

Desbravando Java e Orientação a Objetos

Um guia para o iniciante da linguagem



Casa do
Código

RODRIGO TURINI

© 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

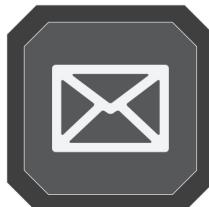
Rua Vergueiro, 3185 - 8º andar

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



Casa do Código
Livros para o programador

**Uma editora de livros técnicos
feita por desenvolvedores
para desenvolvedores.**



**Inscreva-se em nossa newsletter e
receba novidades e lançamentos**

www.casadocodigo.com.br/newsletter



Curta nossa fanpage no Facebook

www.facebook.com/casadocodigo



**Caelum:
Cursos de TI presenciais e online**

www.caelum.com.br



Dê seu feedback sobre o livro. Escreva para contato@casadocodigo.com.br

Já conhece
os nossos
títulos?



E muito mais em:
www.casadocodigo.com.br



“À minha esposa Jordana e ao maior presente que um dia sonhei receber, nossa filha Katherine.”

– Rodrigo Turini

Agradecimentos

Não há outra maneira de começar este livro que não seja agradecendo a todos que incentivaram e contribuíram direta ou indiretamente em sua edição. Victor Harada, Maurício Aniche e Guilherme Silveira foram alguns deles.

Fica um agradecimento especial ao Paulo Silveira, não só pela sua detalhada revisão e suas diversas sugestões e melhorias para esse livro, mas também pela influência direta que tem em meu dia a dia profissional.

Gostaria também de estender o agradecimento para toda equipe da Caelum. São profissionais exemplares que me incentivam a aprender e ensinar diariamente.

Sumário

1	Java	1
1.1	Nosso primeiro código Java	1
1.2	Algumas regras e convenções	3
1.3	Entendendo o método main	4
1.4	Trabalhando com uma IDE	6
1.5	Acesse o código desse livro e entre em contato conosco	9
2	Variáveis e tipos primitivos	11
2.1	Nosso projeto	11
2.2	Declaração e atribuição de variáveis	11
2.3	Tipos primitivos	14
2.4	Casting de valores	16
2.5	Adicionando condicionais	18
2.6	Loopings e mais loopings	21
3	Orientação a objetos	29
3.1	Criando um molde de livros	30
3.2	Criando um novo método	34
3.3	Objetos para todos os lados!	38
3.4	Entendendo a construção de um objeto	51
3.5	Vantagens da orientação a objetos	53

4 Encapsulamento	55
4.1 Limitando desconto do Livro	55
4.2 Isolando comportamentos	59
4.3 Código encapsulado	61
4.4 Getters e Setters	62
4.5 Definindo dependências pelo construtor	67
5 Herança e polimorfismo	75
5.1 Trabalhando com livros digitais	75
5.2 Reescrevendo métodos da superclasse	80
5.3 Regras próprias de um LivroFisico	85
5.4 Vendendo diferentes tipos de Livro	87
5.5 Acumulando total de compras	91
5.6 Herança ou composição?	94
6 Classe abstrata	95
6.1 Qual o tipo de cada Livro?	95
6.2 Minilivro não tem desconto!	98
6.3 Método abstrato	102
6.4 Relembrando algumas regras	104
7 Interface	105
7.1 O contrato Produto	108
7.2 Diminuindo acoplamento com Interfaces	112
7.3 Novas regras da interface no Java 8	114
8 Pacotes	119
8.1 Organizando nossas classes	119
8.2 Modificadores de acesso	126
9 Arrays e exception	129
9.1 Trabalhando com multiplicidade	129
9.2 As diferentes exceções e como lidar com elas	136
9.3 Muitas e muitas Exception	142
9.4 Também podemos lançar exceções!	147

10 Conhecendo a API	151
10.1 Todo objeto tem um tipo em comum	151
10.2 Wrappers dos tipos primitivos	161
10.3 O pacote java.lang	163
11 Collection Framework	169
11.1 O trabalho de manipular arrays	169
11.2 Ordenando nossa List de produtos	178
11.3 Gerenciando cupons de desconto	182
11.4 java.util.Map	188
12 Streams e novidades do Java 8	193
12.1 Ordenando com Java 8	193
12.2 forEach do Java 8	198
12.3 Filtrando livros pelo autor	198
13 Um pouco da história do Java	203
13.1 Origem da linguagem	203
13.2 Escreva uma vez, rode em qualquer lugar!	204
13.3 Linha do tempo	205
14 Continuando seus estudos	211
14.1 Entre em contato conosco	212

Versão: 18.0.6

CAPÍTULO 1

Java

1.1 NOSO PRIMEIRO CÓDIGO JAVA

No lugar de começar com conceitos e teorias, vamos logo partir para uma boa dose de prática! Abra seu editor de texto preferido e escreva o seguinte código Java:

```
class MeuPrimeiroPrograma {  
  
    public static void main(String[] args) {  
        System.out.println("O primeiro de muitos!");  
    }  
}
```

Esse programa imprime um texto simples. Confuso? Não se preocupe, em breve cada uma dessas palavras terá seu sentido bem claro.

Salve o arquivo em algum diretório de sua preferência, mas é importante que ele se chame `MeuPrimeiroPrograma.java`.

Nosso próximo passo será compilar esse código fonte. Faremos isso manualmente utilizando o `javac`, compilador padrão da Oracle.

Abra seu terminal e digite `javac MeuPrimeiroProgramma.java`. Note que para isso você precisa estar na mesma pasta do arquivo, ou passar o caminho completo para ele.

NAVEGANDO PELAS PASTAS E LISTANDO ARQUIVOS

Você pode utilizar o comando `cd` (*change directory*) para navegar pelas suas pastas via terminal. Um exemplo em Windows seria:

```
D:\> cd Desktop\livro  
D:\Desktop\livro>
```

O mesmo comando pode ser utilizado em um ambiente Unix (Linux ou Mac OS). Para listar, há uma diferença. Em Windows utilizamos o comando `dir`:

```
D:\Desktop\livro> dir  
    MeuPrimeiroProgramma.java  
    MeuPrimeiroProgramma.class
```

Porém, nos outros sistemas que foram citados o comando será `ls`. Repare:

```
turini ~ $ cd Desktop/livro  
turini/Desktop/livro ~ $ ls  
    MeuPrimeiroProgramma.java  
    MeuPrimeiroProgramma.class
```

Note que, se nenhum erro de compilação ocorrer, no mesmo diretório agora existirá um novo arquivo com o mesmo nome de nossa classe Java (neste caso, `MeuPrimeiroProgramma`), porém a sua extensão será `.class`. Este novo arquivo é o *bytecode* gerado pelo compilador.

Para executá-lo, ainda pelo terminal, digite o comando `java MeuPrimeiroPrograma`. O comando `java` é o responsável por executar a JVM (máquina virtual, ou *Java Virtual Machine*) que irá interpretar o *byte-code* de seu programa. Repare que neste comando não passamos a extensão do arquivo:

```
turini/Desktop/livro ~ $ java MeuPrimeiroProgramma  
O primeiro de muitos!
```

E pronto! Você já compilou e executou o seu primeiro programa em Java.

Não se preocupe, a JVM, *bytecode* e outros conceitos e siglas importantes serão melhor detalhados mais à frente.

INSTALAÇÃO DO JAVA

Para executar os códigos deste livro, você precisará ter o JDK (*Java Development Kit*) instalado. Se precisar, você pode seguir as instruções do link a seguir para instalar de acordo com seu sistema operacional:

[http://www.caelum.com.br/apostila-java-orientacao-objetos/
appendice-instalacao-do-java/](http://www.caelum.com.br/apostila-java-orientacao-objetos/appendice-instalacao-do-java/)

Precisa de ajuda? Não deixe de nos mandar um e-mail no grupo:

<https://groups.google.com/d/forum/livro-java-oo>

1.2 ALGUMAS REGRAS E CONVENÇÕES

Você pode ter reparado que seguimos algumas regras e convenções até agora. Vamos entendê-las um pouco melhor.

Podemos começar pelo nome de nosso arquivo, `MeuPrimeiroProgramma`. Em Java, o nome de uma classe sempre se inicia com letra maiúscula e, quando necessário, as palavras seguintes também têm seu *case* alterado. Dessa forma “*esse nome de classe*

vira “*EsseNomeDeClasse*

. Essa abordagem é bastante conhecida como *CamelCase*.

Um arquivo `.java` possui a definição de uma classe. É uma prática recomendada nomear a classe e o arquivo da mesma forma, caso contrário

poderá existir alguma confusão no momento de executá-la. Por exemplo, considere que temos um arquivo `meu-primeiro-programa.java` contendo a classe que chamamos de `MeuPrimeiroPrograma`. Para compilá-lo faremos:

```
turini/Desktop/livro ~ $ javac meu-primeiro-programa.java
```

Porém, o arquivo compilado (com o *bytecode*) terá o nome `MeuPrimeiroPrograma.class`. Como executar esse programa? A resposta será com o nome da classe, e não do arquivo. Neste caso:

```
turini/Desktop/livro ~ $ java MeuPrimeiroPrograma
```

Note também que há um par de chaves {} definindo o inicio e final (es-copo) de sua classe e as instruções sempre terminam com ponto e vírgula.

Por fim, é fundamental perceber que esta é uma linguagem *case sensitive*, ou seja, leva em consideração o *case* (caixa) em que as instruções são escritas. Escrever `System` com letra minúscula, por exemplo, resultaria em um erro de compilação.

Você pode ler mais a respeito dessas e de outras convenções da linguagem no documento oficial *Code Conventions for the Java Programming Language*, disponível no site da Oracle:

<http://www.oracle.com/technetwork/java/index-135089.html>

1.3 ENTENDENDO O MÉTODO MAIN

Ao ser executada, a máquina virtual (JVM) procura pelo bloco `main` declarado em sua classe. Esse é um bloco especial (ou **método**, como passaremos a chamar a partir de agora) e se parece com:

```
public static void main(String[] args) {  
    // seu código aqui  
}
```

Suas aplicações Java, em geral, vão possuir apenas um método `main`, um único ponto de partida. Quando rodamos o comando `java` passando o

nome de nossa classe, dissemos para a JVM executar todo o conteúdo que estiver dentro do corpo (das chaves) desse método. Em nosso exemplo, foi uma simples instrução de impressão, o `System.out.println("o primeiro de muitos!")`.

Há ainda como passar argumentos para o método `main` ao executar um programa Java. O parâmetro `String[] args` que o método recebe dentro de seus parênteses será o responsável por armazenar esses argumentos para que possam ser acessados em nosso código. Mas, afinal, o que é esse `String[]`? Um array de `String` em Java, porém, por enquanto não estamos interessados em entender um *array* e todos os seus detalhes, tudo o que precisamos saber é que ele vai armazenar uma multiplicidade de `Strings`.

Um exemplo dessa passagem de parâmetro seria:

```
turini/Desktop/livro ~ $ java MeuPrimeiroPrograma Java Rodrigo
```

Para isso funcionar, modificamos nosso código para exibir o conteúdo guardado nas posições `0` e `1` (`args[0]`, `args[1]`) desse conjunto de argumentos:

```
class MeuPrimeiroProgramma {  
  
    public static void main(String[] args) {  
        System.out.println("O primeiro de muitos códigos  
                        escritos em " +args[0]+ " pelo " +args[1]+ "!");  
    }  
}
```

Execute para conferir o resultado! Não se esqueça que, como modificamos nosso código, precisaremos compilar novamente:

```
turini/Desktop/livro ~ $ javac MeuPrimeiroProgramma.java  
turini/Desktop/livro ~ $ java MeuPrimeiroProgramma Java Rodrigo
```

Nesse exemplo, o resultado impresso será:

O primeiro de muitos códigos escritos em Java pelo Rodrigo!

ERRO DE COMPILAÇÃO?

Um erro de digitação, a falta de um ponto e vírgula ou uma diferença de *case* em seu código são alguns dos muitos motivos que podem resultar em um erro de compilação. Conhecer e entender esses erros é fundamental, talvez você queira inclusive provocar algum deles para ver como seu código se comporta.

Qual será a mensagem caso eu esqueça de escrever um ponto e vírgula? Escreva um teste simples pra descobrir! Um exemplo:

```
System.out.println("sem ponto-e-virgula")
```

A mensagem de erro será:

```
Syntax error, insert ";" to complete BlockStatements
```

Lembre-se que você pode e deve tirar todas as suas dúvidas no GUJ, em <http://guj.com.br>.

1.4 TRABALHANDO COM UMA IDE

Escrever, compilar e executar seu código Java em um bloco de notas junto com terminal é bastante trabalhoso. É fundamental conhecer esse processo, mas em seu dia a dia você provavelmente vai preferir utilizar alguma das mais diversas ferramentas de desenvolvimento conhecidas para ajudá-lo nesse trabalho.

Essas ferramentas são chamadas de IDE (*Integrated Development Environment*) e podem tornar seu desenvolvimento muito mais produtivo e interessante, oferecendo-lhe recursos como *syntax highlight* e *auto complete* das instruções de seu código.

No decorrer deste livro, vamos utilizar o *Eclipse*, você pode fazer o download de sua versão mais recente em:

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

Essa é uma IDE gratuita e open source, sem dúvida uma das preferidas do mercado e instituições de ensino. Seus diversos atalhos e templates prontos

são grandes diferenciais.

Existem diversas outras boas opções no mercado, como por exemplo o *NetBeans* da Oracle, ou o *IntelliJ IDEA*, sendo este último pago.

Criando seu primeiro projeto no Eclipse

Depois de fazer o download, você não precisará instalar a IDE. Esta é uma executável inteiramente escrita em Java, basta executá-la e escolher a pasta onde será a sua área de trabalho (*workspace*). Escolha um local de sua preferência.

Vamos criar nosso projeto! Para isso, você pode por meio do menu escolher as opções `File > New > Java Project`. Vamos chamar o projeto que desenvolveremos durante o curso de *livraria*.

Repare que, depois de concluir, esse projeto será representando em seu sistema operacional como uma pasta, dentro do *workspace* que você escolheu. Por enquanto, dentro desse diretório você encontrará a pasta `src`, onde ficará todo o seu código `.java` e também a pasta `bin`, onde ficará o `bytecode` compilado.

Produtividade extrema

Agora que já criamos o projeto, vamos criar nossa primeira classe pelo Eclipse. Você pode fazer isso pelo menu `File > New > Class`. Para conhecer um pouco da IDE, vamos criar novamente a classe `MeuPrimeiroPrograma`.

Depois de preencher o nome, selecione a opção *finish* e veja o resultado:

```
public class MeuPrimeiroProgramma {  
}
```

A estrutura de sua classe já está pronta, vamos agora escrever seu método `main`. Para fazer isso, escreva a palavra `main` dentro de sua classe e pressione o atalho `Control + Espaço`.

Esse é o atalho de *code completion* da IDE. Se tudo correu bem, seu código ficou assim:

```
public class MeuPrimeiroPrograma {  
  
    public static void main(String[] args) {  
  
    }  
}
```

Interessante, não acha? Além de termos mais produtividade ao escrever, evitamos que erros de digitação aconteçam.

Vamos além, agora dentro do método `main`, digite `sys0` e pressione `Control + Espaço` para fazer *code completion* novamente.

Você pode e deve usar e abusar desse recurso!

```
public class MeuPrimeiroProgramma {  
  
    public static void main(String[] args) {  
        System.out.println("0 primeiro de muitos!");  
    }  
}
```

Agora, com o código pronto, as próximas etapas seriam compilar e executá-lo, mas a compilação já está pronta!

Isso mesmo, conforme você vai escrevendo seu código, a IDE já cuida de compilá-lo. Repare que se você apagar o ponto e vírgula, por exemplo, essa linha ficará sublinhada em vermelho. Essa é uma indicação visual de que seu código não está compilando.

Para executar o código, você pode clicar com o botão direito do mouse em sua classe e selecionar as opções `Run As > Java Application`. Ou por atalho, pressionando `Control + F11`.

Repare que a saída de seu código vai aparecer na aba `Console`.

```
0 primeiro de muitos!
```

ATALHOS DO ECLIPSE

Conhecer os atalhos da IDE vai torná-lo um programador muito mais produtivo. Você pode ler mais a respeito e conhecer mais atalhos pela documentação do Eclipse em:

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

E também no post:

<http://blog.caelum.com.br/as-tres-principais-teclas-de-atalho-do-eclipse/>

Se esse for o seu primeiro contato com a linguagem, recomendamos que você pratique bastante a sintaxe antes de partir para os próximos capítulos. Sinta-se confortável com o processo de escrever, compilar e executar seu código Java. Tenho certeza de que em breve essa será sua rotina.

1.5 ACESSO O CÓDIGO DESSE LIVRO E ENTRE EM CONTACTO CONOSCO

Todos os exemplos deste livro podem ser encontrados no repositório:

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

Mas claro, não deixe de escrever todo o código que vimos para praticar a sintaxe e se adaptar com os detalhes da linguagem. Além disso, sempre que possível faça novos testes além dos aqui sugeridos.

Ficou com alguma dúvida? Não deixe de me mandar um e-mail. A seguinte lista foi criada exclusivamente para facilitar o seu contato conosco e com os demais leitores:

<https://groups.google.com/d/forum/livro-java-oo>

Suas sugestões, críticas e melhorias serão muito mais do que bem-vindas!

Outro recurso que você pode usar para esclarecer suas dúvidas e participar ativamente na comunidade Java é o fórum do GUJ, espero encontrá-lo por lá.

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

CAPÍTULO 2

Variáveis e tipos primitivos

2.1 NOSO PROJETO

Durante o livro, vamos criar e evoluir a aplicação Java de uma livraria como a *Casa do Código*. Esse contexto vai nos possibilitar colocar em prática todos os principais recursos e conceitos da linguagem, desde o mais simples ao mais avançado, além de conhecer e trabalhar com as *APIs* e *features* de sua mais nova versão.

2.2 DECLARAÇÃO E ATRIBUIÇÃO DE VARIÁVEIS

Vamos criar uma nova classe Java cujo objetivo será calcular o valor total de livros do nosso estoque na livraria. Podemos chamá-la de `CalculadoraDeEstoque`.

```
public class CalculadoraDeEstoque {  
  
    public static void main(String[] args) {  
  
    }  
}
```

Precisamos agora armazenar dentro de seu método `main` o valor de nossos livros. Uma forma simples de fazer isso seria:

```
public class CalculadoraDeEstoque {  
  
    public static void main(String[] args) {  
  
        double livroJava8;  
        double livroTDD;  
    }  
}
```

Dessa forma, estamos pedindo ao Java que reserve regiões da memória para futuramente armazenar esses valores. Essa instrução que criamos é conhecida como uma variável.

Repare que essas duas variáveis declaradas possuem um tipo `double` (número com ponto flutuante). Logo, conheceremos os outros tipos existentes, mas é importante reconhecer desde já que toda variável em Java precisaria ter um.

Utilizamos o sinal `=` (igual) para atribuir um valor para estas variáveis que representam alguns de nossos livros:

```
double livroJava8;  
double livroTDD;  
  
livroJava8 = 59.90;  
livroTDD = 59.90;
```

Mas, neste caso, como já sabemos o valor que será atribuído no momento de sua declaração, podemos fazer a atribuição de forma direta no momento em que cada variável é declarada:

```
double livroJava8 = 59.90;
double livroTDD = 59.90;
```

Agora que já temos alguns valores de livros, vamos calcular a sua soma e acumular em uma nova variável. Isso pode ser feito da seguinte forma:

```
double soma = livroJava8 + livroTDD;
```

Pronto, isso é tudo que precisamos por enquanto para completar a nossa `CalculadoraDeEstoque`. Após somar esses valores, vamos imprimir o valor do resultado no *console*:

```
public class CalculadoraDeEstoque {

    public static void main(String[] args) {

        double livroJava8 = 59.90;
        double livroTDD = 59.90;

        double soma = livroJava8 + livroTDD;

        System.out.println("O total em estoque é " + soma);
    }
}
```

Escreva e execute esse código para praticar! Neste exemplo, o resultado será:

```
O total em estoque é 119.8
```

Além do operador de soma (+), você também pode usar os seguintes operadores em seu código Java:

Operador	Função
+	soma
-	subtração
*	multiplicação
/	divisão
%	modulo

2.3 TIPOS PRIMITIVOS

Vimos que é possível representar um número com ponto flutuante utilizando o tipo `double`, mas existem diversos outros tipos para representar os diferentes valores com que trabalhamos no dia a dia.

Para representar um número inteiro, por exemplo, podemos utilizar os tipos `byte`, `short`, `int` ou `long`. Isso mesmo, o código a seguir compila com qualquer um desses tipos:

```
byte inteiro1 = 10;
short inteiro2 = 10;
int inteiro3 = 10;
long inteiro4 = 10;
```

Mas, afinal, quando eu devo utilizar cada um desses? A grande diferença está no tamanho de cada um desses tipos. Por exemplo, um `short` suporta até 2 bytes, enquanto um `long` suporta 8 bytes.

Muito diferente de antigamente, hoje não há uma preocupação tão grande em economizar bytes na declaração de suas variáveis. Você raramente verá um programador utilizando um `short` para guardar um número pequeno,

no lugar de usar logo um `int`. Mas claro, conhecer os diferentes tipos e a capacidade de cada um deles é essencial para um bom programador.

Esse são os possíveis tipos primitivos da linguagem Java:

Operador	Tamanho
<code>boolean</code>	1 bit
<code>byte</code>	1 byte
<code>short</code>	2 byte
<code>char</code>	2 byte
<code>int</code>	4 byte
<code>float</code>	4 byte
<code>long</code>	8 byte
<code>double</code>	8 byte

Um detalhe simples, porém muito importante, sobre os tipos primitivos é que, quando você atribui um valor para eles (utilizando o operador `=`), este valor será **copiado** para a sua variável. Por exemplo, repare no seguinte código:

```
int numero = 4;  
int outroNumeroIgual = numero;  
numero = numero + 5;  
  
System.out.println(numero);  
System.out.println(outroNumeroIgual);
```

Neste caso, com certeza o valor da variável `numero` no momento em que for impressa será 9, mas e quanto ao `outroNumeroIgual`? Repare que ela recebe `numero` e seu valor foi alterado logo em seguida.

Execute esse código e você perceberá que o `outroNumeroIgual` continuará com o valor 4, pois este era o valor da variável `numero` no momento da atribuição. Ou seja, a variável `outroNumeroIgual` guardou uma cópia do valor, e não uma referência a ela ou algo do tipo.

TIPOS NÃO PRIMITIVOS

Logo conhceremos outros tipos (não primitivos) da linguagem Java, mas desde já repare que um texto, assim como qualquer outro valor, também tem um tipo! Esse tipo é conhecido por `String`.

Você poderia, no lugar de fazer:

```
System.out.println("Eu sou uma String");
```

Declarar da seguinte forma:

```
String texto = "Eu sou uma String";
System.out.println(texto);
```

2.4 CASTING DE VALORES

Como você já deve ter percebido, nem todos os valores são compatíveis. Por exemplo, se eu tentar declarar o valor de nossos livros na `CalculadoraDeEstoque` como um `int`, um erro de compilação acontecerá.

```
public class CalculadoraDeEstoque {

    public static void main(String[] args) {

        int livroJava8 = 59.90;
        int livroTDD = 59.90;
```

```
    int soma = livroJava8 + livroTDD;

    System.out.println("O total em estoque é " + soma);
}
}
```

Repare no Eclipse (ou qualquer outra IDE ou editor de texto que estiver utilizando em seus testes) que o seguinte erro será exibido:

```
Type mismatch: cannot convert from double to int
```

Ou seja, um `int` não pode guardar um número com ponto flutuante (`double`). Faz sentido.

Mas, sim, o contrário é totalmente possível. Repare no código:

```
public class CalculadoraDeEstoque {

    public static void main(String[] args) {

        double livroJava8 = 60;
        double livroTDD = 60;

        double soma = livroJava8 + livroTDD;

        System.out.println("O total em estoque é " + soma);
    }
}
```

Mesmo arredondando o valor dos livros para um número inteiro (neste caso, 60), um `double` pode sim guardar esse valor. Portanto, eu posso atribuir valores menores em variáveis com uma capacidade maior; o que eu não posso é o contrário. Uma analogia dev que gosto muito e se encaixa bem aqui é: “*Você pode colocar uma formiga na casa de um cavalo, o contrário não daria certo*”.

Mas repare que nem mesmo o seguinte código compila:

```
double livroJava8 = 60;
int numeroInteiro = livroJava8;
```

Da mesma forma que antes, estamos tentando atribuir um `double` (que é um tipo de tamanho maior) em um `int`. Não importa se o valor que atribuímos a este `double` é um valor inteiro válido (um bom candidato para o tipo `int`). O compilador vai se preocupar com o tipo de suas variáveis, e não com seus valores, até porque, como o próprio nome indica, esses valores são mutáveis, portanto essa atribuição seria perigosa.

Em algum momento, você pode precisar forçar essa conversão de tipos, dizendo ao compilador: *“tudo bem, eu garanto que o valor deste double pode ser moldado para um numero inteiro”*

. Para isso, utilizamos um recurso conhecido como `casting`.

```
int numeroInteiro = (int) livroJava8;
```

Neste caso, seu código vai compilar sem nenhum problema e a variável `numeroInteiro` terá uma cópia do valor 60.

Se o valor do `livroJava8` fosse 59.90, ou seja, um número com ponto flutuante e você fizesse esse `casting` haveria uma perda de precisão. A variável `numeroInteiro` teria apenas o valor 59 copiado.

2.5 ADICIONANDO CONDICIONAIS

Nossa `CalculadoraDeEstoque` precisa de uma nova funcionalidade. Se o valor total de livros for menor que 150 reais, precisamos ser alertados de que nosso estoque está baixíssimo, caso contrário, devemos mostrar uma mensagem indicando de que está tudo sob controle!

Em Java, podemos fazer essa condicional de uma forma bem comum, utilizando um `if` e `else`. Observe:

```
public class CalculadoraDeEstoque {  
  
    public static void main(String[] args) {  
  
        double livroJava8 = 59.90;  
        double livroTDD = 59.90;  
  
        double soma = livroJava8 + livroTDD;  
  
        if (soma < 150) {  
            System.out.println("Atenção! Nossa loja  
            está com estoque baixíssimo!");  
        } else {  
            System.out.println("Tudo sob controle!");  
        }  
    }  
}
```

```
System.out.println("O total em estoque é "+ soma);

if (soma < 150) {
    System.out.println("Seu estoque está muito baixo!");
} else {
    System.out.println("Seu estoque está bom");
}
}
```

Como já é esperado, esse código só vai imprimir a mensagem de estoque baixo se o valor da soma for menor (`<`) que 150. Caso contrário, irá executar o conteúdo de dentro do bloco `else`.

