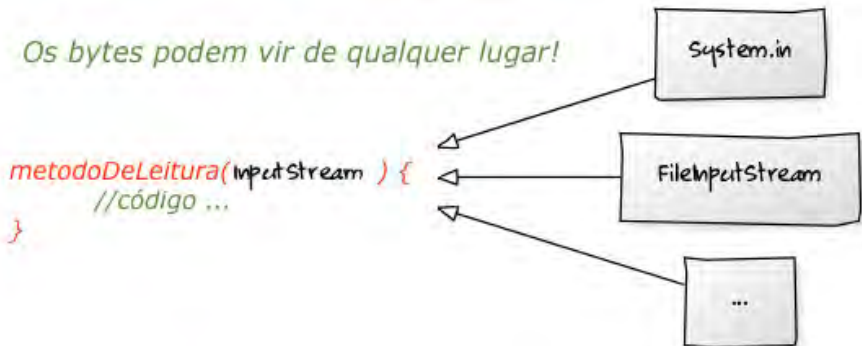


Fig. 8.5: Método recebendo um `InputStream` como parâmetro.

Esses são apenas alguns dos muitos que você provavelmente vai usar em seu dia a dia. Mesmo que não saiba, muitas vezes isso acontece.

CAPÍTULO 9

Próximos passos com Java

É isso, espero que tenha aproveitado bastante o conteúdo. Foi uma longa jornada, que para você talvez tenha começado no livro *Desbravando Java e Orientação a Objetos*. Nos dois livros tentei apresentar todos os conceitos com exemplos mão na massa e com um foco mais prático.

Não pare por aqui, continue praticando! Tente sempre ir além com novos testes e cenários além dos aqui propostos. Para mim, a prática é a forma mais efetiva de aprender e fixar qualquer coisa.

A partir daqui, você pode descobrir as muitas outras funcionalidades da linguagem, assim como suas milhares de bibliotecas, explorar novas APIs etc. Há muito mais o que explorar no ecossistema da plataforma Java.

9.1 ENTRE EM CONTATO CONOSCO

Não se esqueça de que você pode encaminhar suas dúvidas na lista que foi criada especialmente para este livro. Não só dúvidas! As sugestões, críticas, ideias e melhorias também são essenciais para nós:

<https://groups.google.com/group/livro-java>