Passamos uma condição como argumento fazendo uma comparação entre o valor da variável `soma` com o valor 150. Essa condição vai resultar em um valor `true` quando verdadeira, ou `false` caso contrário. Esse tipo de condição é conhecido como **expressão booleana**. Seu resultado sempre será do tipo `boolean`:

```
boolean resultado = soma < 150;
```

Você pode usar qualquer um dos seguintes operadores relacionais pra construir uma expressão booleana: `>` (maior), `<` (menor), `>=` (maior ou igual), `<=` (menor ou igual), `==` (igual sim, são dois iguais! Lembre-se que um único igual significa atribuição) e, por fim, `!=` (diferente).

Há ainda a alternativa de encadear mais condições em nosso `if`. Por exemplo, para receber uma mensagem indicando que o estoque está muito alto, podemos adicionar a seguinte condição:

```
public class CalculadoraDeEstoque {

    public static void main(String[] args) {

        double livroJava8 = 59.90;
        double livroTDD = 59.90;

        double soma = livroJava8 + livroTDD;
```

```
System.out.println("O total em estoque é "+ soma);

if (soma < 150) {
    System.out.println("Seu estoque está muito baixo!");
} else if (soma >= 2000) {
    System.out.println("Seu estoque está muito alto!");
} else {
    System.out.println("Seu estoque está bom");
}
}
```

OPERADOR TERNÁRIO

É muito comum escrevermos condicionais como a seguinte, que, de acordo com alguma condição booleana, retorna um valor diferente. Repare:

```
double valor = 0;
if (v1 > v2) {
    valor = 100;
} else {
    valor = 0;
}
```

Uma alternativa bastante conhecida e utilizada para estes casos é o operador ternário `? :`. Com ele, você pode fazer o mesmo da seguinte forma:

```
double valor = v1 > v2 ? 100 : 0;
```

Sem dúvida, fica bem mais enxuto, não acha? Performaticamente, não há nenhuma vantagem significante em usar uma ou outra estratégia, mas é sempre interessante levar em consideração a legibilidade do código. Em alguns casos, usar o operador ternário pode custar um pouco da legibilidade e deixar o código mais complexo.

2.6 LOOPINGS E MAIS LOOPINGS

Considerando que todos os livros têm o preço 59.90, seria muito trabalhoso criar tantas variáveis para que o valor da `soma` ultrapasse 2000 reais. Para nos ajudar na construção desses livros podemos criar uma estrutura de repetição (um `looping`). Uma das formas de se fazer isso seria utilizando o `while`:

```
double soma = 0;
int contador = 0;

while (contador < 35) {
    double valorDoLivro = 59.90;
    soma = soma + valorDoLivro;
    contador = contador + 1;
}
```

O `while` é bastante parecido com o `if`. A grande diferença é que, enquanto sua expressão booleana for `true`, seu código continuará sendo executado. Neste exemplo estamos criando uma variável `soma` para acumular o valor total dos livros e também uma variável `contador` para controlar a quantidade de vezes que queremos iterar.

Note que estamos adicionando 35 livros, já que a condição é `contador < 35` e que a cada iteração incrementamos 1 ao valor do `contador`.

Nosso código completo fica da seguinte forma:

```
public class CalculadoraDeEstoque {

    public static void main(String[] args) {

        double soma = 0;
        int contador = 0;

        while (contador < 35) {
            double valorDoLivro = 59.90;
            soma = soma + valorDoLivro;
            contador = contador + 1;
        }
    }
}
```

```
System.out.println("O total em estoque é "+ soma);

if (soma < 150) {
    System.out.println("Seu estoque está muito baixo!");
} else if (soma >= 2000) {
    System.out.println("Seu estoque está muito alto!");
} else {
    System.out.println("Seu estoque está bom");
}
}
```

Ao executá-lo a saída será:

```
O total em estoque é 2096.5000000000014
Seu estoque está muito alto!
```

LOOPING INFINITO

É muito comum esquecermos de modificar o valor que está sendo comparado dentro do `while` ou nas diferentes formas que veremos para se fazer um *looping*. Isso vai resultar no que chamamos de *looping infinito*, uma vez que a condição de looping sempre será `true`. Se quiser testar, adicione um comentário na linha que incrementa o valor do `contador` e execute o código:

```
while (contador < 35) {
    double valorDoLivro = 59.90;
    soma = soma + valorDoLivro;
    // contador = contador + 1;
}
```

Podemos deixar nosso código ainda mais enxuto utilizando o operador de atribuição `+=`. Basta, no lugar de fazer:

```
soma = soma + valorDoLivro;
```

Mudarmos o código para:

```
soma += valorDoLivro;
```

Essa é uma forma equivalente de se incrementar o valor de uma variável. O mesmo poderia ser feito com nosso contador, repare:

```
contador += 1;
```

Podemos utilizar essa técnica com os seguintes operadores:

Operador	Equivalente
<code>+=</code>	<code>x = x + x;</code>
<code>-=</code>	<code>x = x - x;</code>
<code>*=</code>	<code>x = x * x;</code>
<code>/=</code>	<code>x = x / x;</code>
<code>%=</code>	<code>x = x % x;</code>

Mas para esse caso há uma forma ainda mais enxuta, utilizando o operador unário `++`:

```
contador ++;
```

O código do `while` agora fica da seguinte forma:

```
while (contador < 35) {
    double valorDoLivro = 59.90;
    soma += valorDoLivro;
    contador++;
}
```

Nosso código pode ficar ainda mais enxuto utilizando uma outra forma de *looping* extremamente conhecida e utilizada, o `for`.

Observe que a estrutura que utilizamos no `while` foi parecida com:

```
// inicialização do contador  
  
while (condição) {  
  
    // atualização do contador  
}
```

Com o `for`, podemos fazer isso de uma forma ainda mais direta:

```
for(inicialização; condição; atualização) {  
  
}
```

Isso mesmo, além da condição *booleana* ele também reserva um local para inicialização de variáveis e atualização de seus valores. Com o `for` nosso código pode ser escrito assim:

```
double soma = 0;  
for (int contador = 0; contador < 35; contador++) {  
    soma += 59.90;  
}
```

Sem dúvida, é uma forma mais direta. Você muitas vezes verá essa variável `contador` que criamos ser chamada de `i` (*index*, ou índice). Seguindo esse padrão, nosso código completo fica assim:

```
public class CalculadoraDeEstoque {  
  
    public static void main(String[] args) {  
  
        double soma = 0;  
  
        for (double i = 0; i < 35; i++) {  
            soma += 59.90;  
        }  
    }  
}
```

```
System.out.println("O total em estoque é " + soma);

if (soma < 150) {
    System.out.println("Seu estoque está muito baixo!");
} else if (soma >= 2000) {
    System.out.println("Seu estoque está muito alto!");
} else {
    System.out.println("Seu estoque está bom");
}
}
```

TRABALHANDO COM **CONTINUE** E **BREAK**

Você pode utilizar a palavra-chave `continue` para pular uma iteração de seu looping e forçar a execução do próximo laço. O código a seguir vai imprimir todos os números de 0 a 10, mas vai pular o número 7:

```
for(int i = 0; i <= 10; i++) {  
    if (i == 7) {  
        continue;  
    }  
    System.out.println(i);  
}
```

Outra possibilidade comum é parar a execução de um looping dada uma determinada condição. Para isso, utilizamos a palavra-chave `break`:

```
for(int i = 0; i <= 10; i++) {  
    if (i == 7) {  
        break;  
    }  
    System.out.println(i);  
}
```

Neste caso, apenas os números de 0 a 6 serão impressos.

OPERADORES LÓGICOS

Você pode e deve usar operadores lógicos em suas expressões booleanas sempre que necessário. Por exemplo, repare no código a seguir:

```
if (v1 > v2) {  
    if (v2 < v3) {  
        // é > que v1 e < que v3  
    }  
}
```

Esse encadeamento de `ifs` prejudica um pouco a legibilidade, não acha? Que tal fazer:

```
if (v1 > v2 && v2 < v3) {  
    // é > que v1 e < que v3  
}
```

O código terá o mesmo efeito com um único `if`, agora usando o operador *and* (`&&`). Outras opções são o *or* (`||`) e a negativa (`!`). Para negar uma condição booleana tudo o que você precisa fazer é adicionar o sinal `!` antes de sua declaração, veja:

```
boolean condicao = 1 > 0;  
if (!condicao) {  
    // não vai entrar aqui  
}
```


CAPÍTULO 3

Orientação a objetos

A linguagem Java tem como forte característica ter como paradigma a **orientação a objetos**, que estudaremos profundamente no decorrer do livro. Esse paradigma existe desde a década de 70, mas foi depois do surgimento do Java que ficou bastante famoso e que passou a ser levado mais a sério.

Repare que nossa `CalculadoraDeEstoque` está fazendo todo o trabalho dentro de seu método `main`, ainda de forma muito procedural. A orientação a objetos propõe uma maneira diferente de fazer isso, você passa a trabalhar de um jeito mais próximo à realidade humana. Para cada necessidade importante teremos objetos que interagem entre si e que são compostos por estado (atributos) e comportamento (métodos). Quer um exemplo? Observe como estamos representando o preço de nossos livros:

```
double soma = 0;
```

```
for (double i = 0; i < 35; i++) {  
    soma += 59.90;  
}
```

O valor 59.90 está fazendo isso. Ele representa o valor do livro; mas, e quanto ao seu nome, descrição e demais informações? Todas essas informações representam o que um livro tem e são extremamente importantes para nosso sistema. O grande problema do paradigma procedural é que não existe uma forma simples de conectar todos esses elementos, já na orientação a objetos podemos fazer isso de um jeito muito simples! Assim como no contexto real, podemos criar um objeto para representar tudo o que um livro tem e o que ele faz.

UMA NOVA FORMA DE PENSAR

Se você já está acostumado com algum outro paradigma, esse é o momento de abrir a sua mente. Repare que esse é um paradigma totalmente diferente, você precisará pensar de maneira diferente e escrever seu código de outra forma.

3.1 CRIANDO UM MOLDE DE LIVROS

Vamos criar uma nova classe Java chamada `Livro`. Essa classe será um molde, que representará o que um livro deve ter e como ele deve se comportar em nosso sistema.

```
public class Livro {  
}
```

Para deixar essa classe mais interessante, vamos adicionar alguns campos para representar o que um livro tem. Esses são os **atributos** de nossa classe:

```
public class Livro {  
  
    String nome;
```

```
    String descricao;
    double valor;
    String isbn;
}
```

Repare que esses atributos são muito parecidos com variáveis que podemos criar dentro de nosso método `main`, por exemplo. Mas eles não estão dentro de um bloco e, sim, dentro do escopo da classe, por isso recebem o nome diferenciado de atributo.

Nosso molde já está pronto para uso. Por enquanto, um livro terá um nome, descrição, valor e ISBN (um número de identificação, *International Standard Book Number*).

Esses campos não serão populados na classe `Livro`, ela é apenas o molde! O que precisamos é criar um objeto a partir desse molde. Para fazer isso, utilizamos a palavra-chave `new`.

```
Livro livro = new Livro();
```

Observe que a variável `livro` tem um tipo, assim como qualquer variável em Java, mas diferente de um `int`, `double` ou `String` como já estamos acostumados, agora seu tipo é a própria classe `Livro`.

Ao criar um objeto de `Livro` e atribuir a uma variável `livro`, estamos estabelecendo uma forma de nos referenciar a esse objeto que até então não tem nenhum valor populado.

Populando os atributos do livro

Agora que fizemos isso, a partir da variável `livro`, podemos acessar o objeto que foi criado em memória e popular os seus atributos. Um exemplo seria:

```
Livro livro = new Livro();
livro.nome = "Java 8 Prático";
livro.descricao = "Novos recursos da linguagem";
livro.valor = 59.90;
livro.isbn = "978-85-66250-46-6";
```

Repare que utilizamos um . (ponto) para acessar os atributos desse objeto em específico. Você também pode recuperar as informações adicionadas nos seus objetos acessando seus atributos da seguinte forma:

```
System.out.println("O nome do livro é " + livro.nome);
```

Neste caso, a saída será:

O nome do livro é Java 8 Prático

Vamos criar uma nova classe chamada `CadastroDeLivros` em nosso projeto. Ela deve ter um método `main` que cria um novo livro, preenche alguns de seus atributos e depois imprime os seus valores. Algo como:

```
public class CadastroDeLivros {  
  
    public static void main(String[] args) {  
  
        Livro livro = new Livro();  
        livro.nome = "Java 8 Prático";  
        livro.descricao = "Novos recursos da linguagem";  
        livro.valor = 59.90;  
        livro.isbn = "978-85-66250-46-6";  
  
        System.out.println(livro.nome);  
        System.out.println(livro.descricao);  
        System.out.println(livro.valor);  
        System.out.println(livro.isbn);  
    }  
}
```

Execute a classe para ver a saída! Deverá ser:

```
Java 8 Prático  
Novos recursos da linguagem  
59.9  
978-85-66250-46-6
```

CLASSE X OBJETO

Uma classe é apenas um molde. Uma especificação que define para a máquina virtual o que um objeto desse tipo deverá ter e como ele deve se comportar. Nossa livraria poderá ter milhares de livros (objetos), mas existirá apenas uma classe `Livro` (molde). Cada objeto que criarmos do tipo `Livro` terá seus próprios valores, ou seja, cada livro terá o seu próprio nome, sua descrição, um valor e um número de ISBN.

Nosso `CadastroDeLivros` está muito simples, com um único livro. Vamos criar mais um, já com seus atributos preenchidos:

```
public class CadastroDeLivros {  
  
    public static void main(String[] args) {  
  
        Livro livro = new Livro();  
        livro.nome = "Java 8 Prático";  
        livro.descricao = "Novos recursos da linguagem";  
        livro.valor = 59.90;  
        livro.isbn = "978-85-66250-46-6";  
  
        System.out.println(livro.nome);  
        System.out.println(livro.descricao);  
        System.out.println(livro.valor);  
        System.out.println(livro.isbn);  
  
        Livro outroLivro = new Livro();  
        outroLivro.nome = "Lógica de Programação";  
        outroLivro.descricao = "Crie seus primeiros programas";  
        outroLivro.valor = 59.90;  
        outroLivro.isbn = "978-85-66250-22-0";  
  
        System.out.println(outroLivro.nome);  
        System.out.println(outroLivro.descricao);  
        System.out.println(outroLivro.valor);  
        System.out.println(outroLivro.isbn);  
    }  
}
```

```
    }  
}
```

Porém, observe que até agora nossa classe `Livro` só tem atributos, ou seja, só guarda valores. Vimos que uma classe vai além, ela também pode ter comportamentos (métodos).

3.2 CRIANDO UM NOVO MÉTODO

A todo momento que criamos um novo objeto do tipo `Livro`, estamos imprimindo seus valores. Essa é uma necessidade comum entre todos os livros de nosso sistema, mas da forma como estamos fazendo, toda hora repetimos as mesmas 4 linhas de código:

```
System.out.println(livro.nome);  
System.out.println(livro.descricao);  
System.out.println(livro.valor);  
System.out.println(livro.isbn);
```

A única coisa que muda é o nome da variável, de `livro` para `outroLivro`.

Pense na seguinte situação, em breve nosso cadastro terá cerca de 100 livros. Para cada livro, teremos 4 linhas de código imprimindo os seus 4 atributos. Ou seja, teremos 400 linhas de código muito parecidas, praticamente repetidas.

Essa repetição de código sempre tem um efeito colateral desagradável, que é a dificuldade de manutenção. Quer ver? O que acontece se adicionarmos um novo atributo no livro, por exemplo a data de seu lançamento? Para imprimir a data toda vez que criar um livro, teremos que mudar 100 partes de nosso código, adicionando o `System.out.println(livro.dataDeLancamento)`.

No lugar de deixar essa lógica de impressão dos dados do livro toda espalhada, podemos isolar esse comportamento comum entre os livros na classe `Livro`! Para isso, criamos um **método**. Uma forma seria:

```
void mostrarDetalhes() {  
    System.out.println(nome);  
    System.out.println(descricao);  
    System.out.println(valor);  
    System.out.println(isbn);  
}
```

Esse método define um comportamento para classe `Livro`. Repare que a sintaxe de um método é um pouco diferente do que estamos acostumados, sua estrutura é:

```
tipoDeRetorno nomeDoComportamento() {  
  
    // código que será executado  
}
```

Nesse caso, não estamos retornando nada e, sim, apenas executando instruções dentro do método, portanto, seu tipo de retorno é `void`. Essa palavra reservada indica que um método não tem retorno.

Um método também pode ter variáveis declaradas, como por exemplo:

```
void mostrarDetalhes() {  
    String mensagem = "Mostrando detalhes do livro ";  
    System.out.println(mensagem);  
    System.out.println(nome);  
    System.out.println(descricao);  
    System.out.println(valor);  
    System.out.println(isbn);  
}
```

A variável `mensagem` foi declarada dentro do método, logo esse será seu escopo, ou seja, ela só existirá e poderá ser utilizada dentro do método `mostrarDetalhes`.

Nossa classe `Livro` ficou desta forma:

```
public class Livro {  
  
    String nome;  
    String descricao;
```

```
    double valor;
    String isbn;

    void mostrarDetalhes() {
        String mensagem = "Mostrando detalhes do livro ";
        System.out.println(mensagem);
        System.out.println(nome);
        System.out.println(descricao);
        System.out.println(valor);
        System.out.println(isbn);
    }
}
```

Agora que cada livro possui o comportamento de exibir os seus detalhes, podemos remover as linhas que faziam esse trabalho no método `main` e passar a invocar esse novo método. Repare:

```
public class CadastroDeLivros {

    public static void main(String[] args) {

        Livro livro = new Livro();
        livro.nome = "Java 8 Prático";
        livro.descricao = "Novos recursos da linguagem";
        livro.valor = 59.90;
        livro.isbn = "978-85-66250-46-6";

        livro.mostrarDetalhes();

        Livro outroLivro = new Livro();
        outroLivro.nome = "Lógica de Programação";
        outroLivro.descricao = "Crie seus primeiros programas";
        outroLivro.valor = 59.90;
        outroLivro.isbn = "978-85-66250-22-0";

        outroLivro.mostrarDetalhes();
    }
}
```

Ao isolar esse comportamento dentro da classe `Livro` já tivemos um ganho evidente. Tivemos que escrever menos linhas em nosso `main`, o código ficou com menos repetições pois teve maior reaproveitamento. Mas essa não é a única vantagem em isolar um comportamento: mesmo que eu tenha 100 livros cadastrados e use o método `mostrarDetalhes` 100 vezes, em quantas partes do meu código eu terei que mexer para modificar a forma como os livros são exibidos? A resposta é: uma!

Agora que o comportamento está isolado, só precisaremos alterar o método, um único ponto do nosso código, para refletir a mudança em todos os lugares que o usam. Vamos colocar isso em prática, basta mudar o método `mostrarDetalhes` para:

```
void mostrarDetalhes() {  
    System.out.println("Mostrando detalhes do livro ");  
    System.out.println("Nome: " + nome);  
    System.out.println("Descrição: " + descricao);  
    System.out.println("Valor: " + valor);  
    System.out.println("ISBN: " + isbn);  
    System.out.println("--");  
}
```

Execute agora o `main` da classe `CadastroDeLivros`. Sem precisar mudar nenhuma linha de seu código, o resultado será:

```
Mostrando detalhes do livro  
Nome: Java 8 Prático  
Descrição: Novos recursos da linguagem  
Valor: 59.9  
ISBN: 978-85-66250-46-6  
--  
Mostrando detalhes do livro  
Nome: Lógica de Programação  
Descrição: Crie seus primeiros programas  
Valor: 59.9  
ISBN: 978-85-66250-22-0  
--
```

Nosso código agora tem uma manutenibilidade muito maior! Sempre devemos criar métodos de forma genérica e **reaproveitável**, assim será muito

mais fácil e produtivo evoluir o código no futuro.

3.3 OBJETOS PARA TODOS OS LADOS!

Falando em evoluir código, precisamos saber mais informações sobre nossos livros. Por exemplo, quem escreveu o livro? Qual o e-mail do autor? E quando foi a sua data de publicação? Todas essas informações são relevantes para nossa livraria e também para nossos clientes. Podemos adicionar essas e outras informações na classe `Livro`:

```
public class Livro {

    String nome;
    String descricao;
    double valor;
    String isbn;
    String nomeDoAutor;
    String emailDoAutor;
    String cpfDoAutor;

    void mostrarDetalhes() {
        System.out.println("Mostrando detalhes do livro ");
        System.out.println("Nome: " + nome);
        System.out.println("Descrição: " + descricao);
        System.out.println("Valor: " + valor);
        System.out.println("ISBN: " + isbn);
        System.out.println("---");
    }
}
```

Mas repare que todas essas novas informações pertencem ao **autor** do livro e não necessariamente ao livro. Se `autor` é um elemento importante para nosso sistema, ele pode e deve ser representado como um objeto! Vamos fazer essa alteração, basta criar a classe `Autor` e declarar seus atributos:

```
public class Autor {

    String nome;
```

```
    String email;
    String cpf;
}
```

Portanto, podemos adicionar na classe `Livro` um atributo do tipo `Autor`, que acabamos de criar. Uma classe pode ter outra classe como atributo, esse é um processo natural conhecido como composição. Nossa código fica assim:

```
public class Livro {

    String nome;
    String descricao;
    double valor;
    String isbn;
    Autor autor;

    void mostrarDetalhes() {
        System.out.println("Mostrando detalhes do livro ");
        System.out.println("Nome: " + nome);
        System.out.println("Descrição: " + descricao);
        System.out.println("Valor: " + valor);
        System.out.println("ISBN: " + isbn);
        System.out.println("--");
    }
}
```

Vamos agora criar alguns autores no `CadastroDeLivros` e associar ao seu devido livro. Uma forma de fazer isso seria:

```
Autor autor = new Autor();
autor.nome = "Rodrigo Turini";
autor.email = "rodrigo.turini@caelum.com.br";
autor.cpf = "123.456.789.10";
```

Nesse código, apenas criamos o `autor` e populamos os seus atributos. Agora precisamos associar esse objeto ao seu livro. Podemos simplesmente fazer:

```
livro.autor = autor;
```

Vamos fazer isso com os dois livros, nosso código completo deve ficar parecido com:

```
public class CadastroDeLivros {  
  
    public static void main(String[] args) {  
  
        Autor autor = new Autor();  
        autor.nome = "Rodrigo Turini";  
        autor.email = "rodrigo.turini@caelum.com.br";  
        autor.cpf = "123.456.789.10";  
  
        Livro livro = new Livro();  
        livro.nome = "Java 8 Prático";  
        livro.descricao = "Novos recursos da linguagem";  
        livro.valor = 59.90;  
        livro.isbn = "978-85-66250-46-6";  
  
        livro.autor = autor;  
  
        livro.mostrarDetalhes();  
  
        Autor outroAutor = new Autor();  
        outroAutor.nome = "Paulo Silveira";  
        outroAutor.email = "paulo.silveira@caelum.com.br";  
        outroAutor.cpf = "123.456.789.10";  
  
        Livro outroLivro = new Livro();  
        outroLivro.nome = "Lógica de Programação";  
        outroLivro.descricao = "Crie seus primeiros programas";  
        outroLivro.valor = 59.90;  
        outroLivro.isbn = "978-85-66250-22-0";  
  
        outroLivro.autor = outroAutor;  
  
        outroLivro.mostrarDetalhes();  
    }  
}
```

Referência a objetos

É fundamental perceber que, quando instanciamos um novo objeto com a palavra reservada `new`, um `Autor` por exemplo, guardamos em sua variável uma referência para esse objeto, e não seus valores. Ou seja, a variável `autor` não guarda o valor de um `nome`, `email` e outros atributos da classe `Autor`, mas sim uma forma de acessar esses atributos do `autor` em memória. Muito diferente de quando trabalhamos com tipos primitivos que guardam uma **cópia** do valor.

Ainda não ficou claro? Observe o seguinte exemplo:

```
public class ComparandoReferencias {  
  
    public static void main(String[] args) {  
  
        Autor autor = new Autor();  
        autor.nome = "Rodrigo Turini";  
        autor.email = "rodrigo.turini@caelum.com.br";  
        autor.cpf = "123.456.789.10";  
  
        Autor autor2 = new Autor();  
        autor2.nome = "Rodrigo Turini";  
        autor2.email = "rodrigo.turini@caelum.com.br";  
        autor2.cpf = "123.456.789.10";  
  
    }  
}
```

Aqui estamos criando dois autores com as mesmas informações, mas o que acontece se eu compará-los da seguinte forma?

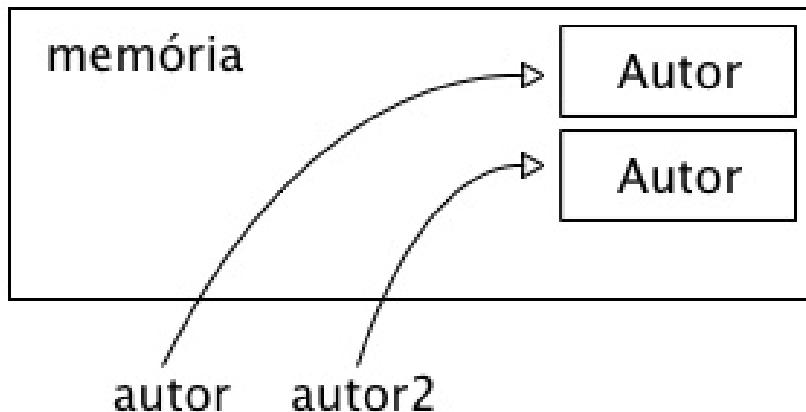
```
if (autor == autor2) {  
    System.out.println("Iguais, mesmo autor!");  
} else {  
    System.out.println("hein!? Por que diferentes?");  
}
```

Rode o código para conferir o resultado! A saída será:

```
hein!? Por que diferentes?
```

A resposta é simples, quando comparamos utilizando o `==` estamos comparando o valor da variável, que **sempre será a referência (endereço) para onde encontrar o objeto na memória**. E ,claro, cada objeto que criamos fica em um endereço diferente na memória.

A imagem a seguir representa os objetos em memória e cada variável de referência:



Que ir além? Observe esse código:

```

Autor autor = new Autor();
autor.nome = "Rodrigo Turini";

Livro livro = new Livro();
livro.autor = autor;

livro.autor.nome = "Guilherme Silveira";

System.out.println(autor.nome);

```

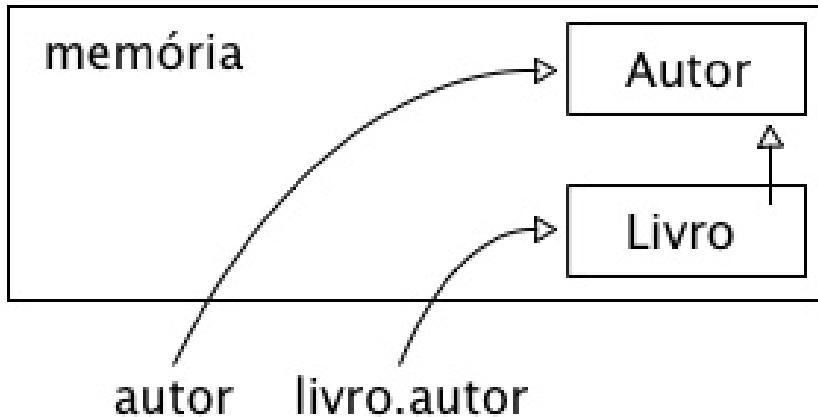
Estamos criando um autor e adicionando o nome Rodrigo Turini. Depois disso, criamos um `livro` e o associamos a esse `autor`. Mas repare que em seguida mudamos `livro.autor.nome` para Guilherme

Silveira. Qual será a saída ao imprimir o nome do autor original (`autor.nome`)?

Execute o código para ver a saída. Se você estava esperando Rodrigo Turini, não fique desapontado.

Acontece que, depois de associar a referência de `autor` ao `livro`, estamos atribuindo ao atributo `livro.autor` o mesmo endereço (referência) de memória que tem o `autor`. Ou seja, das duas formas a seguir estamos acessando o mesmo objeto:

```
autor.nome = "Rodrigo Turini";  
livro.autor.nome = "Guilherme Silveira";
```



Achou complicado? Não se preocupe, durante a leitura faremos muitas atribuições e relacionamentos entre objetos. Afinal, com certeza isso fará parte de seu dia a dia.

Mais e mais métodos

Nosso código evoluiu, agora todo `Livro` pode ter um `Autor`. Podemos evoluir nosso método `mostrarDetalhes` para que ele passe a mostrar as informações do `autor` de cada livro. Uma forma de fazer isso seria:

```

void mostrarDetalhes() {
    System.out.println("Mostrando detalhes do livro ");
    System.out.println("Nome: " + nome);
    System.out.println("Descrição: " + descricao);
    System.out.println("Valor: " + valor);
    System.out.println("ISBN: " + isbn);

    System.out.println("Mostrando detalhes do autor ");
    System.out.println("Nome: " + autor.nome);
    System.out.println("Email: " + autor.email);
    System.out.println("CPF: " + autor.cpf);
    System.out.println("--");
}

```

Isso funcionaria bem, mas ainda não é a solução ideal. O problema desse código é que a classe `Livro` está fazendo mais do que é sua responsabilidade. Mostrar as informações de `autor` deveria ser um comportamento da classe `Autor`! E, além disso, o que aconteceria caso nosso próximo objetivo fosse mostrar apenas os dados de um autor, sem as informações de seu livro? Teríamos que repetir todas essas ultimas linhas de código! Veja:

```

System.out.println("Mostrando detalhes do autor ");
System.out.println("Nome: " + autor.nome);
System.out.println("Email: " + autor.email);
System.out.println("CPF: " + autor.cpf);

```

Para evitar isso, podemos isolar esse comportamento na classe `Autor`! Basta criar um método `mostrarDetalhes` em seu modelo também:

```

public class Autor {

    String nome;
    String email;
    String cpf;

    void mostrarDetalhes() {
        System.out.println("Mostrando detalhes do autor ");
        System.out.println("Nome: " + nome);
        System.out.println("Email: " + email);
    }
}

```

```
        System.out.println("CPF: " + cpf);
    }
}
```

Pronto! Mais uma vez evitamos repetição de código usando bem a orientação a objetos. Toda classe pode e deve, além de seus atributos, ter comportamentos bem definidos, isolando suas regras de negócio. Podemos agora remover essas linhas repetidas do método `mostrarDetalhes` da classe `Livro` e apenas delegar a chamada desse novo método:

```
void mostrarDetalhes() {
    System.out.println("Mostrando detalhes do livro ");
    System.out.println("Nome: " + nome);
    System.out.println("Descrição: " + descricao);
    System.out.println("Valor: " + valor);
    System.out.println("ISBN: " + isbn);
    autor.mostrarDetalhes();
    System.out.println("--");
}
```

Depois de fazer essas mudanças, vamos executar mais uma vez a classe `CadastroDeLivros`. A saída será:

```
Mostrando detalhes do livro
Nome: Java 8 Prático
Descrição: Novos recursos da linguagem
Valor: 59.9
ISBN: 978-85-66250-46-6
Mostrando detalhes do autor
Nome: Rodrigo Turini
Email: rodrigo.turini@caelum.com.br
CPF: 123.456.789.10
--
Mostrando detalhes do livro
Nome: Lógica de Programação
Descrição: Crie seus primeiros programas
Valor: 59.9
ISBN: 978-85-66250-22-0
Mostrando detalhes do autor
```

Nome: Paulo Silveira
Email: paulo.silveira@caelum.com.br
CPF: 123.456.789.10
--

Métodos com parâmetro

Depois de instanciado, um objeto do tipo `Livro` pode precisar ter seu valor ajustado. É comum aplicar um desconto no preço de um livro, por exemplo. Podemos escrever o seguinte código para aplicar 10% de desconto no valor de um livro:

```
public static void main(String[] args) {  
  
    Livro livro = new Livro();  
    livro.valor = 59.90;  
  
    System.out.println("Valor atual: " + livro.valor);  
  
    livro.valor -= livro.valor * 0.1;  
  
    System.out.println("Valor com desconto: " + livro.valor);  
}
```

Note que a linha abaixo está calculando a porcentagem de desconto e já subtraindo do valor do livro.

```
livro.valor -= livro.valor * 0.1;
```

Executando o código, teremos a saída esperada, que é:

```
Valor atual: 59.9  
Valor com desconto: 53.91
```

Mas esse código não está nem um pouco orientado a objetos, estamos escrevendo um comportamento do livro direto no método `main`! Já vimos qual o problema em fazer isso, imagine que existam 100 livros e eu aplique desconto em todos eles. A nossa lógica de aplicar desconto vai ficar repetida e espalhada por todo o código.

Como evitar isso? Se aplicar um desconto em seu valor é um comportamento do livro, podemos isolar esse comportamento em um novo método! No lugar de fazer:

```
livro.valor -= livro.valor * 0.1;
```

Podemos criar um método `aplicaDescontoDe` que recebe um valor como parâmetro. Repare:

```
livro.aplicaDescontoDe(0.1);
```

Sim, um método pode e deve receber parâmetros sempre que for preciso. Veja como fica a sua implementação:

```
public void aplicaDescontoDe(double porcentagem) {  
    valor -= valor * porcentagem;  
}
```

Note que todo parâmetro de método também precisa ter um nome e tipo definido. A `porcentagem` receberá o valor `0.1` (`double`) que foi passado como argumento no momento em que o método foi invocado em nosso método `main`.

NOMES AMBÍGUOS E O `THIS`

O que aconteceria se o nome do parâmetro do método fosse igual ao nome do atributo da classe? Veja como ficaria o nosso código:

```
public void aplicaDescontoDe(double valor) {  
    valor -= valor * valor;  
}
```

No lugar de uma `porcentagem`, chamamos o parâmetro de `valor`, assim como o nome do atributo da classe `Livro`. Como o Java vai saber qual valor queremos atualizar? A resposta é: ele não vai. Nosso código vai subtrair e multiplicar o valor do parâmetro por ele mesmo, já que este tem um escopo menor que o atributo da classe. Ou seja, mesmo com a ambiguidade neste caso o código vai compilar, mas o `valor` considerado será o que possui o menor escopo.

Para evitar esse problema, podemos utilizar a palavra reservada `this` para mostrar que esse é um atributo da classe. Ainda que seja opcional, é sempre uma boa prática usar o `this` em atributos para evitar futuros problemas de ambiguidade e também para deixar claro que este é um atributo da classe, e não uma simples variável.

Com isso, o código de nosso método `aplicaDescontoDe` fica assim:

```
public void aplicaDescontoDe(double porcentagem) {  
    this.valor -= this.valor * porcentagem;  
}
```

E a classe `Livro` completa:

```
public class Livro {  
  
    String nome;  
    String descricao;  
    double valor;  
    String isbn;  
    Autor autor;
```

```
void mostrarDetalhes() {
    System.out.println("Mostrando detalhes do livro ");
    System.out.println("Nome: " + nome);
    System.out.println("Descrição: " + descricao);
    System.out.println("Valor: " + valor);
    System.out.println("ISBN: " + isbn);
    autor.mostrarDetalhes();
    System.out.println("--");
}

public void aplicaDescontoDe(double porcentagem) {
    this.valor -= this.valor * porcentagem;
}
}
```

Métodos com retorno

Nossa classe `Livro` já possui dois métodos, mas os dois são `void`. Vimos que o `void` representa a ausência de um retorno, mas há situações em que precisaremos retornar algo. Por exemplo, como saber se um livro tem ou não um autor? Um alternativa seria criar um método `temAutor`, tendo um `boolean` como retorno. Observe um exemplo de uso:

```
if(livro.temAutor()) {
    System.out.print("O nome do autor desse livro é ");
    System.out.println(livro.autor.nome);
}
```

A implementação desse método é bem simples, ele não receberá nenhum parâmetro e terá o tipo de retorno `boolean`:

```
boolean temAutor(){
    // o que fazer aqui?
}
```

Mas como saber se o `autor` do livro existe ou não? Na verdade, é bem simples: quando um objeto não foi instanciado, ele não tem nenhuma referência, portanto seu valor será `null`. Sabendo disso, tudo o que precisamos

fazer no corpo do método é retornar um boolean, informando se o atributo autor é diferente de null. Para retornar um valor, utilizamos a palavra reservada return:

```
boolean temAutor(){  
    boolean naoEhNull = this.autor != null;  
    return naoEhNull;  
}
```

Podemos fazer isso de uma forma ainda mais simples, retornando diretamente a expressão booleana sem a criação da variável naoEhNull:

```
boolean temAutor(){  
    return this.autor != null;  
}
```

Agora que o Livro ganhou esse novo comportamento, vamos tirar proveito dele dentro do método mostrarDetalhes da própria classe. Podemos mostrar os detalhes do autor apenas quando ele existir. Uma forma de fazer isso seria:

```
void mostrarDetalhes() {  
    System.out.println("Mostrando detalhes do livro ");  
    System.out.println("Nome: " + nome);  
    System.out.println("Descrição: " + descricao);  
    System.out.println("Valor: " + valor);  
    System.out.println("ISBN: " + isbn);  
  
    if (this.temAutor()) {  
        autor.mostrarDetalhes();  
    }  
  
    System.out.println("--");  
}
```

Pronto! Ao imprimir os detalhes de um livro que não tem autor, apenas os dados do livro serão exibidos.

3.4 ENTENDENDO A CONSTRUÇÃO DE UM OBJETO

Agora que já conhecemos um pouco sobre objetos, podemos entender melhor como funciona seu processo de construção. Repare na sintaxe utilizada para criar um novo `Livro`:

```
Livro livro = new Livro();
```

Quando escrevemos a instrução `Livro()` seguida da palavra reservada `new`, estamos pedindo para a JVM procurar a classe `Livro` e invocar o seu construtor, que se parece com:

```
public Livro(){  
}
```

Um construtor é bastante parecido com um método comum, mas ele não é um. Diferente dos métodos, um construtor tem o mesmo nome da classe e não tem um retorno declarado.

Mas, se nunca escrevemos esse construtor, quem o fez? Sempre que você não criar um construtor para suas classes, o compilador fará isso para você.

Quer uma prova? Vamos utilizar o programa `javap` que já vem no JDK para ver o código compilado. Acesse pelo terminal do seu sistema operacional a pasta `bin` de seu projeto e rode o comando `javap Livro`. A saída deverá se parecer com:

```
Turini@ ~/Documents/workspace/livro-oo > javap Livro  
Compiled from "Livro.java"  
public class Livro {  
    java.lang.String nome;  
    java.lang.String descricao;  
    double valor;  
    java.lang.String isbn;  
    public Livro();  
    void mostrarDetalhes();  
    public void aplicaDescontoDe(double);  
}
```

Note que, mesmo sem criar o construtor vazio, ele está presente em nosso código compilado:

```
public Livro();
```

E, sim, todo código que estiver declarado dentro do construtor será executado quando um objeto desse tipo for criado. Por exemplo, o código a seguir imprime uma mensagem sempre que criarmos um novo `Livro`:

```
public class Livro {  
  
    String nome;  
    String descricao;  
    double valor;  
    String isbn;  
    Autor autor;  
  
    public Livro() {  
        System.out.println("novo livro criado");  
    }  
  
    // demais métodos da classe  
}
```

Para testar, podemos criar alguns livros dentro de um método `main` e executar esse código:

```
public static void main(String[] args) {  
    Livro livro1 = new Livro();  
    Livro livro2 = new Livro();  
    Livro livro3 = new Livro();  
    Livro livro4 = new Livro();  
}
```

Como já é esperado, a saída será:

```
novo livro criado  
novo livro criado  
novo livro criado  
novo livro criado
```

Agora que já temos um construtor, o compilador não vai criar mais nenhum. Ele só faz isso quando a sua classe não tem nenhum construtor definido.

Veremos mais adiante que construtores também podem receber parâmetros e inicializar os atributos de suas classes.

3.5 VANTAGENS DA ORIENTAÇÃO A OBJETOS

Vimos neste capítulo apenas algumas das muitas vantagens de se trabalhar com orientação a objetos. Observe com atenção o código que criamos. Você vai conseguir perceber que uma das grandes diferenças da OO é que temos uma forma forte de criar conexão entre informações e funcionalidades. Além disso, nosso código fica muito mais organizado e evitamos muita repetição. O que ganhamos? Um código mais flexível e, com isso, mais fácil de evoluir e manter.

Esse é só o começo, no decorrer do livro veremos vários conceitos e recursos como *encapsulamento*, *herança*, *polimorfismo* e muitos outros que tornam o paradigma ainda mais interessante.

CAPÍTULO 4

Encapsulamento

4.1 LIMITANDO DESCONTO DO LIVRO

Quando criamos o método `aplicaDescontoDe` na classe `Livro`, não determinamos a porcentagem total de desconto que um livro pode ter, mas isso é muito importante para nossa regra de negócio. Um livro pode ter no máximo 30% de desconto.

Uma forma simples de evoluir nosso método seria:

```
public boolean aplicaDescontoDe(double porcentagem) {  
    if (porcentagem > 0.3) {  
        return false;  
    }  
    this.valor -= this.valor * porcentagem;  
    return true;  
}
```

Rpare que agora estamos retornando um boolean para indicar ao usuário se o desconto foi aplicado ou não. No momento de usá-lo podemos fazer algo como:

```
if (!livro.aplicaDescontoDe(0.1)) {
    System.out.println("Desconto não pode ser maior do que 30%");
}
```

CÓDIGO FLEXÍVEL E REAPROVEITÁVEL

O método `aplicaDescontoDe` poderia, sim, no lugar de retornar um boolean já imprimir a mensagem de validação do limite de desconto, como por exemplo:

```
public void aplicaDescontoDe(double porcentagem) {
    if (porcentagem > 0.3) {
        System.out.println("Desconto não pode ser
                           maior do que 30%");
    }
    this.valor -= this.valor * porcentagem;
}
```

Com isso, não seria necessário escrever o `if` em nosso método `main`, mas repare que estamos perdendo a flexibilidade de customizar nossas mensagens. Será que todos que forem utilizar esse método querem mostrar a mensagem *Desconto não pode ser maior do que 30%*? E se eu quiser mudar esse texto dependendo do usuário ou simplesmente não mostrar a mensagem de validação em algum momento?

Lembre-se sempre de evitar deixar informações tão específicas em seus moldes, além de não sobrecarregá-los com comportamentos que não deveriam ser de sua responsabilidade.

Pronto! Nossa lógica já está bem definida, agora um livro não pode ter um desconto maior do que 30%. Mas vamos testá-la para ter certeza. Para isso, basta criar a classe `RegrasDeDesconto` com o seguinte cenário:

```
public class RegrasDeDesconto {
```

```
public static void main(String[] args) {  
  
    Livro livro = new Livro();  
    livro.valor = 59.90;  
  
    System.out.println("Valor atual: " + livro.valor);  
  
    if (!livro.aplicaDescontoDe(0.1)){  
        System.out.println("Desconto não pode ser  
        maior do que 30%");  
    } else {  
        System.out.println("Valor com desconto: "  
            + livro.valor);  
    }  
}  
}
```

Ao executá-la, teremos o resultado:

```
Valor atual: 59.9  
Valor com desconto: 53.91
```

E, mudando o valor do desconto para 0.4, a saída será:

```
Valor atual: 59.9  
Desconto não pode ser maior do que 30%
```

Excelente! É exatamente o comportamento que estamos esperando. Mas há um problema grave. Nada obriga o desenvolvedor a utilizar o método `aplicaDescontoDe`, ele poderia perfeitamente escrever o código desta forma:

```
public class RegrasDeDesconto {  
  
    public static void main(String[] args) {  
  
        Livro livro = new Livro();  
        livro.valor = 59.90;  
  
        System.out.println("Valor atual: " + livro.valor);  
    }  
}
```

```
        livro.valor -= livro.valor * 0.4;
        System.out.println("Valor com desconto: " + livro.valor);
    }
}
```

Ele consegue aplicar o desconto de 40% ou mais, independente de nossa regra de negócio.

O problema é que o atributo `valor` da classe `Livro` pode ser acessado diretamente e modificado pelas outras lógicas do nosso sistema. Isso é muito ruim! Como vimos, ao expor nossos dados, estamos abrindo caminho para que os objetos possam ser modificados independente das condições verificadas em seus métodos.

Apesar de ser um problema muito sério, a solução é bastante simples. Basta modificar a visibilidade do atributo `valor`, que até então é `default` (padrão). Podemos restringir o acesso para esse atributo para que fique acessível apenas pela própria classe `Livro`. Repare:

```
public class Livro {

    private double valor;

    // outros atributos e métodos
}
```

Como estamos utilizando o modificador de visibilidade `private`, ninguém mais além da própria classe `Livro` conseguirá acessar e modificar esse valor. Portanto, a seguinte linha da classe `RegrasDeDesconto` não irá compilar.

```
livro.valor -= livro.valor * 0.4;
```

O erro será `The field Livro.valor is not visible`. Ótimo, isso fará com que a única forma de se aplicar um desconto em um livro seja passando pela nossa regra de negócio, que está isolada no método `aplicaDescontoDe`.

MODIFICADORES DE VISIBILIDADE

Além do `private` e `default` (padrão), existem outros modificadores de visibilidade. São eles: `public` e `protected`. No momento certo entenderemos a fundo cada um deles, mas tenha em mente desde já que todos eles são diferentes e possuem regras bem específicas.

4.2 ISOLANDO COMPORTAMENTOS

Ao alterar a visibilidade do atributo `valor` para `private` todos os lugares que acessavam esse atributo passaram a não compilar, já que apenas a própria classe `Livro` pode fazer isso. Por exemplo, observe como estávamos passando o valor do livro:

```
livro.valor = 59.90;
```

Como esse acesso não pode mais ser feito, precisaremos criar um comportamento (método) na classe `Livro` para fazer a atribuição desse valor. Poderíamos chamá-lo de `adicionaValor`.

```
livro.adicionaValor(59.90);
```

Sua implementação será bem simples, observe:

```
void adicionaValor(double valor) {  
    this.valor = valor;  
}
```

Há outro erro de compilação que precisaremos resolver. Repare na forma como estamos imprimindo o valor do `Livro`:

```
System.out.println("Valor atual: " + livro.valor);
```

Bem, você já pode ter imaginado que a solução será parecida. Preciso criar um método `retornaValor` que não receberá nenhum argumento e retornará o atributo `double valor`. Algo como:

```
double retornaValor() {  
    return this.valor;  
}
```

E como esse método não recebe parâmetros, o uso do `this` será opcional. Agora podemos imprimir dessa forma:

```
System.out.println("Valor atual: " + livro.retornaValor());
```

Quando usar o `private`?

Isso resolveu o problema do `valor` do livro, mas e quanto aos demais atributos? Quando devemos deixar um atributo de classe com visibilidade `private`? Você deve estar pensando: “quando não quero que ninguém acesse esse atributo diretamente”

, e é verdade. Mas a questão é: quando eu quero que alguém acesse algum atributo de forma direta?

O `nome` do livro está com visibilidade `default`, portanto estamos adicionando da seguinte forma:

```
livro.nome = "Java 8 Prático";
```

Considere que depois de alguns meses apareça a necessidade de validar que a `String` passada possui ao menos duas letras, caso contrário não será um nome válido. Ou seja, nosso código precisará fazer algo como:

```
livro.nome = "Java 8 Prático";  
if(livro.nome.length() < 2) {  
    System.out.println("Nome inválido");  
}
```

Repare que o método `length` foi usado para retornar o tamanho de uma `String`. Além desse há diversos outros métodos úteis definidos na classe `String`, que conheceremos melhor adiante. O importante agora é perceber que, se acessamos o atributo `nome` diretamente em mil partes de nosso código, precisaremos replicar essa mudança em mil lugares. Nada fácil.

Já vimos que todo comportamento da classe `Livro` deveria ser isolado em um método, assim evitamos repetição de código possibilitando o reúso

desse método por toda a aplicação. O ideal seria desde o começo ter criado um método `adicionaNome` com esse comportamento.

Mas apenas criar o método `adicionaNome` não teria resolvido o problema, percebe? Se a visibilidade do atributo ainda fosse `default`, haveria duas formas de se adicionar um nome ao livro e nossa lógica de adicionar nome ainda poderia estar espalhada pela aplicação.

Como garantir que isso nunca aconteça? Usando `private` sempre! Todo atributo de classe deve ser privado, assim garantimos que ninguém os acesse diretamente e viole as nossas regras de negócio.

4.3 CÓDIGO ENCAPSULADO

Essa mudança que iniciamos em nossa classe faz parte de um conceito muito importante da OO conhecido como **encapsulamento**. A ideia é simplesmente esconder todos os atributos de suas classes (deixando-os `private`) e *encapsular* seus comportamentos em métodos. Além disso, encapsular é esconder como funcionam suas regras de negócio, os seus métodos.

Um bom termômetro para descobrir se o seu código está bem encapsulado é fazendo as duas perguntas:

- O que esse código faz?
- Como esse código faz?

Veja por exemplo quando aplicamos o desconto acessando o atributo diretamente:

```
livro.valor -= livro.valor * 0.4;
```

Olhando para esse código, nós conseguimos responder as duas perguntas: o que e como é feito, mas **em um código bem encapsulado você só deveria conseguir responder a primeira**.

Observe este segundo exemplo:

```
livro.aplicaDescontoDe(0.1);
```

Repare que fica claro o que o método `aplicaDescontoDe` está fazendo, mas apenas olhando para ele não conseguimos responder **como** isto será feito. Se em algum momento no futuro resolvemos mandar um e-mail sempre que algum desconto for aplicado, não precisaremos mudar milhares de partes de nosso código, mas sim esse único método que está bem encapsulado. Portanto, o **encapsulamento deixa nosso código muito mais passível de mudanças**.

INTERFACE DA CLASSE

Os métodos públicos existentes em sua classe são comumente chamados de *interface da classe*, afinal em um código encapsulado eles são a única maneira de você interagir com os objetos dessa classe.

Pensar em algum contexto do mundo real pode ajudar a entender encapsulamento e interface da classe. Por exemplo, quando você vai escrever um texto em um computador, o que importa é o teclado (a interface que você usa para fazer isso). Pouco importa **como** ele funciona internamente. Quando você mudar de computador ou simplesmente de tipo de teclado, não precisará reaprender a usar um teclado.

Por mais diferente que eles funcionem internamente, isso não fará diferença para um usuário final. Ninguém precisa desmontar a lateral de um computador e manualmente mudar os circuitos para que o computador possa interpretar os sinais enviados e mostrar os dados na tela, afinal esse comportamento está muito bem encapsulado na interface do teclado.

4.4 GETTERS E SETTERS

Se todos os atributos de nossas classes forem `private`, precisaremos criar um método sempre que quisermos que alguém consiga adicionar um valor ao atributo e o mesmo quando quisermos que alguém consiga ler e exibir este valor. Assim como fizemos com os métodos `adicionaValor` e `retornaValor`.

A convenção de nome para esses dois métodos é utilizar o prefixo `get`

(pegar) e `set` (atribuir). Por exemplo, no lugar de chamar os métodos de `adicionaValor` e `retornaValor`, podemos chamá-los de `setValor` e `getValor`, respectivamente.

Nosso código fica assim:

```
public class Livro {  
  
    private double valor;  
  
    // demais atributos e métodos  
  
    public double getValor() {  
        return valor;  
    }  
  
    public void setValor(double valor) {  
        this.valor = valor;  
    }  
}
```

Repare que o método `get` não recebe nenhum parâmetro e apenas retorna o atributo, enquanto o `set` sempre recebe um parâmetro com o mesmo tipo do que o atributo que será atualizado.

ECLIPSE: GGAS E AUTOCOMPLETE

Criar `getters` e `setters` pode parecer um processo trabalhoso, mas, por ser um padrão de código muito comum, as principais *IDEs* do mercado possuem atalhos e templates que nos ajudam a fazer isso.

No Eclipse, você pode fazer isso de forma muito produtiva utilizando o atalho `Control + 3` e depois digitando `ggas`. Repare que essas são as iniciais do comando `Generate Getters and Setters`. Selecione esta opção você poderá escolher para quais atributos criar os métodos e clicar em `Finish`.

Tudo pronto, o Eclipse gerou o código todo pra você!

Os getters e setters possibilitam o acesso dos atributos de forma controlada. Mas é muito importante perceber que nem todo atributo deve ter um getter e setter, você só deve criar esses métodos quando houver uma real necessidade. Há um post do Paulo Silveira muito interessante ilustrando essa situação:

<http://blog.caelum.com.br/2006/09/14/nao-aprender-oo-getters-e-setters/>

Depois de criar os getters e setters necessários, nossa classe Livro deve ficar assim:

```
public class Livro {  
  
    private String nome;  
    private String descricao;  
    private double valor;  
    private String isbn;  
    private Autor autor;  
  
    // demais métodos omitidos  
  
    public double getValor() {  
        return valor;  
    }  
  
    public void setValor(double valor) {  
        this.valor = valor;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    public String getDescricao() {  
        return descricao;  
    }  
}
```

```
public void setDescricao(String descricao) {
    this.descricao = descricao;
}

public String getIsbn() {
    return isbn;
}

public void setIsbn(String isbn) {
    this.isbn = isbn;
}

public Autor getAutor() {
    return autor;
}

public void setAutor(Autor autor) {
    this.autor = autor;
}
}
```

Precisamos fazer o mesmo para nossa classe `Autor`:

```
public class Autor {

    private String nome;
    private String email;
    private String cpf;

    // demais métodos omitidos

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

```
public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getCpf() {
    return cpf;
}

public void setCpf(String cpf) {
    this.cpf = cpf;
}

}
```

E, por fim, será necessário modificar a classe `CadastroDeLivros` para que deixe de acessar os atributos diretamente. Seu código final deve ficar parecido com:

```
public class CadastroDeLivros {

    public static void main(String[] args) {

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

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

        livro.setAutor(autor);
    }
}
```

```
livro.mostrarDetalhes();

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

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

outroLivro.setAutor(outroAutor);

outroLivro.mostrarDetalhes();
}

}
```

4.5 DEFININDO DEPENDÊNCIAS PELO CONSTRUTOR

Autor é um atributo obrigatório

Agora que nosso código já está bem encapsulado, precisamos atender outra necessidade de nossa livraria. Todo `Livro` precisa ter um `Autor`. Isso é bem simples: assim como já está sendo feito, logo após criar um livro, podemos associá-lo ao seu autor pelo método `setAutor`:

```
Autor autor = new Autor();
autor.setNome("Rodrigo Turini");
// set dos outros atributos

Livro livro = new Livro();
// set dos outros atributos

livro.setAutor(autor);
```

Isso já resolve o problema, mas há uma questão. Dessa forma, por alguns instantes o `Livro` existe em memória sem ter um `Autor` relacionado. E

esse nem é o pior dos problemas, o que acontecerá se esquecermos de chamar o método `setAutor`? A resposta é: nada. Ou seja, da forma como nosso código está escrito, um livro pode existir sem seu autor.

Como resolver isso? Bem, como já vimos, toda classe tem um construtor. Ainda que eu não tenha declarado, o compilador fará isso para mim. O que ainda não vimos é que construtores podem receber parâmetros. Com isso, podemos obrigar a passagem de alguns valores logo no momento de criação de nossos objetos.

Vamos criar um construtor na classe `Livro` que deve receber um `Autor` como parâmetro. Seu código deve ficar assim:

```
public Livro(Autor autor) {  
    this.autor = autor;  
}
```

Repare que ele ficou com a responsabilidade do método `setAutor`, que é atribuir o `autor` do parâmetro ao atributo da classe.

A partir do momento em que criamos esse construtor com parâmetro na classe `Livro`, o compilador não criará mais o construtor *default* (vazio). Portanto, o seguinte código não compila:

```
Livro livro = new Livro();
```

Se o único construtor existente na classe recebe um `Autor`, a única forma de fazer esse código compilar e criar um `Livro` será passando um `Autor` como argumento. Veja:

```
Autor autor = new Autor();  
autor.setNome("Rodrigo Turini");  
// set dos outros atributos
```

```
Livro livro = new Livro(autor);  
// set dos outros atributos
```

Claro, há situações em que queremos deixar as duas formas válidas, ou seja, queremos criar um `Livro` passando ou não um `Autor`. Para que isso funcionasse, precisaríamos adicionar na classe `Livro` um novo construtor que não recebe nenhum argumento. Seu código ficaria assim:

```
public class Livro {  
  
    private String nome;  
    private String descricao;  
    private double valor;  
    private String isbn;  
    private Autor autor;  
  
    public Livro(Autor autor) {  
        this.autor = autor;  
    }  
  
    public Livro() {  
    }  
  
    // setters, getters e outros métodos  
}
```

Uma classe pode, sim, ter mais de um construtor, isso é conhecido como uma sobrecarga (*overloaded*) de construtor. Mas, assim como nos demais métodos, isso funcionará contanto que os construtores não tenham a mesma quantidade de parâmetro. Por exemplo, se eu tentar fazer:

```
public Livro() {  
}  
  
public Livro() {  
    System.out.println("novo livro criado");  
}
```

Esse código não compilará, afinal, quando alguém fizer `new Livro()` qual dos construtores seria chamado?

Como nossa regra de negócio exige um `Autor` para cada `Livro`, vamos manter apenas o construtor com parâmetro. Para isso, precisaremos mudar as nossas classes que criam um novo `Livro` para atender essa condição. A classe `CadastroDeLivros` deve ficar assim:

```
public class CadastroDeLivros {
```

```
public static void main(String[] args) {  
  
    Autor autor = new Autor();  
    autor.setNome("Rodrigo Turini");  
    autor.setEmail("rodrigo.turini@caelum.com.br");  
    autor.setCpf("123.456.789.10");  
  
    Livro livro = new Livro(autor);  
    livro.setNome("Java 8 Prático");  
    livro.setDescricao("Novos recursos da linguagem");  
    livro.setValor(59.90);  
    livro.setIsbn("978-85-66250-46-6");  
  
    livro.mostrarDetalhes();  
  
    Autor outroAutor = new Autor();  
    outroAutor.setNome("Paulo Silveira");  
    outroAutor.setEmail("paulo.silveira@caelum.com.br");  
    outroAutor.setCpf("123.456.789.10");  
  
    Livro outroLivro = new Livro(outroAutor);  
    outroLivro.setNome("Lógica de Programação");  
    outroLivro.setDescricao("Crie seus primeiros programas");  
    outroLivro.setValor(59.90);  
    outroLivro.setIsbn("978-85-66250-22-0");  
  
    outroLivro.mostrarDetalhes();  
}  
}
```

MANTEMOS OU NÃO O MÉTODO SETAUTOR?

Se o construtor já atribuiu o `Autor` do livro, precisamos manter um `setter` para esse atributo? Isso vai depender de nossa necessidade, claro. Pode ser que faça bastante sentido para nossa livraria modificar o autor de um livro depois que ele foi criado. Mas fique atento! Você só deve manter o `setter` se ele realmente for necessário, lembre-se da importância de manter suas rotinas *encapsuladas*.

Inicializando atributos da classe

Por fim, outro ponto que precisamos resolver em nossa livraria é relacionado ao número de *ISBN* de nossos livros. Em nosso `CadastroDeLivros`, sempre que criamos um `Livro`, já atribuímos seu número de *ISBN*, mas normalmente esse identificador pode demorar alguns dias para ficar pronto.

Repare qual será o resultado do método `mostrarDetalhes` quando o *ISBN* não estiver preenchido:

```
Mostrando detalhes do livro
Nome: Java 8 Prático
Descrição: Novos recursos da linguagem
Valor: 59.9
ISBN: null
Mostrando detalhes do autor
Nome: Rodrigo Turini
Email: rodrigo.turini@caelum.com.br
CPF: 123.456.789.10
```

O valor `null` será impresso. Isso acontece pois, diferente dos *tipos primitivos*, que sempre possuem um valor padrão (*default*), objetos como a `String` não possuem, portanto, antes de serem inicializadas terão sua referência igual a `null`.

Esse é o valor *default* de cada tipo primitivo:

Tipo Primitivo	Valor Default
boolean	false
byte	0
short	0
char	'\u0000'
int	0
float	0.0f
long	0l
double	0.0d

Outro uso bem comum de um `construtor` é para inicialização de atributos, como o caso do *ISBN*. Para fornecer um valor padrão para este atributo, poderíamos simplesmente inicializá-lo no `construtor` da classe:

```
public Livro(Autor autor) {
    this.autor = autor;
    this.isbn = "000-00-00000-00-0";
}
```

Nesse caso, ao `mostrarDetalhes` de um `Livro` sem *ISBN*, teremos a saída:

```
Mostrando detalhes do livro
Nome: Java 8 Prático
Descrição: Novos recursos da linguagem
Valor: 59.9
```

ISBN: 000-00-00000-00-0
Mostrando detalhes do autor
Nome: Rodrigo Turini
Email: rodrigo.turini@caelum.com.br
CPF: 123.456.789.10

DELEGANDO PARA OUTROS CONSTRUTORES

Repare que, se a classe `Livro` também tivesse um construtor sem argumentos, como a seguir:

```
public Livro(Autor autor) {  
    this.autor = autor;  
    this.isbn = "000-00-00000-00-0";  
}  
  
public Livro() {  
}
```

O valor de `ISBN` só seria inicializado quando o construtor com um `Autor` fosse chamado. Para resolver isso, você pode encadear a chamada dos construtores utilizando a palavra reservada `this`, como a seguir:

```
public Livro(Autor autor) {  
    this();  
    this.autor = autor;  
}  
  
public Livro() {  
    this.isbn = "000-00-00000-00-0";  
}
```


CAPÍTULO 5

Herança e polimorfismo

5.1 TRABALHANDO COM LIVROS DIGITAIS

Por causa da sua praticidade e fácil portabilidade, muitos leitores preferem livros digitais (*e-books*) aos livros físicos. Nossa livraria não pode ficar para trás, portanto, passaremos a trabalhar com esse novo tipo de livro.

Mas, afinal, como podemos diferenciar um `Livro` impresso de um `Ebook`? Uma forma simples de fazer isso seria adicionando um atributo que define o tipo de livro. Poderia simplesmente ser um `boolean`, marcando se um livro é impresso ou não. Repare:

```
public class Livro {  
  
    private String nome;  
    private String descricao;  
    private double valor;
```

```
private String isbn;
private Autor autor;
private boolean impresso;

public Livro(Autor autor) {
    this.autor = autor;
    this.isbn = "000-00-00000-00-0";
    this.impresso = true;
}

// outros métodos da classe
}
```

Note que em nosso construtor já definimos um valor padrão para o atributo `impresso`. Todo `Livro` que não tenha o valor de `impresso` definido será considerado um livro impresso.

Essa é uma forma simples de resolver o problema e talvez até possa atender bem as nossas necessidades, mas o problema dessa abordagem é que um `Ebook` tem alguns comportamentos bastante diferentes do que um livro impresso. Por exemplo, o método `aplicaDescontoDe` da classe `Livro` atualmente limita a porcentagem de desconto em 30%.

```
public boolean aplicaDescontoDe(double porcentagem) {
    if (porcentagem > 0.3) {
        return false;
    }
    this.valor -= this.valor * porcentagem;
    return true;
}
```

Mas quando se trata de um `Ebook`, a regra é um pouco diferente: podemos aplicar no máximo 15% de desconto. Podemos resolver esse problema adicionando mais uma condição ao método:

```
public boolean aplicaDescontoDe(double porcentagem) {
    if (porcentagem > 0.3) {
        return false;
    } else if (!this.impresso && porcentagem > 0.15) {
        return false;
    }
```

```
    }
    this.valor -= this.valor * porcentagem;
    return true;
}
```

Agora nosso código funciona como esperado, mas está um pouco mais verboso e difícil de entender. O grande problema aqui é que, a cada método cuja regra seja diferente, teremos que colocar um novo `if` parecido com esse, isso sem contar que novos tipos de `Livro` podem aparecer e tornar nosso código ainda mais complicado e cheio de condicionais. Devemos sempre escrever nosso código pensando em como será sua evolução no futuro.

Além disso, existem alguns comportamentos e atributos que só servem para um `Ebook`. Um deles é o `watermark` (marca d'água). Essa é a forma de identificar discretamente o nome e e-mail do dono daquele livro digital, normalmente no rodapé das páginas.

Se `Ebook` é um elemento importante, possui comportamentos e atributos específicos, ele deveria ser representado como um `Objeto`! Podemos criar uma classe `Ebook` definindo os atributos e comportamentos específicos desse novo tipo.

```
public class Ebook {

    private String nome;
    private String descricao;
    private double valor;
    private String isbn;
    private Autor autor;
    private String waterMark;

    public void setWaterMark(String waterMark) {
        this.waterMark = waterMark;
    }

    public String getWaterMark() {
        return waterMark;
    }
}
```

```
// getters, setters e outros métodos  
}
```

Nosso código já está um pouco mais interessante, afinal não estamos mais sobrecarregando a classe `Livro` com atributos e métodos que serão utilizados apenas quando o tipo do livro for um *ebook*. Mas há muito código repetido aqui: além dos comportamentos do `Ebook`, temos todos os atributos e métodos já escritos na classe `Livro`.

Para evitar toda essa repetição de código, podemos ensinar ao compilador que o `Ebook` é um tipo de `Livro`, ou seja, além de seus próprios atributos e métodos, essa classe possui tudo o que um `Livro` tem. Para fazer isto, basta `Ebook` dizer na declaração da classe que ela é um `Livro`, que é uma **extensão** dessa classe:

```
public class Ebook extends Livro {  
  
    private String waterMark;  
  
    public Ebook(Autor autor) {  
        super(autor);  
    }  
  
    public void setWaterMark(String waterMark) {  
        this.waterMark = waterMark;  
    }  
  
    public String getWaterMark() {  
        return waterMark;  
    }  
}
```

DELEGATE CONSTRUCTOR

Como a classe `Livro` tinha um construtor obrigando a passagem de um `Autor` como parâmetro, ao herdar de um `Livro`, a classe `Ebook` também herdou essa responsabilidade. Repare que utilizamos a palavra `super` para delegar a responsabilidade para a *superclasse* que já tem esse comportamento bem definido.

```
public Ebook(Autor autor) {  
    super(autor);  
}
```

Ao utilizar a palavra reservada `extends`, estamos dizendo que um `Ebook` (*subclasse*) **herda** tudo o que a classe `Livro` (*superclasse*) tem. Portanto, mesmo sem ter nenhum desses métodos declarados diretamente na classe `Ebook`, podemos executar o seguinte código sem nenhum problema:

```
Ebook ebook = new Ebook();  
ebok.setNome("Java 8 Prático");
```

Como a classe `Livro` tem os *setters* para o atributo `nome` declarados, um `Ebook` também terá.

HERANÇA MÚLTIPLA

Uma regra importante da herança em Java é que nossas classes só podem herdar diretamente de **uma** classe pai. Ou seja, não há herança múltipla como na linguagem C++. Mas sim, uma classe pode herdar de uma classe que herda de outra e assim por diante. Você pode encadear a herança de suas classes, no entanto, veremos mais à frente que essa estratégia não é muito interessante por aumentar de mais o **acoplamento** entre suas classes.

5.2 REESCREVENDO MÉTODOS DA SUPERCLASSE

Agora que temos uma forma forte de se representar um `Ebook`, podemos criar um método `aplicaDescontoDe` com o desconto máximo de 15%. Assim, evitamos aquele `if` que verificava se um `Livro` era impresso ou não. Na classe `Livro` o método deve ficar assim:

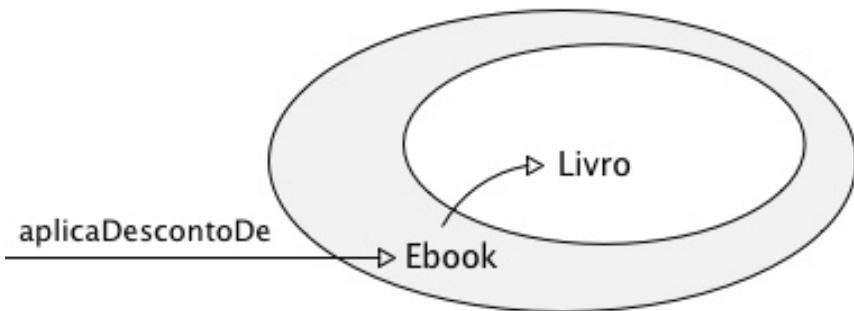
```
public boolean aplicaDescontoDe(double porcentagem) {  
    if (porcentagem > 0.3) {  
        return false;  
    }  
    this.valor -= this.valor * porcentagem;  
    return true;  
}
```

Mas no lugar de a classe `Ebook` herdar esse comportamento (que deve ser diferente para ela), podemos reescrevê-lo como a seguir:

```
public class Ebook extends Livro {  
  
    private String watermark;  
  
    public Ebook(Autor autor) {  
        super(autor);  
    }  
  
    public boolean aplicaDescontoDe(double porcentagem) {  
        if (porcentagem > 0.15) {  
            return false;  
        }  
        this.valor -= this.valor * porcentagem;  
        return true;  
    }  
  
    // get e set do watermark  
}
```

Por mais que um `Ebook` seja um `Livro`, quando alguém chamar o método `aplicaDescontoDe`, o desconto será de 15%, e não de 30% com o está definido na classe pai.

A JVM sempre vai procurar o método primeiro no objeto que foi instanciado e apenas quando não encontrar procurará em sua *superclasse*. Você pode ver um exemplo aqui:



@Override

Opcionalmente podemos marcar os métodos reescritos com a anotação `@Override`. Uma anotação é apenas uma marcação, como um *post-it* com uma observação importante sobre o local anotado. As *annotations* surgiram no Java 1.5, e têm um papel bastante importante na maior parte das bibliotecas mais atuais da linguagem. Por questões de compatibilidade, o uso do `@Override` é facultativo, mas é bastante interessante anotar os seus métodos dessa forma, já que, com isso, o compilador nos ajudará a validar que esse método é realmente uma reescrita.

```
@Override  
public boolean aplicaDescontoDe(double porcentagem) {  
    if (porcentagem > 0.15) {  
        return false;  
    }  
    this.valor -= this.valor * porcentagem;  
    return true;  
}
```

Ao anotar nosso método com `@Override`, o código não compilará caso esse método não exista na classe pai (*superclasse*).

Antes de testar essa mudança, precisamos fazer uma ultima alteração para que esse método compile.

Como a visibilidade do atributo `valor` da classe `Livro` é `private`, a linha que o acessa diretamente do método `aplicaDescontoDe` da classe `Ebook` não vai funcionar. Afinal, um atributo `private` só pode ser acessado pela própria classe, nem mesmo as classes filhas (*subclasses*) podem violar essa regra.

Para que o código funcione, precisamos aumentar essa visibilidade, mas já conhecemos o problema de deixar os atributos `public`. Uma alternativa é modificar a visibilidade dos atributos da classe `Livro` para `protected`, que é um meio termo entre `public` e `private`.

A visibilidade `protected` deixará os atributos visíveis também para as classes filhas, mas normalmente evitamos utilizar esse modificador de visibilidade porque ele acaba tendo mais algumas exceções que afrouxam o nosso encapsulamento. Conheceremos mais sobre o `protected` adiante, mas desde já vamos evitar essa alternativa.

Uma forma bem interessante e simples de resolver o problema é utilizando a *interface da classe* pai, seus próprios métodos. Por exemplo, no lugar de acessar o atributo diretamente, poderíamos utilizar os métodos `getValor` e `setValor` como a seguir:

```
@Override  
public boolean aplicaDescontoDe(double porcentagem) {  
    if (porcentagem > 0.15) {  
        return false;  
    }  
    double desconto = this.getValor() * porcentagem;  
    this.setValor(this.getValor() - desconto);  
    return true;  
}
```

Dessa forma, não perderemos nem um pouco do encapsulamento da classe `Livro`! Quanto mais você estudar e se habituar com a orientação a objetos, mais você perceberá que garantir o encapsulamento das classes é fundamental. Ainda que um pouco, quando liberamos acesso dos atributos da *superclasse* para suas classes filhas, estamos violando o encapsulamento dessa classe.

THIS OU SUPER?

No método `aplicaDescontoDe` da classe `Ebook`, chamamos o getter e setter do atributo `valor` utilizando a palavra reservada `this`. Uma outra forma de fazer isso seria utilizando o `super`. Assim, estamos deixando claro que o método invocado é da classe pai. Existem situações em que é muito interessante utilizar `super`, como neste caso:

```
@Override  
public boolean aplicaDescontoDe(double porcentagem) {  
    if (porcentagem > 0.15) {  
        return false;  
    }  
    return super.aplicaDescontoDe(porcentagem);  
}
```

Repare que, logo após fazer a nossa condição, estamos delegando para a lógica do método `aplicaDescontoDe` da classe pai. Com isso, evitamos repetir a lógica que já está definida na *superclasse*. Mas o que aconteceria se no lugar de `super` tivéssemos utilizado o `this`?

Nesse caso, o método `aplicaDescontoDe` da classe `Ebook` estaria chamando ele mesmo! Entraríamos em um *looping infinito*.

Pronto, chegou a hora de testar esse código para garantir que tudo está funcionando como esperado. Abra (ou crie, caso já não exista) a classe `RegrasDeDesconto` e escreva o seguinte código:

```
public class RegrasDeDesconto {  
  
    public static void main(String[] args) {  
  
        Autor autor = new Autor();  
        autor.setNome("Rodrigo Turini");  
  
        Livro livro = new Livro(autor);  
        livro.setValor(59.90);  
    }  
}
```

```
if (!livro.aplicaDescontoDe(0.3)){
    System.out.println("Desconto no livro não pode ser
        maior do que 30%");
} else {
    System.out.println("Valor do livro com desconto: "
        + livro.getValor());
}

Ebook ebook = new Ebook(autor);
ebook.setValor(29.90);

if (!ebook.aplicaDescontoDe(0.3)){
    System.out.println("Desconto no ebook não pode ser
        maior do que 15%");
} else {
    System.out.println("Valor do ebook com desconto: "
        + ebook.getValor());
}
}
```

A saída será:

```
Valor do livro com desconto: 41.93
Desconto no ebook não pode ser maior do que 15%
```

Mudando o valor de desconto passado para o e-book de 0.3 para 0.15, o resultado será:

```
Valor do livro com desconto: 41.93
Valor do ebook com desconto: 25.415
```

Ótimo, cada tipo de livro teve a sua regra de desconto devidamente aplicada.

5.3 REGRAS PRÓPRIAS DE UM LIVRO FÍSICO

Deixamos os comportamentos específicos de um Ebook bem encapsulados em sua classe, mas repare que há um problema em

nossa herança. Em que lugar devemos colocar os métodos que fazem sentido apenas para um livro físico? Por exemplo, todo livro físico possui uma taxa de impressão. Podemos criar um `getTaxaImpressao` para nos retornar esse valor, como a seguir:

```
public double getTaxaImpressao() {  
    return this.getValor() * 0.05;  
}
```

Mas onde adicionamos esse método? Se adicionarmos na classe `Livro`, o `Ebook` herdará esse comportamento que não deveria existir em seu tipo.

Mais uma vez entramos na regra: se um elemento é importante e tem regras específicas, ele deve ser representado como um objeto. Podemos criar uma nova classe em nosso projeto, para representar tudo o que um `LivroFisico` tem e como ele se comporta. Observe:

```
public class LivroFisico extends Livro {  
  
    public LivroFisico(Autor autor) {  
        super(autor);  
    }  
  
    public double getTaxaImpressao() {  
        return this.getValor() * 0.05;  
    }  
}
```

Dessa maneira, também temos uma forma forte de representar um `LivroFisico`, bem encapsulada e que não causa nenhum efeito colateral nos demais filhos de `Livro`, como o `Ebook`.

UMA NOVA CLASSE A CADA TIPO DE LIVRO?

Uma reação bem comum de quem está aprendendo orientação a objetos é perguntar: *então eu devo criar uma nova classe pra cada tipo de livro?*. A resposta é sim. E você logo perceberá que isso não é um problema.

É muito mais interessante trabalhar com classes menores, que representam bem os seus tipos e possuem suas regras específicas bem encapsuladas, do que trabalhar com uma única classe carregando toda essa responsabilidade.

Poderíamos sim ter uma única classe `Livro` que tivesse todos os comportamentos de um `LivroFisico`, um `Ebook` e dos demais tipos que podem aparecer. Mas imagine o tamanho dessa classe, a quantidade de métodos e `ifs` encadeados para lidar com as diferentes regras de cada tipo de `Livro`. A classe seria enorme e bem difícil de manter.

Lembre-se sempre que *Java é uma linguagem fortemente tipada*, você pode e deve criar objetos para representar cada tipo de elemento da sua aplicação.

5.4 VENDENDO DIFERENTES TIPOS DE LIVRO

Para dar mais um passo em nossa aplicação, criaremos agora uma classe `RegistroDeVendas`, por enquanto com um método `main` e o seguinte conteúdo:

```
public class RegistroDeVendas {  
  
    public static void main(String[] args) {  
  
        Autor autor = new Autor();  
        autor.setNome("Mauricio Aniche");  
  
        LivroFisico fisico = new LivroFisico(autor);  
        fisico.setNome("Test-Driven Development");  
    }  
}
```

```
        Ebook ebook = new Ebook(autor);
        ebook.setNome("Test-Driven Development");
    }
}
```

Repare que criamos um `LivroFisico` e um `Ebook` com o mesmo Autor. Nossa próximo passo será adicionar esses dois elementos em um `CarrinhoDeCompras`. O código deve ficar parecido com este:

```
CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
carrinho.adiciona(fisico);
carrinho.adiciona(ebook);
```

Vamos agora criar a classe `CarrinhoDeCompras` e seu método `adiciona`. Como vamos adicionar `LivroFisicos` e também `Ebooks`, uma forma de fazer isso seria criando dois métodos `adiciona`, um para cada tipo de `Livro` (uma *sobrecarga*):

```
public class CarrinhoDeCompras {

    public void adiciona(LivroFisico livro) {
        System.out.println("Adicionando: " + livro);
    }

    public void adiciona(Ebook livro) {
        System.out.println("Adicionando: " + livro);
    }
}
```

Esse código funcionaria sem nenhum problema, mas repare que está muito repetido. Além disso, a cada novo tipo de `Livro`, precisaremos criar um novo método `adiciona` que receba esse tipo como parâmetro, o que seria um tanto trabalhoso e difícil de manter.

Para evitar isso, podemos utilizar um recurso da linguagem bastante útil e poderoso. Como existe uma herança envolvida, podemos dizer que tanto um `LivroFisico` como um `Ebook` são filhos (*extensões*) da classe `Livro`. Poderíamos criar um único método `adiciona`, que recebe um `Livro` (*superclasse*) como parâmetro:

```
public class CarrinhoDeCompras {  
  
    public void adiciona(Livro livro) {  
        System.out.println("Adicionando: " + livro);  
    }  
}
```

Nosso código compilará e funcionará como esperado, pois podemos nos referenciar a esses objetos dessa forma mais genérica, pela sua classe pai. Esse interessante recurso é conhecido como **Polimorfismo**.

Veja o que acontece quando executamos nossa classe RegistroDeVendas adicionando os dois tipos de Livro no CarrinhoDeCompras, que agora possui um único método adiciona:

```
Adicionando: LivroFisico@9e89d68  
Adicionando: Ebook@3b192d32
```

Como estamos imprimindo o objeto inteiro e não um de seus atributos, o comportamento padrão é mostrar o nome da classe mais um @codigoEstranho. Entenderemos a fundo esse comportamento mais à frente, mas o importante agora é perceber que o objeto passado continua sendo um LivroFisico ou Ebook, apenas a forma como nos referimos a ele é que mudou.

NÃO ESTAMOS TRANSFORMANDO OBJETOS

Lembre-se que uma variável guarda uma referência para um objeto, e não um objeto em si. Até agora, estávamos sempre declarando um Ebook com o tipo Ebook. Repare:

```
Ebook ebook = new Ebook();
```

Mas, como vimos, também podemos dizer que um Ebook é do tipo Livro, afinal ele herda (**é um**) Livro.

```
Livro ebook = new Ebook();
```

Mas é fundamental perceber que não estamos transformando esse objeto. Se criamos um Ebook, ele será um Ebook e ponto. Estamos apenas nos referenciando a ele como um Livro, uma abstração.

Perceba que, como estamos referenciando o parâmetro passado para o método adiciona da classe CarrinhoDeCompras como um Livro, apenas os métodos presentes na classe Livro poderão ser invocados sem que um erro de compilação ocorra.

Para ficar mais claro, ainda que passando um objeto do tipo Ebook para o adiciona, ao tentar invocar seu método getWaterMark, o seguinte erro de compilação ocorrerá:

```
The method getWaterMark() is undefined for the type Livro
```

Faz sentido. Para que isso funcione, precisaríamos fazer um casting moldando o parâmetro livro para o tipo Ebook:

```
public void adiciona(Livro livro) {  
    Ebook ebook = (Ebook) livro;  
    ebook.getWaterMark();  
    // restante do código omitido  
}
```

Isso vai funcionar muito bem, contanto que o valor passado de fato seja do tipo Ebook. Isso é um tanto perigoso, pois podemos muito bem passar

um `LivroFisico` como argumento. Neste caso até seria possível fazer um `if` para validar que o objeto passado é uma instância de `Ebook`, mas deixaria de ser uma vantagem usar *polimorfismo* aqui.

5.5 ACUMULANDO TOTAL DE COMPRAS

O `CarrinhoDeCompras` precisa acumular o valor total que está sendo comprado. Podemos fazer isso de uma forma bem simples, adicionando um atributo `total` e incrementando com o valor do `Livro` a cada adição.

```
public void adiciona(Livro livro) {  
    System.out.println("Adicionando: " + livro);  
    total += livro.getValor();  
}
```

Além disso, antes de acumular o total da compra, o método `adiciona` aplicará um desconto de 5% para cada livro adicionado:

```
public void adiciona(Livro livro) {  
    System.out.println("Adicionando: " + livro);  
    livro.aplicaDescontoDe(0.05);  
    total += livro.getValor();  
}
```

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

```
public class CarrinhoDeCompras {  
  
    private double total;  
  
    public void adiciona(Livro livro) {  
        System.out.println("Adicionando: " + livro);  
        livro.aplicaDescontoDe(0.05);  
        total += livro.getValor();  
    }  
  
    public double getTotal() {  
        return total;  
    }  
}
```

Na classe RegistroDeVendas, vamos adicionar os valores de cada tipo de livro e depois disso imprimir o total adicionado no carrinho:

```
public class RegistroDeVendas {  
  
    public static void main(String[] args) {  
  
        Autor autor = new Autor();  
        autor.setNome("Mauricio Aniche");  
  
        LivroFisico fisico = new LivroFisico(autor);  
        fisico.setNome("Test-Driven Development");  
        fisico.setValor(59.90);  
  
        Ebook ebook = new Ebook(autor);  
        ebook.setNome("Test-Driven Development");  
        ebook.setValor(29.90);  
  
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();  
  
        carrinho.adiciona(fisico);  
        carrinho.adiciona(ebook);  
  
        System.out.println("Total " + carrinho.getTotal());  
    }  
}
```

O resultado da execução dessa classe será:

```
Adicionando: LivroFisico@9e89d68  
Adicionando: Ebook@3b192d32  
Total 85.31
```

Como ter certeza de que a JVM chamou o método certo?

Como o parâmetro recebido no método adiciona é um Livro, qual método `aplicaDescontoDe` foi executado? Do Livro, Ebook ou LivroFisico?

```
public void adiciona(Livro livro) {  
    System.out.println("Adicionando: " + livro);
```

```
livro.aplicaDescontoDe(0.05);
total += livro.getValor();
}
```

Apesar do *Polimorfismo*, em Java, o método executado sempre será escolhido em tempo de execução (*runtime*) e não em compilação. Ou seja, a JVM vai localizar o objeto instanciado em memória, um `Ebook` por exemplo, e chamar o método `aplicaDescontoDe` de sua classe e não da classe `Livro`, que é o tipo pelo qual ele foi referenciado.

Quer uma prova? Modifique o método `aplicaDescontoDe` da classe `Ebook` adicionando um `println` com a mensagem a seguir:

```
@Override
public boolean aplicaDescontoDe(double porcentagem) {
    if (porcentagem > 0.15) {
        return false;
    }
    System.out.println("aplicando desconto no Ebook");
    return super.aplicaDescontoDe(porcentagem);
}
```

Faça o mesmo com a classe `Livro`. Depois disso, mude o desconto aplicado no método `adiciona` para 40% e execute a classe `RegistroDeVendas` mais uma vez. O resultado será:

```
Adicionando: LivroFisico@9e89d68
Adicionando: Ebook@3b192d32
Total 89.8
```

Note que nenhum desconto foi aplicado, 89.90 é a soma do valor integral dos dois livros.

Mude agora o método `adiciona` para aplicar 16% de desconto. Ao executar a classe `RegistroDeVendas` teremos o resultado:

```
Adicionando: LivroFisico@9e89d68
aplicando desconto no Livro
Adicionando: Ebook@3b192d32
Total 80.21600000000001
```

Como esperado, apenas o desconto do `LivroFisico` foi aplicado, pois o método `aplicaDescontoDe` da classe `Ebook` limita o desconto a 15%.

Portanto, não importa a forma como nós nos referenciamos a nossos objetos, no fim o método executado sempre será o dele.

5.6 HERANÇA OU COMPOSIÇÃO?

Entenderemos melhor o problema mais à frente, mas *sempre que possível procure favorecer o uso da composição entre classes no lugar de utilizar herança*. Veremos que o uso da herança aumenta bastante o acoplamento de suas classes e de alguma forma sempre acaba comprometendo o *encapsulamento*.

Tudo isso será melhor aprofundado nos próximos capítulos, mas desde já você pode se interessar em ler o seguinte artigo no qual *James Gosling*, considerado pai do Java, fala sobre o assunto:

<http://www.artima.com/intv/gosling3P.html>

CAPÍTULO 6

Classe abstrata

6.1 QUAL O TIPO DE CADA LIVRO?

Agora que temos uma forma forte de representar os diferentes tipos de Livro, podemos nos expressar bem ao fazer uma venda:

```
Ebook ebook = new Ebook(autor);  
ebook.setNome("CDI");  
  
carrinho.adiciona(ebook);
```

Além de `Ebook`, também temos um `LivroFisico`, que já provou ser um objeto diferente por ter comportamentos diferentes. Mas o que estamos vendendo quando fazemos `new` em um `Livro`?

```
Livro livro = new Livro(autor);  
livro.setNome("CDI");
```

```
carrinho.adiciona(livro);
```

Afinal, o que é um `Livro` agora? Um `Ebook` ou um `LivroFisico`? Na verdade, nenhum dos dois. Um `Livro` é apenas uma **abstração** de tudo que os diferentes tipos de livro devem ter (*herdar*) em nossa livraria.

Ao fazer uma venda, queremos saber ao certo o tipo de livro que está sendo vendido, nunca deveríamos permitir a venda de um `Livro`, mas sim de suas *subclasses*.

Para nossa sorte, há uma forma bem simples de impedir que a classe `Livro` seja instanciada e utilizada dessa forma: podemos simplesmente adicionar em sua declaração o modificador `abstract`. Repare:

```
public abstract class Livro {  
    // continuação do código  
}
```

A partir do momento em que tornamos nossa classe abstrata, o compilador vai impedir que qualquer código tente instanciar um `Livro`. Por exemplo, em nosso `CadastroDeLivros`:

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

O erro de compilação apresentado será:

```
Cannot instantiate the type Livro
```

Bem claro, não acha? Não se pode mais criar um `Livro` e ponto final.

POR QUE PRECISAMOS DA CLASSE LIVRO?

Afinal, para que serve a classe `Livro`, se não podemos mais instanciá-la? Lembre-se que essa classe idealiza tudo o que um `Livro` tem, ela ainda está sendo muito útil isolando todos os atributos e comportamentos que são um padrão entre os diferentes tipos de livro. A classe passa a servir exclusivamente para *herança* e *polimorfismo*, que são recursos bastante úteis e poderosos.

Todo livro criado agora precisa ter seu tipo bem definido, portanto precisaremos mudar todas as partes de nosso código que criava um `Livro` para usar uma de suas classes filhas. A classe `CadastroDeLivros`, por exemplo, ficará assim:

```
public class CadastroDeLivros {  
  
    public static void main(String[] args) {  
  
        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(autor);  
        livro.setNome("Java 8 Prático");  
        livro.setDescricao("Novos recursos da linguagem");  
        livro.setValor(59.90);  
        livro.setIsbn("978-85-66250-46-6");  
  
        livro.mostrarDetalhes();  
  
        // outros livros cadastrados  
    }  
}
```

Apesar de não poder instanciar uma classe abstrata, você ainda pode (e muitas vezes deve) usá-la como referência. Como no método `adiciona` do

CarrinhoDeCompras:

```
public void adiciona(Livro livro) {  
    System.out.println("Adicionando: " + livro);  
    livro.aplicaDescontoDe(0.16);  
    total += livro.getValor();  
}
```

Repare que ainda estamos recebendo um `Livro` como parâmetro para continuar usando o *polimorfismo*. Isso é perfeitamente possível, afinal um `Ebook` e um `LivroFisico` são filhos de `Livro`, portanto herdam o seu tipo.

QUANDO UMA CLASSE DEVE SER ABSTRATA?

Quando você for planejar a hierarquia e herança de suas classes, você verá que algumas classes são bastante específicas e que jamais deveriam ser instanciadas. A classe `Animal` pode ser vista como um exemplo. O que exatamente é um `Animal`? Poderia ser um `Leao`, um `Pinguim` ou qualquer outro `Animal` do planeta. Essa classe pode definir tudo o que todos os animais têm em comum, mas cada tipo de `Animal` tem suas particularidades e deve ser representado de uma forma própria.

6.2 MINILIVRO NÃO TEM DESCONTO!

Agora que `Livro` é abstrato, podemos adicionar um novo tipo de `Livro` em nossa livraria. Será um `Minilivro`, representando um livro mais enxuto e com algumas particularidades que logo conheceremos. Vamos começar criando essa nova classe:

```
public class MiniLivro extends Livro {  
  
    public MiniLivro(Autor autor) {  
        super(autor);  
    }  
}
```

O que acontecerá quando modificarmos o código da classe `RegrasDeDesconto` criando um `MiniLivro` onde anteriormente criávamos um `Livro`? Vamos fazer essa alteração e executá-la para ver o resultado. A classe deve ficar assim:

```
public class RegrasDeDesconto {  
  
    public static void main(String[] args) {  
  
        Autor autor = new Autor();  
        autor.setNome("Rodrigo Turini");  
  
        Livro livro = new MiniLivro(autor);  
        livro.setValor(39.90);  
  
        if (!livro.aplicaDescontoDe(0.3)){  
            System.out.println("Desconto no livro não pode  
                ser maior do que 30%");  
        } else {  
            System.out.println("Valor do livro com desconto: "  
                + livro.getValor());  
        }  
        // outros descontos omitidos  
    }  
}
```

Ao executar esse código o retorno será:

```
aplicando desconto no Livro  
Valor do livro com desconto: 27.93
```

Observe que foi aplicado um desconto de 30% no valor do livro, mas um `MiniLivro` não pode ter desconto! Ele já é um livro com preço promocional, não podemos permitir que nenhum outro valor de desconto seja aplicado.

Ao criar um `MiniLivro`, mesmo sem definirmos isso, ele já poderia ter um desconto de até 30% pois herdou esse comportamento da classe `Livro`. Há uma forma simples de resolver o problema: poderíamos reescrever o método `aplicaDescontoDe` da classe `MiniLivro` para sempre retornar `false`. Seu código ficaria assim:

```
public class MiniLivro extends Livro {

    public MiniLivro(Autor autor) {
        super(autor);
    }

    @Override
    public boolean aplicaDescontoDe(double porcentagem) {
        return false;
    }
}
```

Essa solução resolveria o problema, mas é um tanto sensível. Será que qualquer pessoa que criasse esse novo tipo de `Livro` se lembraria de modificar o desconto padrão? Provavelmente não.

É bastante improvável que um novo desenvolvedor, ou mesmo quem tenha criado a regra, lembre-se deste detalhe sempre que um novo tipo de livro for criado. Depender da memória humana nunca é uma boa estratégia.

No lugar de precisar sobrescrever o método sempre que houver desconto, poderíamos inverter a situação deixando o comportamento padrão do `Livro` abstrato retornar `false`. Dessa forma, por padrão nenhum livro terá desconto. Sobrescrevemos essa regra apenas quando for necessário.

Nosso código ficaria assim:

```
public abstract class Livro {

    private String nome;
    private String descricao;
    private double valor;
    private String isbn;
    private Autor autor;

    public Livro(Autor autor) {
        this.autor = autor;
        this.isbn = "000-00-00000-00-0";
    }

    public boolean aplicaDescontoDe(double porcentagem) {
```

```
        return false;
    }

    // outros métodos, getters e setters
}
```

Agora que o método `aplicaDescontoDe` da classe pai sempre retorna `false`, precisaremos mudar a classe `LivroFisico` para permitir até 30% de desconto:

```
public class LivroFisico extends Livro {

    public LivroFisico(Autor autor) {
        super(autor);
    }

    public double getTaxaImpressao() {
        return this.getValor() * 0.05;
    }

    public boolean aplicaDescontoDe(double porcentagem) {
        if (porcentagem > 0.3) {
            return false;
        }
        double desconto = getValor() * porcentagem;
        setValor(getValor() - desconto);
        System.out.println("aplicando desconto no LivroFisico");
        return true;
    }
}
```

Nossa classe `MiniLivro` não precisará mais sobrescrever o método. Seu código pode ficar assim:

```
public class MiniLivro extends Livro {

    public MiniLivro(Autor autor) {
        super(autor);
    }
}
```

Essa estratégia é um pouco mais interessante, afinal não permitiremos mais desconto do que uma classe deveria ter. O problema é que precisaremos lembrar de reescrever esse comportamento em todo novo tipo de `Livro` que tenha desconto. Ainda corremos o risco de esquecer e, desta vez, não permitir desconto para um `Livro` que tenha direito.

6.3 MÉTODO ABSTRATO

Se cada novo `Livro` terá uma estratégia de desconto diferente, ou seja, se não há um desconto padrão entre todos os tipos de livro, poderíamos simplesmente apagar esse método dessa classe abstrata e escrevê-lo apenas nas classes que devem ter desconto. O problema disso é que perderíamos o *polimorfismo*, afinal, se o método não estiver presente na tipo `Livro`, o seguinte código não funcionará:

```
public void adiciona(Livro livro) {  
    System.out.println("Adicionando: " + livro);  
    livro.aplicaDescontoDe(0.16);  
    total += livro.getValor();  
}
```

Nem todo `Livro` terá o método `aplicaDescontoDe`, não há nenhuma garantia disso.

Podemos resolver o problema de uma forma mais efetiva. Toda classe abstrata, como é o caso da nossa classe `Livro`, pode ter **métodos abstratos**. Toda classe filha (*subclasse*) concreta (não abstrata) é obrigada a escrever os métodos abstratos da classe pai (*superclasse*), caso contrário seu código não compilará.

Para tornar o método `aplicaDescontoDe` abstrato na classe `Livro`, basta adicionar o modificador `abstract` em sua declaração e remover todo o corpo, colocando apenas um ponto e vírgula. Repare:

```
public abstract class Livro {  
  
    private String nome;  
    private String descricao;  
    private double valor;
```

```
private String isbn;
private Autor autor;

public Livro(Autor autor) {
    this.autor = autor;
    this.isbn = "000-00-00000-00-0";
}

public abstract boolean aplicaDescontoDe(double porcentagem);

// outros métodos, getters e setters
}
```

Apenas classes abstratas podem ter métodos abstratos. Afinal, se alguém invocasse o método `aplicaDescontoDe` da classe `Livro`, qual seria o resultado? Não há nenhuma implementação nele, portanto ele não poderia ser executado.

A partir do momento em que tornamos o método `aplicaDescontoDe` abstrato, todas as classes filhas precisam escrevê-lo. Por esse motivo, a classe `MiniLivro` vai parar de compilar. O erro será:

```
The type MiniLivro must implement the inherited abstract method
Livro.aplicaDescontoDe(double)
```

Para que tudo funcione a classe precisará ficar assim:

```
public class MiniLivro extends Livro {

    public MiniLivro(Autor autor) {
        super(autor);
    }

    @Override
    public boolean aplicaDescontoDe(double porcentagem) {
        return false;
    }
}
```

Ainda não é uma solução perfeita, estamos sendo obrigados a escrever esse método na classe `MiniLivro` mesmo que ela não possua essa regra de

desconto. Mas a grande vantagem é que eliminamos totalmente a possibilidade de esquecer de definir esse comportamento em novas implementações da classe `Livro`. A partir de agora, todo filho de `Livro` logo quando criado já precisará definir qual a sua regra de desconto, caso contrário o código não compila.

6.4 RELEMBRANDO ALGUMAS REGRAS

Não é nem um pouco complicado trabalhar com classes abstratas, mas existem algumas regras que precisam ser respeitadas. Apenas para relembrar, são elas:

- Uma classe pode ser abstrata sem ter nenhum método abstrato. A partir do momento em que ela se tornar abstrata, nenhum código poderá mais instanciá-la.
- Se você declarar um método abstrato, precisará tornar a classe abstrata também! Você não pode ter métodos abstratos em uma classe que não é abstrata.
- Uma classe abstrata pode ter métodos abstratos e não abstratos (concretos).
- Toda classe filha (*subclasse*) precisa implementar os métodos abstratos da classe pai (*superclasse*). A não ser que ela também seja abstrata.

CAPÍTULO 7

Interface

Além dos mais diversos tipos de `Livros`, nossa livraria também trabalhará com `Revistas` e futuramente outros produtos. Podemos criar uma nova classe para representá-la, como a seguir:

```
public class Revista {  
  
    private String nome;  
    private String descricao;  
    private double valor;  
    private Editora editora;  
  
    // getters e setters  
  
    public boolean aplicaDescontoDe(double porcentagem) {  
        if (porcentagem > 0.1) {
```

```
        return false;
    }
    double desconto = getValor() * porcentagem;
    setValor(getValor() - desconto);
    return true;
}
}
```

Repare que, além de um `nome`, `descricao` e `valor`, uma revista também possui uma regra de desconto e é composta pela classe `Editora`. Essa é uma outra classe bastante simples:

```
public class Editora {

    private String nomeFantasia;
    private String razaoSocial;
    private String cnpj;

    // getters e setters
}
```

Precisamos agora evoluir nosso `CarrinhoDeCompras` para que seja possível, além de `Livros`, adicionar `Revistas`. Uma solução seria duplicar seu método `adiciona`:

```
public class CarrinhoDeCompras {

    private double total;

    public void adiciona(Livro livro) {
        System.out.println("Adicionando: " + livro);
        livro.aplicaDescontoDe(0.05);
        total += livro.getValor();
    }

    public void adiciona(Revista revista) {
        System.out.println("Adicionando: " + revista);
        revista.aplicaDescontoDe(0.05);
        total += revista.getValor();
    }
}
```

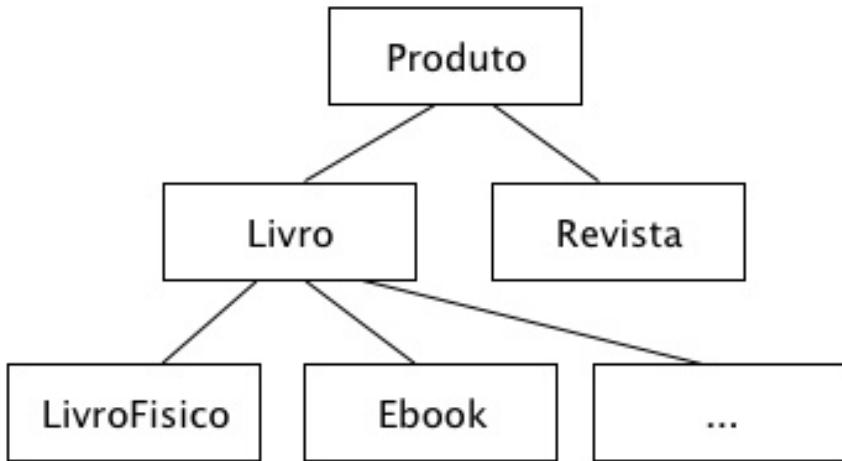
```
    }

    public double getTotal() {
        return total;
    }
}
```

Mas note como os dois métodos ficaram bastante parecidos, há muita repetição de código! Além disso, para cada novo produto precisaríamos criar um novo método, o que tornaria trabalhosa a evolução dessa classe.

Poderíamos fazer a classe `Revista` herdar de `Livro`, de modo que o *polimorfismo* seria aplicável e teríamos um único método adiciona. Mas ao fazer isso toda `Revista` teria um `Autor`, `ISBN` além dos demais atributos e comportamentos que só se aplicam para os `Livros`, uma verdadeira bagunça.

Sim, podemos criar um nível a mais na hierarquia de nossas classes adicionando a classe `Produto`, assim `Livro` e `Revista` herdariam de `Produto` e poderíamos utilizar esse tipo no *polimorfismo*. A hierarquia de nossas classes seria:



O grande problema dessa abordagem é que aumentaria demais o **acoplamento entre as classes**, ou seja, o quanto uma classe depende da outra. A herança cria uma relação muito forte entre as classes. Ao mudar a classe `Produto`, por exemplo, todas as *subclasses* (e também *subclasses* das *subclasses*) seriam influenciadas.

E se eu quiser aplicar a mudança apenas para algumas classes? Teríamos que sobrescrever esse novo comportamento de uma forma desnecessária em todas as outras, assim como fizemos com a classe `Minilivro`. Mesmo sem existir desconto para essa classe, ela foi obrigada a sobrescrever o método `aplicaDescontoDe` para apenas retornar `false`.

Com o passar do tempo, o uso da herança faz com que nossas classes fiquem cada vez mais acopladas e isso dificulta sua evolução.

7.1 O CONTRATO PRODUTO

Em Java, há uma outra forma para se tirar proveito de todos os benefícios do *polimorfismo* sem ter que acoplar tanto as suas classes com vários níveis de herança. Você pode estabelecer um fator em comum entre as classes, criando uma espécie de contrato.

Para esse contrato, não importa a forma como será implementado, a única coisa que importa é que seus métodos (*cláusulas*) sejam implementados de alguma forma. Isso lembra algo? Sim, é bastante parecido com um *método abstrato* cujo corpo você só define na *superclasse* para que todas as *subclasses* herdem a obrigação de implementá-lo.

Esse tipo de contrato Java é conhecido como *Interface*.

NÃO SE TRATA DE UMA INTERFACE GRÁFICA

É muito comum confundir no inicio, mas não estamos falando de uma interface gráfica de usuário (*GUI*). Também não estamos falando da *interface da classe*, seus métodos públicos. Você perceberá no decorrer do capítulo que se trata de um recurso diferente, um **contrato Java**.

Podemos criar uma interface `Produto`, que por enquanto terá um único método abstrato estabelecendo que todo produto deve ter o método

`getValor`. Uma interface se parece bastante com uma classe abstrata que tenha apenas métodos abstratos, mas no lugar de declará-la como uma classe, utilizamos a palavra reservada `interface`:

```
public interface Produto {  
  
    public abstract double getValor();  
}
```

Como todo método *sem corpo* de uma interface é abstrato, o uso do modificador `abstract` é opcional. Não precisamos também adicionar o modificador `public`, pois seus métodos também são públicos por padrão. Podemos simplificar a escrita da interface `Produto` deixando apenas:

```
public interface Produto {  
  
    double getValor();  
}
```

Uma interface não pode ter atributos e, até a versão 1.7 da linguagem, também não pode ter nenhum método concreto, ou seja, com implementação. Veremos que, a partir do Java 1.8, isso mudou um pouco.

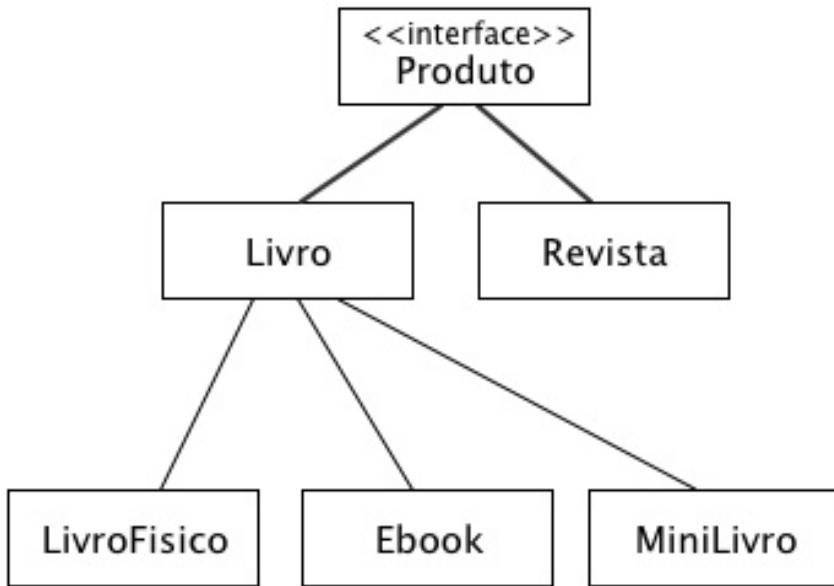
Podemos agora fazer com que todas as classes que idealizam *produtos* de nossa livraria assinem o contrato `Produto`. Para fazer isso, basta adicionar a palavra-chave `implements` seguida do nome da interface que deve ser implementada na declaração das classes, veja:

```
public abstract class Livro implements Produto {  
    // atributos e métodos omitidos  
}  
  
public class Revista implements Produto {  
    // atributos e métodos omitidos  
}
```

Como todas essas classes já possuem o método `getValor` declarado, você não perceberá nenhuma diferença. Nosso código passa a funcionar como esperado. Mas é importante perceber que se apagarmos o método

`getValor` a classe deixará de compilar, afinal toda classe que implementa a interface `Produto` é obrigada a implementar seus métodos abstratos.

Veja que com a interface o diagrama de nossas classes fica assim:



Como todos que implementam uma interface podem ser referenciados por este tipo, podemos usar *polimorfismo* com *interfaces*. Por exemplo, no método `adiciona` do `CarrinhoDeCompras` podemos receber um `Produto` como parâmetro:

```

public void adiciona(Produto produto) {
    System.out.println("Adicionando: " + produto);
    produto.aplicaDescontoDe(0.16);
    total += produto.getValor();
}
  
```

O *polimorfismo* funcionará, mas o problema desse código é que nem todo `Produto` tem o método `aplicaDescontoDe`, mas apenas os filhos da classe

Livro. Sim, poderíamos mover o método abstrato `aplicaDescontoDe` para a interface `Produto`, mas se nem todos os produtos têm um desconto, devemos evitar isso. Por enquanto, removeremos esse desconto, mas logo veremos uma forma mais interessante de resolver o problema. Nossa método deve ficar assim:

```
public void adiciona(Produto produto) {  
    System.out.println("Adicionando: " + produto);  
    total += produto.getValor();  
}
```

Rode a classe `RegistroDeVendas` para ver que tudo está funcionando como esperado. A classe continua assim:

```
public class RegistroDeVendas {  
  
    public static void main(String[] args) {  
  
        Autor autor = new Autor();  
        autor.setNome("Mauricio Aniche");  
  
        LivroFisico fisico = new LivroFisico(autor);  
        fisico.setNome("Test-Driven Development");  
        fisico.setValor(59.90);  
  
        Ebook ebook = new Ebook(autor);  
        ebook.setNome("Test-Driven Development");  
        ebook.setValor(29.90);  
  
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();  
  
        carrinho.adiciona(fisico);  
        carrinho.adiciona(ebook);  
  
        System.out.println("Total " + carrinho.getTotal());  
    }  
}
```

Ao executá-la, o resultado será:

Adicionando: LivroFisico@16f65612

Adicionando: Ebook@311d617d

Total 89.8

7.2 DIMINUINDO ACOPLAMENTO COM INTERFACES

O uso da interface já nos ajudou a resolver o problema de *polimorfismo* de nossos produtos. Agora há uma forma simples e flexível para representar qualquer Produto de nossa livraria. Mas ainda precisamos resolver o problema do método `aplicaDescontoDe`.

Ao adicionar esse método abstrato na classe `Livro`, obrigamos todas as suas *subclasses* a implementá-lo, mas não é bem isso que precisamos. Essa solução deixa de ser interessante quando nem todos os `Livros` tenham esse comportamento, como é o caso do `MiniLivro`, que não pode ter um desconto.

Outra questão é que apenas os filhos da classe `Livros` têm essa obrigação, sendo que uma `Revista` também deve possuir o método `aplicaDescontoDe`. Podemos diminuir esse acoplamento de uma forma bem simples, criando uma nova interface! Vamos chamá-la de `Promocional`:

```
public interface Promocional {  
  
    boolean aplicaDescontoDe(double porcentagem);  
}
```

Agora podemos remover o método `aplicaDescontoDe` abstrato da classe `Livro` e dizer que apenas as classes promocionais, que possuem desconto, implementam essa nova interface:

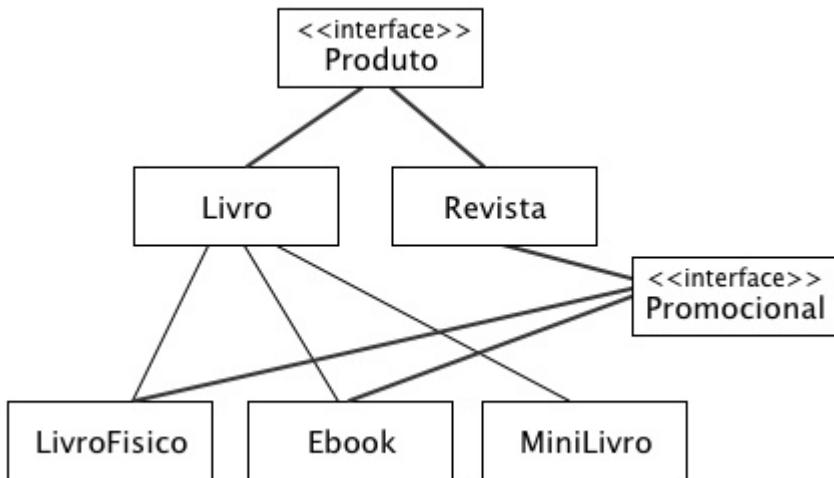
```
public class LivroFisico extends Livro implements Promocional {  
    // atributos e métodos omitidos  
}  
  
public class Ebook extends Livro implements Promocional {  
    // atributos e métodos omitidos  
}
```

```
public class Revista implements Produto, Promocional {  
    // atributos e métodos omitidos  
}
```

Assim como podemos assinar diversos contratos ao longo de nossas vidas, uma classe também pode implementar diversas interfaces. Repare que a classe `Revista` agora implementa duas interfaces, utilizando uma vírgula em sua declaração.

A grande vantagem de trabalhar com interfaces é que apenas as classes que a implementam são obrigadas a implementar seus métodos, portanto, se eu não quero que `MiniLivro` tenha desconto, basta não implementar a interface `Promocional`.

Observe como a estrutura de nossas classes está mais flexível. Qualquer nova classe pode passar a ser promocional, sem herdar nenhuma outra obrigação. O mesmo ocorre com o `Produto`: tudo que uma classe precisa fazer para ter o tipo `Produto` é assinar esse contrato e implementar seu método `getValor`, sem nenhum efeito colateral indesejado.



Você sempre pode e deve favorecer interfaces para criar polimorfismo entre suas classes, seu código fica muito mais flexível e com menor acoplamento.

7.3 NOVAS REGRAS DA INTERFACE NO JAVA 8

default methods

Desde o Java 8, uma *interface* pode ter métodos concretos. Com isso, suas implementações não são obrigadas a reescrevê-los. Esse novo recurso é conhecido como *default method*.

Basta adicionar a palavra reservada `default` no início da declaração de um método de interface para que ele possa ter código implementado, por exemplo:

```
public interface Promocional {  
  
    boolean aplicaDescontoDe(double porcentagem);  
  
    default boolean aplicaDescontoDe10Porcento() {  
        return aplicaDescontoDe(0.1);  
    }  
}
```

Dessa forma, toda classe que implementar a interface `Promocional` terá um novo método `aplicaDescontoDe10Porcento`, sem a obrigação de implementar nenhuma linha de código.

Podemos testar essa mudança na classe `RegistroDeVendas`, chamando o método `aplicaDescontoDe10Porcento` no `LivroFisico`:

```
if (fisico.aplicaDescontoDe10Porcento()) {  
    System.out.println("Valor agora é " + fisico.getValor());  
}
```

Ao executar a classe, teremos como resultado:

```
Valor agora é 53.91  
Adicionando: LivroFisico@21588809
```

Adicionando: Ebook@2aae9190

Total 83.81

Mais à frente veremos que diversos *default methods* foram adicionados nas *interfaces* da API do Java. Foi possível evoluir seu comportamento sem quebrar compatibilidade com suas implementações já existentes e bastante utilizadas.

Interface funcional

Uma interface não precisa ter um único método abstrato, mas essa é uma estrutura bem comum. Normalmente, trabalhar com interfaces menores é uma estratégia interessante, afinal temos mais flexibilidade. Se alguém implementar aquela interface é porque realmente precisa do comportamento que ela estabelece.

A partir do Java 8, as *interfaces* que obedecem essa regra de ter um único método abstrato podem ser chamadas de *interface funcional*. Mesmo sem ter esse propósito, nossas interfaces `Produto` e `Promocional` possuem essa estrutura e se enquadram no perfil de uma *interface funcional*.

Logo veremos que a *interface funcional* é a chave para outros novos recursos da linguagem, que são as expressões *lambda* e *method references*.

A anotação `@FunctionalInterface`

Podemos marcar uma interface como funcional explicitamente, para que o fato de ela ser uma *interface funcional* não seja pela simples coincidência de ter um único método abstrato. Para fazer isso, usamos a anotação `@FunctionalInterface`:

```
@FunctionalInterface  
public interface Promocional {  
  
    boolean aplicaDescontoDe(double porcentagem);  
  
    default boolean aplicaDescontoDe10Porcento() {  
        return aplicaDescontoDe(0.1);  
    }  
}
```

Ao fazer essa alteração, note que nada mudou. Ela continua compilando e, executando nossas classes de teste, o resultado será o mesmo. Mas diferente do caso de não termos anotado nossa interface com `@FunctionalInterface`, tente alterá-la da seguinte forma, adicionando um novo método *abstrato*:

```
@FunctionalInterface  
public interface Promocional {  
  
    boolean aplicaDescontoDe(double porcentagem);  
    boolean naoSouMaisUmaInterfaceFuncional();  
  
    default boolean aplicaDescontoDe10Porcento() {  
        return aplicaDescontoDe(0.1);  
    }  
}
```

O código deixará de compilar, com a seguinte mensagem:

```
Invalid '@FunctionalInterface' annotation;  
Promocional is not a functional interface
```

E ao tentar compilar fora do Eclipse, com o `javac` como fizemos no primeiro capítulo, a mensagem seria ainda mais explicativa:

```
java: Unexpected @FunctionalInterface annotation  
Promocional is not a functional interface  
    multiple non-overriding abstract methods found  
    in interface Promocional
```

Outro detalhe que você já deve ter percebido é que podemos ter um ou mais *default methods* declarados em nossa interface e isso não influencia o fato de ela ser ou não uma `interface funcional`, apenas métodos abstratos são considerados.

Herança múltipla no Java 8?

Métodos defaults foram adicionados para permitir que interfaces evoluam sem quebrar código existente. Essa frase foi bastante repetida na

lista de discussão da especificação dessa nova versão da linguagem. Eles não foram criados para permitir alguma variação de herança múltipla ou de *mix-ins*. Vale lembrar que há uma série de restrições para esses métodos. Em especial, eles não podem acessar atributos de instância, até porque isso não existe em interfaces! Em outras palavras, não há herança múltipla ou compartilhamento de estado.

CAPÍTULO 8

Pacotes

8.1 ORGANIZANDO NOSSAS CLASSES

Por enquanto nossas classes estão todas em um mesmo arquivo, dentro da pasta `src`. Conforme o projeto vai evoluindo, mais e mais classes são criadas e fica cada vez mais difícil manter a organização de nosso projeto. Mas organização não será o único problema aqui.

Com o passar do tempo, trabalharemos com classes de terceiros (*bibliotecas*) e classes da própria *API* da linguagem, o que torna ainda maior o risco de criarmos uma classe com o nome igual a outra existente em alguma dessas bibliotecas.

Quer um exemplo? Poderíamos criar uma classe que se chama `Date`:

```
class Date {  
    private int dia;
```

```
private int mes;  
private int ano;  
  
// getters e setters  
}
```

Em um método `main` qualquer de nossa aplicação, poderíamos instanciar essa classe fazendo:

```
public static void main(String[] args) {  
    Date date = new Date();  
}
```

Mas na API do Java, já existe uma classe chamada `Date`. Como a JVM sabe diferenciar quando precisamos instanciar a nossa classe `Date` ou quando deve ser o `Date` do Java? A resposta é bem simples: em Java classes são agrupadas por pacotes (*packages*).

O pacote da classe `Date` da API do Java com certeza será diferente do pacote de nossa classe, portanto, para tirar a ambiguidade podemos instanciar a classe pelo seu **nome completo**, como por exemplo:

```
public static void main(String[] args) {  
    java.util.Date date = new java.util.Date();  
}
```

Estranho? Não se preocupe, vamos melhorar a legibilidade desse código logo. Mas repare que o nome completo da classe, ou *fully qualified name* como é comumente chamado, é composto por **nome do pacote** . (ponto) **nome da classe**. Neste caso, o nome do pacote onde se encontra a classe `Date` é `java.util`.

PACOTES DA API DO JAVA

Além do `java.util`, existem diversos pacotes com classes essenciais para nosso dia a dia trabalhando com Java. Algumas classes como a `String` e `System` não precisam ser escritas com o nome completo, pois fazem parte do pacote `java.lang`, que é o pacote padrão do Java. No próximo capítulo estudaremos a fundo o pacote `java.lang` e, no decorrer do livro, veremos alguns dos demais principais pacotes da *API*.

Observando o código a seguir você já deve ter sentido uma perda de legibilidade:

```
java.util.Date date = new java.util.Date();
```

Podemos simplificar esse código adicionando um `import` com o nome completo da classe no início de nosso arquivo. Repare:

```
import java.util.Date;

class Teste {
    public static void main(String[] args) {
        Date date = new Date();
    }
}
```

Só será necessário escrever o nome completo da classe sem um `import` se por alguma razão você precisar usar duas classes com o mesmo nome dentro do mesmo arquivo. Nesse caso, você poderá fazer o `import` de uma delas, mas utilizar o nome completo quando for se referir à outra.

A essa altura você já deve ter se perguntado: Qual o nome completo da nossa classe `Date`? Como nós criamos a classe diretamente no diretório `src`, ela não tem um pacote definido. Dizemos que ela está no *default package*.

Isso não é nada bom, toda classe deve ser agrupada em pacotes. Isso, além de ajudar na organização de nossos projetos, ajudará quando houver uma ambiguidade de nomes.

NOMENCLATURA PADRÃO DOS PACOTES JAVA

Por padrão, um pacote em Java sempre:

- é escrito em letra minúscula (*lowercase*);
- deve ser um nome de domínio, iniciado com *com, edu, gov* etc.

É muito natural que o pacote seja o seu domínio (ou da empresa), como `br.com.casadocodigo`, `br.com.alura` ou `br.com.caelum`. O link a seguir fala um pouco mais sobre as convenções de nomenclaturas de pacotes do Java:

<http://www.oracle.com/technetwork/java/codeconventions-135099.html>

Agora que já sabemos disso, podemos criar alguns pacotes para melhor organizar o nosso projeto. No Eclipse isso pode ser feito pelo menu `File > New > Package`, ou utilizando o atalho `Control + N` e selecionando a opção `package`.

A princípio, criaremos os pacotes:

- `br.com.casadocodigo.livraria` para as classes `Autor` e `Editora`;
- `br.com.casadocodigo.livraria.produtos` para as interfaces `Promocional`, `Produto` e suas implementações;
- `br.com.casadocodigo.livraria.teste` para nossas classes executáveis, ou seja, que possuem o método `main`.

Antes de mover as classes para seus devidos pacotes, é importante conhecer algumas regras. A primeira delas é que **nossas classes devem ser públicas para que fiquem visíveis entre os diferentes pacotes**. Logo falaremos mais sobre os modificadores de visibilidade e entenderemos a fundo as suas regras, mas desde já mantenha a regra em mente.

Outro detalhe importante é que, para o caso das classes públicas, o arquivo `.java` obrigatoriamente tem que ter o nome da classe. Por exemplo, a classe `Livro` tem que ser salva no arquivo `Livro.java`. Como vimos no início deste livro, o ideal é você sempre nomear suas classes dessa forma. Essa é considerada uma boa prática em Java.

Agora sim, mãos à massa! Você pode arrastar as classes com o mouse para mover entre pacotes, mas uma alternativa interessante é o atalho `Control + Alt + V`. Esse atalho é equivalente ao menu `Refactor > Move....`

Vamos selecionar as classes `Autor` e `Editora` e mover para o pacote `br.com.casadocodigo.livraria`. Nesse momento, alguns erros de compilação devem aparecer, mas **devemos terminar de mover as classes antes de corrigir qualquer um deles**.

Promocional, `Produto`, `Livro`, `LivroFisico`, `MiniLivro`, `Ebook`, `Revista` e qualquer outra implementação de `Produto` que você tenha criado podem ser movidos para o pacote `br.com.casadocodigo.livraria.produtos`.

Para terminar, vamos mover as demais classes, que têm um método `main`, para o pacote `br.com.casadocodigo.livraria.testes`. Lembre-se, em um projeto real não criariamós várias classes com método `main`, só estamos fazendo isso para testar os nossos exemplos.

package, import e possíveis erros

Se você moveu as classes utilizando o *Eclipse* ou alguma *IDE* equivalente, repare que foi adicionado um `import` para toda classe que referencia alguma presente em outro pacote. Caso você não tenha movido pela *IDE* ou por algum motivo o `import` não tenha sido adicionado, você precisará fazer isso manualmente como no exemplo:

```
package br.com.casadocodigo.livraria.testes;  
  
import br.com.casadocodigo.livraria.Autor;  
// outros imports  
  
public class CadastroDeLivros {
```

```
public static void main(String[] args) {  
  
    Autor autor = new Autor();  
    autor.setNome("Rodrigo Turini");  
  
    // continuação da classe  
}
```

Como a classe `Autor` está em um pacote (`package`) diferente da classe `CadastroDeLivros`, o `import` é necessário. Ele sempre é necessário quando queremos utilizar classes de outros pacotes. E não se esqueça que, para que uma classe seja visível para outro pacote, ela deve ser pública.

Um outro detalhe é que agora todas as classes têm o `package` a que pertencem declarado. Neste exemplo, o `package` da classe `CadastroDeLivros` é:

```
package br.com.casadocodigo.livraria.testes;
```

A ordem das instruções de seus arquivos `.java` agora será: primeiro o `package` ao qual ela pertence, seguido do(s) `import`(s) quando necessário e, por último, a declaração da classe.

IMPORTANDO TODAS AS CLASSES DE UM PACKAGE

No lugar de fazer os seguintes `imports`:

```
import br.com.casadocodigo.livraria.Autor;  
import br.com.casadocodigo.livraria.Editora;
```

Você também pode utilizar o `*` para importar todas as classes de um package, neste caso o `import` seria:

```
import br.com.casadocodigo.livraria.*;
```

É comum ouvirmos que isso prejudica a performance, mas na realidade não afetará o tempo de execução. O problema dessa abordagem é que não previne a ambiguidade de nomes. Por isso, é sempre recomendado que você importe classe a classe, pois, além de evitar problemas com classes de mesmo nome, tornará a legibilidade dos `imports` mais clara.

Ainda existe um erro de compilação em nosso projeto (ou talvez mais, se o seu código não estiver idêntico ao do livro). Em nosso caso, o problema é na própria classe `CadastroDeLivros`, quando ela invoca o método `mostrarDetalhes` do `LivroFisico`. O erro de compilação é o seguinte:

The method `mostrarDetalhes()` from the type `Livro` is not visible

Isso acontece porque, assim como as classes, para que um método seja visível em outro pacote ele precisa ser `public`. Podemos acessar a classe `Livro` e adicionar o modificador de visibilidade `public` no método ou fazer isso pelo atalho `Control + 1` do Eclipse (*quickfix*), selecionando a opção *Change visibility of method mostrarDetalhes() to public*. No final, o método deve ficar assim:

```
public void mostrarDetalhes() {  
    System.out.println("Mostrando detalhes do livro ");  
    System.out.println("Nome: " + nome);  
    System.out.println("Descrição: " + descricao);
```

```
System.out.println("Valor: " + valor);
System.out.println("ISBN: " + isbn);

if (this.temAutor()) {
    autor.mostrarDetalhes();
}
System.out.println("==");

}
```

ESTRUTURA DE PASTAS

Note que um pacote é representado como uma estrutura de pastas no seu sistema operacional. Por exemplo, a classe `Livro` que está localizada no pacote `br.com.casadocodigo.livraria` está dentro do diretório `br/com/casadocodigo/livraria`, que se encontra dentro da pasta `src` de seu projeto.

8.2 MODIFICADORES DE ACESSO

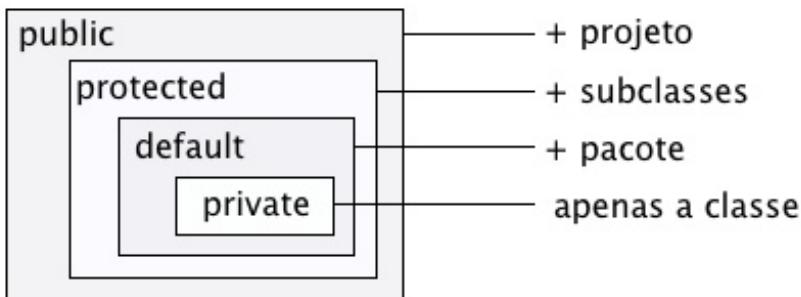
Agora que nossas classes estão organizadas em pacotes, podemos entender um pouco melhor os diferentes modificadores de acesso. Já vimos que, para uma classe ou método ser acessado de outro pacote, eles precisam ter visibilidade `public`. A regra é clara: uma classe pública pode ser acessada por qualquer outra classe presente no mesmo projeto. O mesmo vale para atributos, métodos e construtores.

Também já conhecemos o `private`. Esse modificador de acesso torna classes, atributos, métodos ou construtores visíveis apenas para a própria classe. Por esse motivo, uma classe não deve ser anotada com `private`, quem poderá acessá-la? Mas vimos que faz bastante sentido, para manter o encapsulamento, sempre deixar seus atributos `private`.

Há ainda a visibilidade `default` (quando não há modificador algum). A essa altura, você já pode ter percebido que neste caso apenas classes **do mesmo pacote** podem ter acesso aos atributos, construtores, métodos ou classes com a ausência de um modificador de acesso.

O último modificador é o `protected`, que tentamos evitar no capítulo de herança. Todo elemento que for `protected` ficará visível para a própria classe, para suas classes filhas e **também para quem estiver no mesmo pacote**. Classes também não podem utilizar esse modificador de acesso.

A imagem a seguir representa cada modificador de acesso e sua visibilidade:



CAPÍTULO 9

Arrays e exception

9.1 TRABALHANDO COM MULTIPLICIDADE

Nosso CarrinhoDeCompras só está fazendo uma parte de seu trabalho, que é acumular o valor total dos produtos adicionados. Também precisaremos ter um atributo nessa classe para guardar uma referência a todos esses Produtos adicionados.

Poderíamos adicionar um número limitado de atributos do tipo Produto, como:

```
public class CarrinhoDeCompras {  
  
    private double total;  
    private Produto produto1;  
    private Produto produto2;  
    private Produto produto3;
```

```
private Produto produto4;  
// ...  
  
// demais métodos da classe  
}
```

Deixando esses atributos com o tipo `Produto`, no lugar de um `Livro` ou `Revista` por exemplo, o polimorfismo nos daria a flexibilidade de adicionar qualquer classe que implemente essa interface `Produto`. Mas, fora isso, nosso código não está nem um pouco flexível! Além de termos um número **limitado** de produtos, no método `adiciona` precisaríamos fazer várias condições para verificar qual atributo está disponível antes de adicionar:

```
public class CarrinhoDeCompras {  
  
    private double total;  
    private Produto produto1;  
    private Produto produto2;  
    private Produto produto3;  
    private Produto produto4;  
    //..  
  
    public void adiciona(Produto produto) {  
        System.out.println("Adicionando: " + produto);  
  
        if (this.produto1 != null) {  
            this.produto1 = produto;  
        }  
        else if (this.produto2 != null) {  
            this.produto2 = produto;  
        }  
        else if (this.produto3 != null) {  
            this.produto3 = produto;  
        }  
        else if (this.produto4 != null) {  
            this.produto4 = produto;  
        }  
        // ...  
    }  
}
```

```
        else {
            System.out.println("Não tem mais espaços");
            return;
        }
    total += produto.getValor();
}
}
```

Imagine como seria para mostrar os detalhes dos produtos adicionados, novamente precisaríamos repetir esses `ifs` todos com as condicionais. Esse código seria bem difícil de manter, quase inviável. Além disso, quantos atributos do tipo `Produto` um `CarrinhoDeCompras` teria? 10? 20? Isso vai depender muito do cliente, ele pode estar comprando um único livro ou uma coleção completa e diversos outros produtos.

Criando um array de Produto

Evitamos até agora trabalhar com esse recurso, mas, como vimos no primeiro capítulo, podemos resolver esse problema trabalhando com arrays! Não lembra o que é isso? Dê uma boa olhada novamente na declaração do método `main` de qualquer uma de suas classes:

```
public static void main(String[] args) {
    // seu código aqui
}
```

Repare que ele recebe um array de `Strings` como parâmetro! O `String[] args`.

Podemos utilizar essa mesma estratégia em nosso `CarrinhoDeCompras`, basta remover todos estes atributos do tipo `Produto` e declarar um único array. Como fazer isso? É bem simples, basta adicionar `[]` depois do tipo e pronto. Nosso `CarrinhoDeCompras` vai ficar assim:

```
public class CarrinhoDeCompras {

    private double total;
    private Produto[] produtos;
```

```
public void adiciona(Produto produto) {  
    // precisamos adicionar no array  
}  
}
```

Um `array` é um tipo, um objeto. Isso significa que, antes de fazer qualquer operação com o atributo `produtos`, precisaremos instanciá-lo. A sintaxe é um pouco diferente, mas é bem simples:

```
public class CarrinhoDeCompras {  
  
    private double total;  
    private Produto[] produtos = new Produto[10];  
  
    ...  
}
```

Note que ao criar o `array` fomos obrigados a informar o seu tamanho! Neste caso, criamos um `array` de 10 posições. Poderíamos sim receber a quantidade de `Produtos` no construtor da classe ou criar um método `setProdutos`, isso a deixaria um pouco mais flexível. Mas a princípio, deixaremos o `CarrinhoDeCompras` assim, limitado a 10 produtos.

Agora que estamos trabalhando com um `array`, precisamos mudar o método `adiciona`. Podemos fazer algo como:

```
public void adiciona(Produto produto) {  
    System.out.println("Adicionando: " + produto);  
    this.produtos[1] = produto;  
    this.total += produto.getValor();  
}
```

Note que estamos adicionando todos os produtos na mesma posição do `array`, na posição 1. Um detalhe muito importante é que a posição 1 é a segunda posição do `array`, e não a primeira. Isso acontece pois seus índices vão de 0 (zero) até seu tamanho -1, ou seja, como criamos um `array` de 10 posições, ele vai de 0 a 9.

Uma alternativa seria mudar a assinatura do método `adiciona` para receber um `int indice` como parâmetro, mas deixar esse controle na mão

de quem vai chamar o método pode não ser muito interessante, isso abriria caminho para chamar o método duas vezes com o mesmo índice e um elemento sobrepor o outro, por exemplo.

No lugar disso podemos criar um atributo `contador` que é incrementado a cada nova adição, veja:

```
public class CarrinhoDeCompras {

    private double total;
    private Produto[] produtos = new Produto[10];
    private int contador = 0;

    public void adiciona(Produto produto) {
        System.out.println("Adicionando: " + produto);
        this.produtos[contador] = produto;
        contador++;
        this.total += produto.getValor();
    }

    public double getTotal() {
        return total;
    }
}
```

Problema resolvido! Sim, ainda estamos deixando passar algumas situações, como por exemplo tentar adicionar mais produtos do que cabe dentro de nosso array. E se eu tentar adicionar 11 produtos? Precisamos tratar isso de alguma forma. Por enquanto, o código ficará assim, mas em breve voltaremos para refatorar e cuidar desse tipo de detalhe da classe `CarrinhoDeCompras`.

ARRAY OU COLLECTIONS?

Os arrays estão tirando o seu sono? Trabalhar com arrays pode nem sempre ser tão simples e intuitivo. Isso é natural, afinal é um recurso carregado de uma série de conceitos e uma sintaxe um pouco diferente. Mais à frente conheceremos uma forma mais interessante de se trabalhar com coleções em Java, com a API de Collections!

Exibindo os Produtos adicionados

Devemos agora criar um método `getProdutos` para retornar o array de `Produto` e possibilitar a exibição dos detalhes de cada produto adicionado no `CarrinhoDeCompras`. Um possível uso desse método seria exibirmos o valor de cada `Produto` depois de adicionado:

```
Produto[] produtos = carrinho.getProdutos();

for (int i = 0; i < produtos.length; i++) {
    Produto produto = produtos[i];
    if (produto != null) {
        System.out.println(produto.getValor());
    }
}
```

Repare que estamos verificando se o `Produto` não é nulo antes de tentar imprimir seu valor, afinal temos um array com 10 posições mas só estamos ocupando duas. Ao executar esse código no `RegistroDeVendas`, o resultado será parecido com:

```
Valor agora é 53.91
Adicionando:
    br.com.casadocodigo.livraria.produtos.LivroFisico@2aae9190
Adicionando:
    br.com.casadocodigo.livraria.produtos.Ebook@2f333739
Total 83.81
53.91
29.9
```

O código está funcionando da forma como esperamos, mas um simples erro poderia causar problemas. Observe nosso `for`, ele faz a condição `i < produtos.length`. O atributo `length` de um `array` carrega o seu tamanho total, portanto neste exemplo será 10. O que aconteceria se, no lugar de `<`, tivéssemos escrito `<=?` Faça a alteração e rode o código para ver o resultado! Será algo parecido com:

```
Valor agora é 53.91
```

```
Adicionando:
```

```
    br.com.casadocodigo.livraria.produtos.LivroFisico@2aae9190  
Adicionando:
```

```
    br.com.casadocodigo.livraria.produtos.Ebook@2f333739
```

```
Total 83.81
```

```
53.91
```

```
29.9
```

```
Exception in thread "main"
```

```
java.lang.ArrayIndexOutOfBoundsException: 10
```

```
  at br.com.casadocodigo.livraria.testes  
    .RegistroDeVendas.main(RegistroDeVendas.java:38)
```

Note que, ao adicionar o operador de `=` na condição, estamos tentando iterar em uma posição (10) fora do limite do `array`, que vai de 0 a 9. Mas o que toda essa mensagem significa? Pode ser a primeira ou a milésima vez que vê uma mensagem como essa, mas a partir de agora passaremos a chamá-la pelo seu nome. Essa é uma `stacktrace`!

O ENHANCED-FOR DO JAVA 5

Desde o Java 1.5, é possível iterar em um `array` (ou qualquer outro `Iterable`, que veremos nos próximos capítulos) com uma sintaxe um pouco diferente. No lugar de:

```
for (int i = 0; i < produtos.length; i++) {  
    Produto produto = produtos[i];  
    if (produto != null) {  
        System.out.println(produto.getValor());  
    }  
}
```

Podemos fazer o mesmo da seguinte forma, com o `enhanced-for`:

```
for (Produto produto : produtos) {  
    if (produto != null) {  
        System.out.println(produto.getValor());  
    }  
}
```

Repare que nesse caso não temos um índice e não precisamos conhecer o tamanho (`length`) do array para conseguir percorrê-lo, isso evitaria a confusão com o `<=` que causou o `ArrayIndexOutOfBoundsException`.

9.2 AS DIFERENTES EXCEÇÕES E COMO LIDAR COM ELAS

Conhecendo a Stacktrace

Pode parecer um pouco assustador no começo, mas a `stacktrace` normalmente nos passa as informações necessárias e fundamentais para a compreensão e resolução dos problemas. Vamos analisá-la com mais atenção:

```
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException: 10
```

```
at br.com.casadocodigo.livraria.testes  
    .RegistroDeVendas.main(RegistroDeVendas.java:38)
```

Essa é uma `stacktrace` bem pequena, você pode perceber o quanto é fácil de entender. A primeira linha está nos informando qual o nome do problema (ou `exception`, como passaremos a chamá-lo) que ocorreu no código e nos diz em qual posição do array isso aconteceu (10). Na segunda e última linha temos o `fully qualified name` da classe em que a `exception` aconteceu, que ainda nos diz em qual linha, neste caso 38!

INVESTIGANDO PROBLEMAS

É muito comum enviarmos a `stacktrace` quando estamos procurando por ajuda em fóruns (como o <http://guj.com.br>) ou listas de discussões. Isso facilita bastante na investigação do problema.

Tratando exceptions

Evitar essa `exception` seria bem simples, bastaria corrigir a comparação com `<=` ou utilizar o `enhanced-for`. Mas há uma forma bastante conhecida e já utilizada em diversas linguagens para **tentar** executar um bloco de código de risco e **capturar** caso ocorra uma `exception` neste bloco, evitando que a `stacktrace` seja exibida para o usuário final e que a exceção pare a execução de nosso código. Esse recurso é conhecido como `try/catch`.

Para testá-lo, coloque um `sysout` logo após o `for` que está gerando o problema no `RegistroDeVendas`, algo como:

```
for (int i = 0; i <= produtos.length; i++) {  
    Produto produto = produtos[i];  
    if (produto != null) {  
        System.out.println(produto.getValor());  
    }  
}  
  
System.out.println("Fui executado!");
```

Rode novamente e repare na saída, a mensagem "Fui executado!" não foi impressa. Vamos agora adicionar o `try/catch` e executar o código novamente:

```
for (int i = 0; i <= produtos.length; i++) {  
    try {  
        Produto produto = produtos[i];  
        if (produto != null) {  
            System.out.println(produto.getValor());  
        }  
    } catch (ArrayIndexOutOfBoundsException e) {  
        System.out.println("deu exception no indice: "+ i);  
    }  
}  
  
System.out.println("Fui executado!");
```

Agora a saída será:

```
Total 83.81  
53.91  
29.9  
deu exception no indice: 10  
Fui executado!
```

Como a `exception` foi devidamente tratada, a execução do código não foi interrompida.

ENTENDENDO A SINTAXE DO TRY/CATCH

Pode parecer um pouco estranho ou até mesmo assustador no começo, mas a sintaxe do `try/catch` não tem nada de tão complicado. Veja:

```
try {  
    // algum código de risco  
} catch (ArrayIndexOutOfBoundsException e) {  
    // o que fazer aqui?  
}
```

Nesse caso, dentro do `catch` podemos colocar o código que queremos executar apenas quando uma `exception` do tipo declarado acontecer. Ou seja, nele colocamos o código que queremos executar apenas caso um `ArrayIndexOutOfBoundsException` ocorra.

Toda exception é filha de `Exception`

Em nosso `try/catch`, estamos capturando apenas o `ArrayIndexOutOfBoundsException`, mas o que aconteceria se o array `produtos` fosse `null`? Mesmo que com o `try/catch` declarado, receberíamos um outro tipo de exception, a `java.lang.NullPointerException`.

Isso acontece porque estamos deixando claro no parâmetro de nosso `catch` que queremos capturar um `ArrayIndexOutOfBoundsException`, e não um `NullPointerException`. Poderíamos fazer algo mais genérico como:

```
for (int i = 0; i <= produtos.length; i++) {  
    try {  
        Produto produto = produtos[i];  
        if (produto != null) {  
            System.out.println(produto.getValor());  
        }  
    } catch (Exception e) {  
        System.out.println("deu exception no indice: "+ i);  
    }  
}
```

```
    }  
}  
  
System.out.println("Fui executado!");
```

Veja que passamos a capturar `Exception` no lugar de `ArrayIndexOutOfBoundsException`, o que faz com que **qualquer Exception seja capturada**. Como você pode perceber, `Exceptions` é um objeto e uma *superclasse*, como qualquer outra ela é polimórfica.

Para deixar claro qual `exception` ocorreu podemos, por exemplo, dentro do `catch` invocar o método `printStackTrace` que está presente na classe, repare:

```
try {  
    Produto produto = produtos[i];  
    if (produto != null) {  
        System.out.println(produto.getValor());  
    }  
} catch (Exception e) {  
    System.out.println("deu exception no indice: "+ i);  
    e.printStackTrace();  
}  
  
System.out.println("Fui executado!");
```

Neste caso a saída será parecida com:

Valor agora é 53.91

Adicionando:

```
    br.com.casadocodigo.livraria.produtos.LivroFisico@2aae9190  
Adicionando:  
    br.com.casadocodigo.livraria.produtos.Ebook@2f333739  
Total 83.81  
Exception in thread "main" java.lang.NullPointerException  
    at br.com.casadocodigo.livraria.testes  
        .RegistroDeVendas.main(RegistroDeVendas.java:39)  
Fui executado!
```

Note como apesar de printar a `stacktrace`, o código continuou sendo executado.

Capturar `Exception` nem sempre é muito interessante, afinal se trata de um tipo muito genérico! Muitas vezes queremos dar uma tratativa diferente para cada tipo de exceção do código. Uma alternativa para isso seria capturar mais de uma `exception` em um mesmo bloco `try`:

```
try {
    Produto produto = produtos[i];
    if (produto != null) {
        System.out.println(produto.getValor());
    }
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("deu exception no indice: " + i);
} catch (NullPointerException e) {
    System.out.println("A array não foi instanciado!");
}
```

Agora estamos mostrando uma mensagem diferente para cada situação.

MULTICATCH DO JAVA 7

Caso você queira executar a mesma ação para dois tipos de `Exception` diferentes, desde o Java 7 você pode utilizar uma sintaxe um pouco diferente:

```
try {
    Produto produto = produtos[i];
    if (produto != null) {
        System.out.println(produto.getValor());
    }
} catch
(ArrayIndexOutOfBoundsException | NullPointerException e) {
    System.out.println("foi uma das duas");
}

A mensagem "foi uma das duas" será exibida caso uma ArrayIndexOutOfBoundsException ou um NullPointerException tenha ocorrido.
```

Executando uma ação com ou sem exception

Existem momentos em que precisamos executar alguma ação após um `try` ou `catch`, independente se houve ou não uma `exception` envolvida. Nesses casos, podemos utilizar uma terceira cláusula chamada `finally`.

Observe sua sintaxe:

```
try {
    // código de risco
} catch (NullPointerException e) {
    // tratando NPE
} catch (Exception e) {
    // tratando Exception
} finally {
    // alguma ação importante
}
```

As instruções do bloco `finally` serão executadas sempre, independente de tudo ter ido bem ou de alguma `exception` ter acontecido.

Esse recurso é muito comum quando estamos trabalhando com conexão do banco de dados ou leitura/escrita de arquivos. Independente de qualquer coisa, depois de usar uma conexão, por exemplo, sempre queremos fechá-la.

9.3 MUITAS E MUITAS EXCEPTION

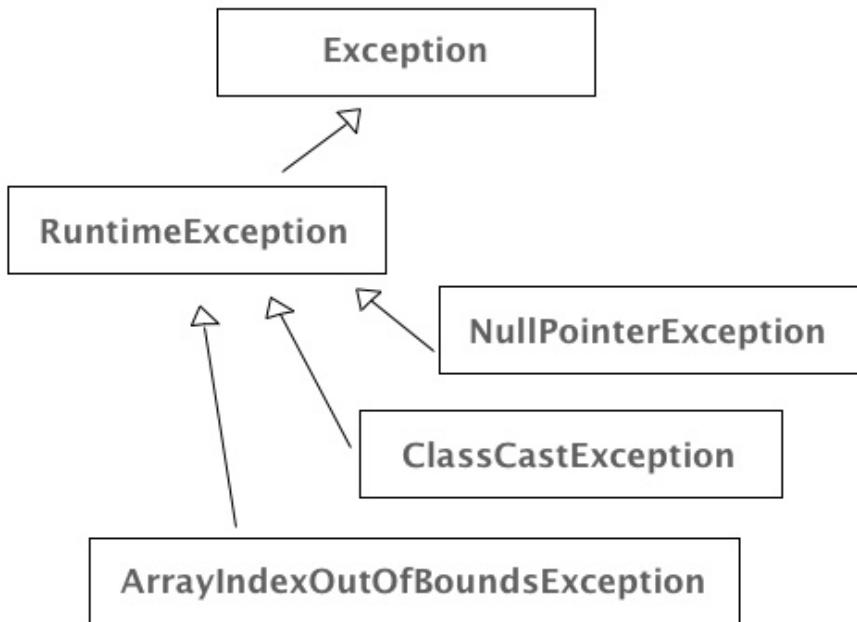
As `RuntimeExceptions`

Claro, além de um `for` tentando acessar mais índices do que deveria e a tentativa de executarmos ações em referências nulas, existem diversas outras situações de risco em que a JVM lançará uma `exception`. Cada situação dessa tem uma forma (um tipo) forte de ser representada; conhecê-las pode ser muito importante para nosso dia a dia.

Veremos diversas dessas situações e diferentes tipos de `Exception` no decorrer deste capítulo e dos demais do livro, mas desde já é interessante entendermos que todos estes casos que vimos até agora poderiam facilmente ser **evitados** de forma simples pelo programador.

Conhecemos estes tipos de `Exceptions` como `unchecked`, pois o compilador não verifica (checa) se as estamos tratando. Em outras palavras, nosso código compilará com ou sem os `try/catches` declarados.

As `unchecked exceptions` não herdam diretamente da classe `Exception`, mas sim de sua filha `RuntimeException`. Os casos mais comuns são:



Checked Exceptions

Diferente disso, quando estamos fazendo a leitura de um arquivo ou qualquer outra operação de risco que não pode ser facilmente evitada (o arquivo pode não existir, por exemplo) o compilador nos obriga a tratar esse código! Esse é um outro tipo de `exception` da linguagem, conhecida como `checked exception`.

Para testar, vamos escrever o seguinte código que realiza a leitura de um

arquivo que não existe. Não se preocupe com o código em si, entenderemos a API de IO (leitura e escrita) mais à frente. O ponto aqui é perceber que esse código não compila:

```
new java.io.FileInputStream("arquivo.txt");
```

O erro será “**Unhandled exception type FileNotFoundException**

. Note que o compilador checou que ela não está sendo tratada. Para que o código compile poderíamos fazer algo como:

```
try {
    new java.io.FileInputStream("arquivo.txt");
} catch (FileNotFoundException e1) {
    System.out.println("Não consegui abrir o arquivo");
}
```

Delegando a tratativa com `throws`

Uma alternativa para o `try/catch` nesse caso seria adicionando um `throws FileNotFoundException`. Isso significa que estamos delegando a tratativa para quem chamar este método:

```
public void abreArquivo() throws FileNotFoundException {
    new java.io.FileInputStream("arquivo.txt");
}
```

Dessa forma, o `throws` está indicando ao compilador que quem chamar o método `abreArquivo` deverá tratar o `FileNotFoundException`. Em outras palavras, isso diz: “*Ei, esse é um código de risco, ele pode lançar uma FileNotFoundException*

Chamando-o a partir de um método `main` poderíamos tratar o erro:

```
public static void main(String[] args) {
    try {
        abreArquivo();
    } catch (FileNotFoundException e1) {
        System.out.println("Não consegui abrir o arquivo");
    }
}
```

A saída será:

```
Não consegui abrir o arquivo
```

Ou ainda, usar o `throws` novamente, no próprio método `main`:

```
public static void main(String[] args)
    throws FileNotFoundException {
    abreArquivo();
}
```

Dessa forma, ninguém tratará a `exception`, estamos deixando passar para a JVM. A saída será:

```
Exception in thread "main" java.io.FileNotFoundException:
arquivo.txt (No such file or directory)
```

```
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.<init>(FileInputStream.java:131)
at java.io.FileInputStream.<init>(FileInputStream.java:87)
at br.com.casadocodigo.livraria.testes
    .RegistroDeVendas.abreArquivo(RegistroDeVendas.java:62)
at br.com.casadocodigo.livraria.testes
    .RegistroDeVendas.main(RegistroDeVendas.java:17)
```

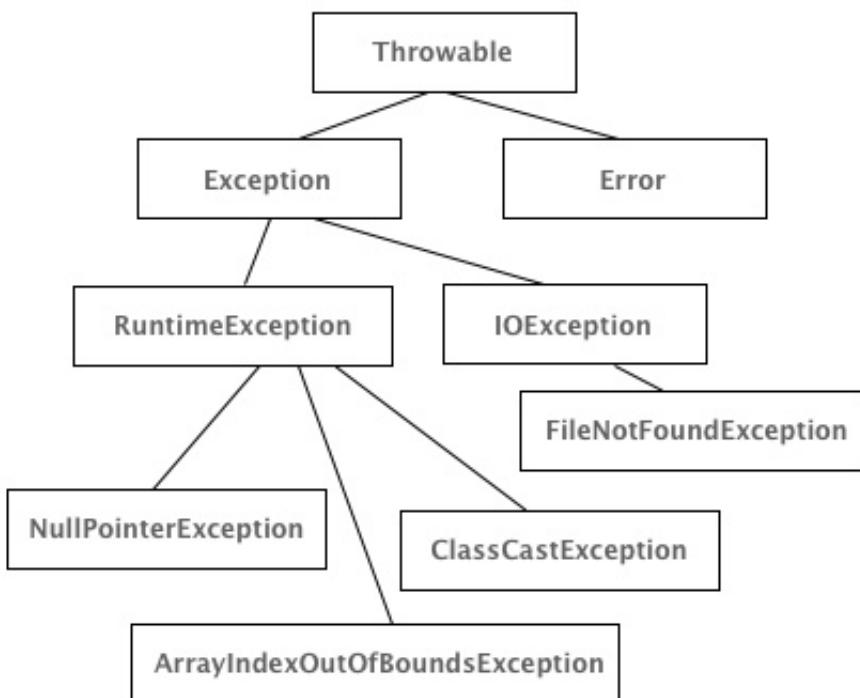
THROW OU THROWS?

É muito comum confundir no início, mas lembre-se que usamos o `throw` no imperativo quando estamos efetivamente lançando uma exception. Quando escrito no indicativo, o `throws` apenas informa ao compilador e a quem mais interessar que determinado método lança uma exception.

A família Throwable

Vimos que existem diversas Exceptions, sejam elas checked ou unchecked, porém, além delas existem os tão temidos Errors. Estes são como um tipo de exception, mas normalmente representam problemas na JVM (como o `OutOfMemoryError`). Apesar de em alguns casos conseguirmos capturar/tratar estes erros, muitas vezes o melhor a fazer é deixar a JVM encerrar sua execução. E apesar de ser possível, não devemos lançar Error em nosso código. Isso não é considerado uma boa prática.

O Error e Exception são irmãos, pois ambos herdam de `Throwable`! A imagem a seguir mostra um pouco da imensa família `Throwable`:



9.4 TAMBÉM PODEMOS LANÇAR EXCEÇÕES!

Nosso código também pode lançar uma `exception` caso algo inesperado aconteça. E não estou falando de lançar uma `exception` accidentalmente. Assim como um `array` dispara o `ArrayIndexOutOfBoundsException` quando fazemos algo inesperado (tentar acessar mais valores do que o `array` suporta), podemos fazer isso com nossas regras.

Um exemplo simples é na própria classe `Livro`, onde desde o início definimos que seria obrigatório passar um `Autor` como argumento. A regra é clara: não se pode ter um `Livro` sem `Autor` e, para garantir isso, criamos o construtor que obrigava a passagem desse parâmetro:

```
public Livro(Autor autor) {  
    this.autor = autor;  
    this.isbn = "000-00-00000-00-0";  
}
```

Isso já é bem interessante, mas o grande problema é que ainda podemos passar uma referência nula de `Autor` no momento em que criamos o livro, como:

```
Livro livro = new LivroFisico(null);
```

Até poderíamos validar o valor passado para o construtor:

```
public Livro(Autor autor) {  
  
    if (autor == null) {  
        // o que fazer aqui?  
    }  
    this.autor = autor;  
    this.isbn = "000-00-00000-00-0";  
}
```

Mas o que poderia ser feito se o autor fosse `null`? Assim como a API do Java, podemos lançar `exceptions` para casos como esse. Essa é uma forma mais robusta do que validar e retornar um `boolean` ou uma mensagem que pode simplesmente ser ignorada.

Veja como fica nosso código:

```
public Livro(Autor autor) {  
  
    if (autor == null) {  
        throw new RuntimeException();  
    }  
    this.autor = autor;  
    this.isbn = "000-00-00000-00-0";  
}
```

Repare que a palavra reservada `throw` precede a `exception` que está sendo disparada, neste caso uma `RuntimeException`. Há ainda a possibilidade de passar uma mensagem via construtor, comportamento presente na superclasse `Exception`:

```
public Livro(Autor autor) {  
  
    if (autor == null) {  
        throw new RuntimeException(  
            "O Autor do Livro não pode ser nulo");  
    }  
    this.autor = autor;  
    this.isbn = "000-00-00000-00-0";  
}
```

A saída será:

```
Exception in thread "main" java.lang.RuntimeException:  
O Autor do Livro não pode ser nulo  
    at br.com.casadocodigo.  
        livraria.produtos.Livro.<init>(Livro.java:16)  
    at br.com.casadocodigo.  
        livraria.produtos.LivroFisico.<init>(LivroFisico.java:9)  
    at br.com.casadocodigo.  
livraria.testes.RegistroDeVendas.main(RegistroDeVendas.java:16)
```

Criando sua própria `Exception`

No lugar de lançar uma `RuntimeException`, podemos criar uma `exception` bem específica para esse comportamento inesperado. Vamos criar a `AutorNuloException`!

Como fazer isso? Na verdade, é bem simples: assim como qualquer outra Exception essa será uma classe normal que herda de `Exception` ou de uma de suas filhas. Para que ela seja evitável (`unchecked`), vamos herdar de `RuntimeException`:

```
package br.com.casadocodigo.livraria.exception;

public class AutorNuloException extends RuntimeException {

    public AutorNuloException(String mensagem) {
        super(mensagem);
    }
}
```

Note que criamos um construtor que delega a mensagem para o construtor da superclasse, dessa forma nosso código poderá ficar assim:

```
public Livro(Autor autor) {

    if (autor == null) {
        throw new AutorNuloException(
            "O Autor do Livro não pode ser nulo");
    }
    this.autor = autor;
    this.isbn = "000-00-00000-00-0";
}
```

Rode novamente o código problemático e veja que a saída será:

```
Exception in thread "main" br.com.casadocodigo.livraria.exception
    .AutorNuloException: O Autor do Livro não pode ser nulo
    at br.com.casadocodigo.livraria.produtos
        .Livro.<init>(Livro.java:17)
    at br.com.casadocodigo.livraria.produtos
        .LivroFisico.<init>(LivroFisico.java:9)
    at br.com.casadocodigo.livraria.testes
        .RegistroDeVendas.main(RegistroDeVendas.java:16)
```

Você pode ler mais sobre `Exceptions` em sua documentação:

<http://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html>

Repare que logo no inicio deste arquivo temos suas `subclasses` diretas (*Direct Known Subclasses*). Achou que são muitas? E essas são apenas as filhas **diretas**, ainda temos as filhas de `RuntimeException` e de suas diversas outras `subclasses`.

Por isso, sempre antes de criar uma `Exception` própria, tente conferir se já não há uma implementação existente que represente bem a situação inesperada.

CAPÍTULO 10

Conhecendo a API

10.1 TODO OBJETO TEM UM TIPO EM COMUM

Criamos a interface `Produto` para o `CarrinhoDeCompras` suportar a adição de `Revistas` e `Livros`. Essa interface estabelece um tipo em comum entre essas duas classes, portanto o `CarrinhoDeCompras` ficou assim:

```
public class CarrinhoDeCompras {  
  
    private double total;  
    private Produto[] produtos = new Produto[10];  
    private int contador = 0;  
  
    public void adiciona(Produto produto) {  
        System.out.println("Adicionando: " + produto);  
        this.produtos[contador] = produto;  
    }  
}
```

```
    contador++;
    this.total += produto.getValor();
}

// getters para o total e array de produtos
}
```

O polimorfismo é a chave para que isso tudo funcione, nosso método adiciona `recebe um Produto` como parâmetro e o acumula no `array de Produtos`.

Essa foi uma boa estratégia, mas as duas classes não precisariam implementar nenhuma interface para ter um tipo em comum, pois sempre que criamos uma nova classe em Java ela obrigatoriamente terá uma superclasse.

Até a classe `Revista` tem uma superclasse, mas repare em sua assinatura:

```
public class Revista implements Produto, Promocional {

    // código omitido
}
```

Não há nenhum `extends` declarado nessa classe, como ela pode estar herdando de alguém? Essa é uma situação parecida com a do construtor `default`, lembra? Na ausência de um construtor, o compilador adicionará um construtor vazio para você. Assim é com a herança, na ausência de um `extends`, o Java dirá que sua classe `extends` um tipo conhecido como `Object`.

Com isso, podemos concluir que **toda classe em Java é um Object**, sem exceção. Pode não ser diretamente, mas ainda que indiretamente, ela será, pois herdará de alguém que herda diretamente ou indiretamente de `Object`.

A grande vantagem desse tipo em comum é que toda classe herda alguns métodos declarados na classe `Object` e ainda podem ser referenciadas como tal. Logo veremos alguns desses métodos e como tirar proveito deles em nosso dia a dia.

Castings e mais castings

Se toda classe já tem esse tipo em comum, por que não mudamos nosso CarrinhoDeCompras para receber e acumular Object, no lugar de Produto, como estamos fazendo? Nosso código ficaria assim:

```
public class CarrinhoDeCompras {  
  
    private double total;  
    private object[] objects = new object[10];  
    private int contador = 0;  
  
    public void adiciona(object object) {  
        System.out.println("Adicionando: " + object);  
        this.objects[contador] = object;  
        contador++;  
        this.total += object.getValor();  
    }  
  
    // getters para o total e array de objects  
}
```

Tente mudar seu código, que logo você perceberá o primeiro problema. A seguinte linha não compila:

```
this.total += object.getValor();
```

Já descobriu por quê? Object não tem o método getValor, não há garantia para o compilador de que qualquer coisa que passarmos como argumento para esse método terá esse método.

Como resolver? Bem, poderíamos fazer um casting como a seguir, moldando a referência do Object para um Produto:

```
Produto moldado = (Produto) object;  
this.total += moldado.getValor();
```

Mas, além de muito feio, esse código é muito perigoso, afinal e se o argumento passado (que agora pode ser qualquer coisa) não implementar Produto? O resultado seria um ClassCastException.

E o problema iria além. Por exemplo, para imprimir o valor dos produtos na classe `RegistroDeVendas`, estamos fazendo o seguinte código:

```
Produto[] produtos = carrinho.getProdutos();

for (int i = 0; i <= produtos.length; i++) {
    try {
        Produto produto = produtos[i];
        if (produto != null) {
            System.out.println(produto.getValor());
        }
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("deu exception no indice: " + i);
        e.printStackTrace();
    }
}
```

Mas agora que já vimos que essa não é a melhor forma de iterar em um array, podemos utilizar o `enhanced-for` do Java 7. Nossa código ficaria assim:

```
Produto[] produtos = carrinho.getProdutos();

for (Produto produto : produtos) {
    System.out.println(produto.getValor());
}
```

Muito mais simples, não acha? Só que se `produtos` fosse um `array` de `Object`, esse código não funcionaria como esperamos. Repare:

```
Object[] produtos = carrinho.getProdutos();

for (Object object : produtos) {
    System.out.println(object.getValor());
}
```

Novamente: `Object` não tem o método `getValor`, portanto, o código não compila. Poderíamos fazer o `casting`, claro, mas veja como ficaria nosso código:

```
Object[] produtos = carrinho.getProdutos();

for (Object object : produtos) {
    try {
        Produto moldado = (Produto) object;
        System.out.println(moldado.getValor());
    } catch(ClassCastException e) {
        System.out.println("O objeto passado não implementa Produto");
    }
}
```

Difícil de ler, manter e nem um pouco flexível. Você já deve ter percebido que receber um `Object` como parâmetro nem sempre é uma boa alternativa, pois sempre precisaremos de `castings` e mais `castings`, não há muito como fugir. Manter o tipo do parâmetro e `array` da classe `RegistroDeVendas` como `Produto` neste caso será uma estratégia bem mais interessante.

CASTING EM UMA SÓ LINHA

Também podemos fazer nossos `castings` em uma única linha, como no exemplo:

```
double valor = ((Produto)object).getValor();
```

O efeito do código seria o mesmo, mas talvez com uma sintaxe um pouco mais carregada custando legibilidade.

Alguns dos métodos da classe `Object`

Vimos que o custo de fazer polimorfismo com `Object` pode nem sempre valer a pena, mas isso não invalida o quanto ela é importante para a linguagem Java. Nela, estão presentes todos os métodos que toda classe em Java deve ter!

Quer um exemplo? Alguma vez você já deve ter escrito o seguinte código:

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

Note que estamos imprimindo um ebook, não seu nome, valor ou algum de seus demais atributos. Como a classe Ebook será impressa?

A saída será parecida com:

```
br.com.casadocodigo.livraria.produtos.Ebook@2f333739
```

Veja que foi impresso o fully qualified name da classe concatenado com @ e uma espécie de identificador único para a classe, seu hashCode. Não se preocupe em tentar entender o hashCode agora, logo voltaremos a falar dele. O importante neste momento é perceber que a saída será idêntica se imprimirmos dessa forma:

```
System.out.println(ebook.toString());
```

Rode o código para conferir! A saída será:

```
br.com.casadocodigo.livraria.produtos.Ebook@2f333739
```

Isso acontece porque, quando passamos um objeto para o método `println`, ele invoca seu método `toString`, portanto os dois códigos que fizemos são equivalentes.

Mas em que momento declaramos o método `toString` na classe `Ebook`? Nenhum, esse é um dos métodos que toda classe herda de `Object`. Achou que sua saída padrão não é tão interessante? Não tem problema, assim como quando herdamos de qualquer outra classe, podemos sobrescrever esse comportamento! Observe como é simples:

```
public class Ebook extends Livro implements Promocional {  
  
    // demais métodos omitidos  
  
    @Override  
    public String toString() {  
        return "Eu sou um Ebook";  
    }  
}
```

Pronto! Note que, com a anotação `@Override`, temos uma garantia de que realmente estamos modificando o comportamento da classe pai, caso

contrário, o código não compilaria. Vamos testar, basta imprimir um `ebook` novamente:

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

Agora a saída será:

```
Eu sou um Ebook
```

É sempre interessante sobrescrever esse método, porque assim conseguimos estabelecer como representar nossas classes como um texto de uma forma mais elegante do que o padrão do `Object`. Já estávamos fazendo isso com o método `mostrarDetalhes`, lembra? Podemos modificar sua assinatura para que agora seja o `toString` das classes `Livro` e `Autor`, um exemplo:

```
public abstract class Livro implements Produto {  
  
    // código omitido  
  
    @Override  
    public String toString() {  
        return "Nome: " + nome  
            + "\nDescrição: " + descricao  
            + "\nValor: " + valor  
            + "\nISBN: " + isbn;  
    }  
}
```

Comparando objetos com `equals`

Outra situação muito comum em nosso dia a dia é a necessidade de comparar objetos. Vimos nos primeiros capítulos do livro que `==` sempre compara o valor dos atributos, que em caso de objetos será sua referência. Isso significa que o resultado da seguinte comparação será `false`:

```
Autor autor = new Autor();  
autor.setNome("Rodrigo Turini");
```

```
Autor autor2 = new Autor();
autor2.setNome("Rodrigo Turini");

if (autor == autor2) {
    System.out.println("Igual");
} else {
    System.out.println("Diferente");
}
```

Execute o código para comprovar: será impressa a palavra `Diferente`. Faz sentido, já era de se esperar. Mas e se estivermos interessados em comparar os valores dos atributos? Podemos utilizar outro método muitíssimo interessante da classe `Object`, o `equals`. Mas o Java por si só não vai saber como queremos comparar nossos `Autores`, se eu apenas mudar o código para utilizar o `equals` da forma a seguir, o resultado ainda será `false`:

```
if (autor.equals(autor2)) {
    System.out.println("Igual");
} else {
    System.out.println("Diferente");
}
```

O resultado ainda será `false`, pois o método `equals` da classe `Object` faz uma comparação com `==`, assim como já estávamos fazendo. Mas, afinal, qual a vantagem de usar o `equals`? Podemos sobrescrevê-lo ensinando quais são os critérios de comparação. Um exemplo:

```
public class Autor {

    // atributos e métodos omitidos

    @Override
    public boolean equals(Object obj) {
        Autor outro = (Autor) obj;
        return this.nome.equals(outro.nome);
    }
}
```

Estamos definindo que, sempre que um `Autor` tiver um nome igual ao outro, eles são iguais; em outras palavras, o `equals` deve retornar `true`. Vamos entender melhor essa sobrescrita, a começar pela linha:

```
Autor outro = (Autor) obj;
```

Note que o `equals` recebe `Object` como argumento, portanto a primeira coisa que fizemos ao implementá-lo foi moldar esse parâmetro para o tipo `Autor`. Sim, há um risco bem grande de um `ClassCastException` caso o parâmetro passado não seja realmente um `Autor`, depois veremos algumas alternativas para lidar com isso.

Veja que logo em seguida fazemos o seguinte retorno:

```
return this.nome.equals(outro.nome);
```

Qual seria o problema de fazer o seguinte?

```
return this.nome == outro.nome;
```

Já percebeu? Claro, `String` é um objeto, logo estariamos comparando sua referência. Adiante falaremos mais sobre o problema da comparação de `Strings`, mas o importante agora é perceber que estamos delegando a comparação do nome (`String`) para o método `equals` da própria classe `String`. Ele foi sobreescrito para fazer a comparação pelo texto passado, e não pela referência de memória.

Lidando com o `ClassCastException`

Vimos brevemente que uma `ClassCastException` pode ser lançada caso o parâmetro passado para o `equals` não seja do mesmo tipo moldado, como no seguinte exemplo:

```
if(autor.equals("Rodrigo")) {  
    System.out.println("iguais");  
}
```

Esse código compilará sem nenhum problema, afinal `String` é um `Object`. Mas ao executá-lo recebemos uma

`java.lang.ClassCastException: java.lang.String cannot be cast to br.com.casadocodigo.livraria.Autor.` Como resolver? Uma maneira seria tratando com um `try/catch`, mas veja como o código fica verboso e difícil de ler:

```
@Override  
public boolean equals(Object obj) {  
  
    Autor outro;  
    try {  
        outro = (Autor) obj;  
    } catch (ClassCastException e) {  
        return false;  
    }  
  
    return this.nome.equals(outro.nome);  
}
```

Uma forma mais interessante seria utilizando o `instanceof`, como a seguir:

```
@Override  
public boolean equals(Object obj) {  
    if (!(obj instanceof Autor)) return false;  
    Autor outro = (Autor) obj;  
    return this.nome.equals(outro.nome);  
}
```

Dessa maneira, se o parâmetro passado não foi uma instância do tipo `Autor`, retornamos `false` indicando que não são objetos iguais.

Existem outros métodos interessantes na classe `Object`, alguns deles são o `getClass` e `hashCode`. Falaremos deste último no próximo capítulo, mas saiba desde já que ele sempre acompanha o `equals`. Um exemplo simples de uso do `getClass` seria para mostrar o nome simples da classe, sem seu pacote (`fully qualified name`). Repare:

```
System.out.println(autor.getClass().getSimpleName());
```

10.2 WRAPPERS DOS TIPOS PRIMITIVOS

Já não é nenhuma surpresa que o seguinte código compile:

```
Object objeto = new Autor("Rodrigo");
```

Nem mesmo se for apenas uma `String`, pois **todo objeto em Java herda diretamente ou indiretamente de `Object`.**

```
Object objeto = "Uma String";
```

Mas o que você acha do código a seguir?

```
Object objeto = 10;
```

Se você estiver utilizando uma versão maior que Java 1.4, ele compilará. Isso não significa que os tipos primitivos como o `int` herdam de `Object` ou são do mesmo tipo que uma referência, esse é apenas um truque do compilador adicionado no Java 1.5 para simplificar um pouco a sintaxe. Esse recurso é conhecido como `autoboxing`.

Sem isso, para que o código compile precisaríamos fazer:

```
Object objeto = new Integer(10);
```

Note que a classe `Integer` é usada para embrulhar (`wrapping`) o tipo primitivo `int`, de forma que podemos tratá-la como uma referência, um `Object`. O mesmo se aplica aos demais tipos primitivos. A imagem a seguir mostra cada um deles e sua classe `wrapper` (`box`).

Tipo Primitivo	Classe Wrapper
boolean	Boolean
byte	Byte
short	Short
char	Character
int	Integer
float	Float
long	Long
double	Double

Também podemos desembrulhar os valores primitivos das classes wrappers:

```
Integer integer = new Integer(10);  
int valor = integer.intValue();
```

Mas essa não é a única utilidade das classes wrappers, elas estão repletas de métodos estáticos que podem ser bastante úteis para manipular os tipos primitivos.

Conversão de valores

Como transformar um boolean em uma String? E o contrário? Como transformar um texto em um double válido? Todas essas conversões podem ser feitas de forma bem simples a partir das classes que embrulham esses

tipos primitivos. Por exemplo, o método `parseBoolean` da classe `Boolean` transforma uma `String` em um booleano. Veja:

```
boolean resultado = Boolean.parseBoolean("false");
```

O mesmo pode ser feito para os demais tipos:

```
byte parseByte = Byte.parseByte("1");
short parseShort = Short.parseShort("10");
int parseInt = Integer.parseInt("10");
long parseLong = Long.parseLong("10");
float parseFloat = Float.parseFloat("10.0");
double parseDouble = Double.parseDouble("10.0");
```

Para transformar qualquer um desses valores em uma `String`, podemos utilizar seu método estático `valueOf`:

```
String numeroEmTexto = String.valueOf(10);
```

Claro, precisamos tomar bastante cuidado com o valor passado, pois adivinhe qual será o resultado do código a seguir?

```
int parseInt = Integer.parseInt("ABC");
```

Uma `Exception!` Neste caso: `java.lang.NumberFormatException`: For input string: "ABC". Faz bastante sentido, afinal, como transformar ABC em um número?

MAIS MÉTODOS ÚTEIS

Aproveite o momento para explorar os diversos métodos existentes em `Object` e nas classes *wrappers*, como a `Integer`. Eles com certeza serão bem úteis em seu dia a dia codando em Java!

10.3 O PACOTE JAVA.LANG

Você já deve ter percebido que até agora não precisamos fazer `import` de nenhuma das novas classes que vimos neste capítulo. Outro detalhe é que em

nenhum momento anterior precisamos fazer `import` da classe `String` ou mesmo da classe `System`, não é?

Isso porque `Object`, `String`, `System`, classes wrappers dos tipos primitivos entre diversas outras estão todas presentes no pacote padrão do Java, o `java.lang`. Este pacote é automaticamente incluído em todas as suas classes, diferente de todos os demais da linguagem.

Nele você encontrará classes como a `Math`, que tem um arsenal completo de métodos que o auxiliam em operações numéricas como arredondamento (`round`), mínimo (`min`), máximo (`max`), seno (`sin`), entre muitos outros. Alguns exemplos de uso:

```
long round = Math.round(3.99);
long max = Math.max(100, 10);
int min = Math.min(100, 10);
int abs = Math.abs(-5);
double sqrt = Math.sqrt(4);
```

Os valores das variáveis, respectivamente, serão:

```
4
100
10
5
2.0
```

Você pode ver a documentação completa dessa classe em:

<http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>

Outra classe muito útil em nosso dia a dia é a `Random`, veja como é fácil imprimir um número aleatório de 0 a 10:

```
Random random = new Random();
System.out.println(random.nextInt(10));
```

Da mesma forma como o `nextInt`, também existe um método `next` para cada tipo primitivo, como o `nextBoolean` ou `nextDouble`.

java.lang.String

Para fechar nosso passeio pelo pacote padrão do Java, vamos entender um pouco melhor a classe `String`. É muito comum confundi-la com um tipo primitivo, mas `Strings` guardam referências a objetos. A prova disso é que podemos instanciá-la da seguinte forma:

```
String java = new String("Java");
```

Assim como qualquer outra variável de referência, não devemos compará-la utilizando o operador `==`, mas sim o `equals` que foi reescrito na classe `String` para comparar cada caractere do texto. Portanto, o resultado do código a seguir será `true`:

```
String java = "java";
String java2 = "java";

System.out.println(java.equals(java2));
```

Mas algo bastante inesperado acontecerá ao compararmos duas `Strings` desta forma:

```
String java = "java";
String java2 = "java";

System.out.println(java == java2);
```

Aqui, o resultado também será `true`! Por quê? Isso acontece pois, por otimização, o Java cria um `String pool`, um tipo de cache de `Strings`. Sempre antes de adicionar em memória, a JVM consulta esse `pool` para verificar se não há uma `String` igual que possa ser reutilizada.

Para que isso funcione bem, o Java não poderia permitir que a mudança no valor de uma `String` afetasse outras `Strings` que tivessem seu mesmo valor. Por isso, **toda String é imutável**.

Quer uma prova? Tente substituir o valor de uma `String` com seu método `replace`. É bem simples, o primeiro argumento será o valor atual e o segundo o valor que deverá tomar o seu lugar, por exemplo:

```
java.replace("v", "c");
System.out.println(java);
```

É natural esperar que o resultado impresso seja “jaca”, uma vez que substituímos o caractere “v” por “c”, mas rode o código para ver o resultado:

```
java
```

É isso mesmo, o valor da variável `java` continuou o mesmo, já que toda `String` é imutável. Para obter o resultado esperado, podemos resgatar o retorno do método `replace`, que será uma nova `String`. Repare:

```
String novaString = java.replace("v", "c");
System.out.println(novaString);
```

O mesmo vale para todo método que aplica transformações da `String`, eles sempre retornam uma referência nova para o valor transformado.

Agora que já conhecemos essa característica fundamental, podemos usar os seus mais de 60 métodos. Alguns exemplos são:

```
String replace = java.replace("v", "c");
String upperCase = java.toUpperCase();
String lowerCase = "JAVA".toLowerCase();
char charAt = java.charAt(0);
boolean endsWith = java.endsWith("a");
boolean startsWith = java.startsWith("s");
boolean equals = java.equalsIgnoreCase("JAVA");
```

Os valores das variáveis, respectivamente, serão:

```
jaca
JAVA
java
j
true
false
true
```

Dominar o pacote padrão do Java é um grande diferencial. Muitas vezes já há algo implementado para nos ajudar com situações do dia a dia. Em

outras palavras, o pacote `java.lang` com certeza fará parte de sua rotina. Aproveite para estudar seus diversos outros métodos e classes. Você pode encontrar a documentação completa em:

<http://docs.oracle.com/javase/8/docs/api/java/lang/package-summary.html>

CAPÍTULO 11

Collection Framework

11.1 O TRABALHO DE MANIPULAR ARRAYS

A princípio, deixamos nosso `CarrinhoDeCompras` limitando o total de `Produtos` em 10 itens. Um número fixo e nem um pouco real. Uma forma simples de flexibilizar a quantidade de `Produtos` seria forçando a passagem do array de `Produtos` pelo construtor:

```
public class CarrinhoDeCompras {  
  
    private double total;  
    private Produto[] produtos;  
    private int contador = 0;  
  
    public CarrinhoDeCompras(Produto[] produtos) {  
        this.produtos = produtos;  
    }  
}
```

```
    }  
  
    // outros métodos da classe  
}
```

Isso torna seu uso um pouco mais interessante. No momento de criar um `CarrinhoDeCompras`, passamos a quantidade de `Produtos` que poderá ser adicionada, como:

```
CarrinhoDeCompras carrinho =  
    new CarrinhoDeCompras(new Produto[ 10 ]);
```

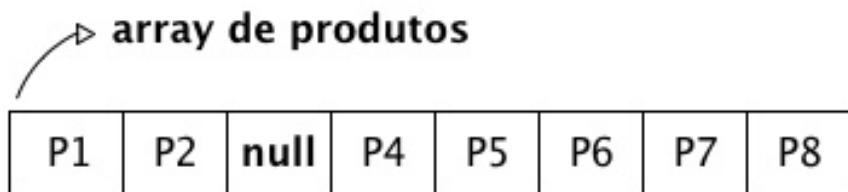
Mas como saber exatamente se teremos 2, 10, ou 100 `Produtos` neste carrinho? Não há um valor fixo, afinal o cliente poderá comprar quantos produtos ele bem entender. O ponto é que a necessidade de passar um número fixo no momento de instanciar um `array` não ajuda muito neste momento. E o tamanho de um `array` nunca muda; depois de criado, ele terá aquela capacidade e ponto, nunca poderá ser redimensionado.

Poderíamos, sim, no momento de adicionar novos produtos verificar se há espaço livre no `array`, e caso não haja, criar um novo com mais espaço e mover os objetos do antigo para ele. Mas isso seria muito trabalhoso, não acha? E não é o único trabalho que teríamos ao utilizar um `array`.

Outra necessidade seria remover produtos do `CarrinhoDeCompras`. Uma implementação simples do método `remove` seria:

```
public void remove(int posicao) {  
    this.produtos[posicao] = null;  
}
```

Ao atribuir `null` para determinada posição, ganhamos uma série de problemas. Repare como ficará nosso `array` após removermos o elemento de posição 2:



E agora, como faremos ao adicionar novos itens? Estávamos inserindo sequencialmente, mas precisaremos considerar as possíveis posições vazias no meio do `array`.

Uma possibilidade seria criar um outro `array`, guardando as posições livres, e consultá-lo antes de inserir um novo `Produto`. Outra possibilidade seria sempre reordenar o `array` depois de remover um item. Mas repare bem, todas elas tornariam nosso código mais complexo e difícil de manter.

Manipular um `array` é bastante trabalhoso. Essas foram apenas algumas das muitas questões que apareceriam em nosso código. Por esse e outros motivos, o Java 2 1.2 introduziu um conjunto de classes e interfaces bastante conhecido como **Collections Framework**. Essa robusta *API* traz estruturas bem mais interessantes para lidar com os mais diferentes tipos de dados.

ArrayList versus array

No lugar de trabalhar diretamente com `arrays`, podemos utilizar uma `ArrayList` para representar a multiplicidade de `Produtos` e nos auxiliar nas operações necessárias em nosso `CarrinhoDeCompras`. Essa classe é uma das introduções da *API* de `Collections`, veja a seguir como seu uso é simples.

```
ArrayList lista = new ArrayList();
String valor = "conhecendo uma ArrayList";
lista.add(valor);
System.out.println(lista.contains(valor));
lista.remove(valor);
```

```
System.out.println(lista.contains(valor));
```

A saída será:

```
true  
false
```

O `ArrayList` tem um conjunto de métodos que, na maior parte do tempo, representam bem as nossas necessidades. O método `remove` pode agora receber o próprio valor a ser removido, mas também tem uma sobre-carga que recebe a posição que deve ser removida. Você pode escolher qual opção lhe atender melhor.

Um ponto muito interessante é que o `ArrayList`, assim como as demais classes da API de `Collection`, trabalham de forma genérica. Caso contrário, haveria uma `ArrayList` para cada tipo de dado que precisássemos adicionar, como uma `String`, um `int`, ou mesmo uma data. Todos os seus métodos trabalham com `Object`.

Portanto, o seguinte código compila:

```
ArrayList lista = new ArrayList();  
lista.add(10);  
lista.add("uma string");  
lista.add(new Revista());
```

Mas já vimos o problema de trabalhar com `Object` dessa forma: será necessário um `casting` sempre que precisarmos recuperar um valor dessa lista. Por exemplo:

```
int valor = (int) lista.get(0);
```

E se o valor do índice zero não fosse um `int`? Como saber exatamente o tipo de cada elemento em cada posição da lista? Trabalhar dessa forma não seria nem um pouco simples, não acha? E na maior parte do tempo, queremos ter uma lista de um único tipo, e não vários como fizemos agora. Por exemplo, nosso `CarrinhoDeCompras` deverá ter uma lista de `Produtos`.

Para auxiliar nesse trabalho, desde o Java 5 podemos restringir o tipo de objetos de uma determinada lista utilizando um recurso conhecido como `Generics`. Repare:

```
ArrayList<Produto> produtos = new ArrayList<Produto>();
```

A sintaxe pode parecer estranha no começo, mas note que a única mudança é que agora estamos indicando ao compilador que essa lista deve trabalhar com `Produtos`. Há dois grandes ganhos aqui, o primeiro será pela não necessidade de um *casting* ao recuperar valores desta lista. Veja:

```
Produto buscado = produtos.get(0);
```

O código fica bem mais limpo sem os *castings* e evitamos as recorrentes `ClassCastException`s. O outro ganho será a segurança de que não conseguiremos adicionar nada que não seja do tipo `Produto` nessa lista. O seguinte código não compila:

```
produtos.add("Eu não sou um produto");
```

DIAMOND OPERATOR DO JAVA 7

Desde o Java 7, no lugar de passar o tipo genérico duas vezes como fizemos ao instanciar a lista de `Produtos`, podemos utilizar um recurso conhecido como *diamond operator* (operador diamante), deixando o código assim:

```
ArrayList<Produto> produtos = new ArrayList<>();
```

Esse recurso era bastante limitado, funcionava basicamente se a declaração fosse feita na mesma linha. No Java 8, a inferência de tipos foi bastante melhorada e, como um efeito colateral, agora podemos fazer códigos como este:

```
public class CarrinhoDeCompras {  
  
    private ArrayList<Produto> produtos;  
  
    public CarrinhoDeCompras() {  
        this.produtos = new ArrayList<>();  
    }  
}
```

Note que só estamos instanciando o `ArrayList` no construtor da classe, mas como o compilador sabe inferir bem qual o tipo do `this.produtos`, conseguimos utilizar o diamante sem nenhum problema.

Veja como fica nossa classe `CarrinhoDeCompras` utilizando uma lista:

```
public class CarrinhoDeCompras {  
  
    private double total;  
    private ArrayList<Produto> produtos;  
  
    public CarrinhoDeCompras() {  
        this.produtos = new ArrayList<>();  
    }
```

```
}

public void adiciona(Produto produto) {
    this.produtos.add(produto);
}

public void remove(int posicao) {
    this.produtos.remove(posicao);
}

public double getTotal() {
    return total;
}

public ArrayList<Produto> getProdutos() {
    return produtos;
}
}
```

Muito mais simples, não acha? Não precisamos de um atributo `contador`, não temos que limitar o tamanho do `array`, entre outras desvantagens comentadas.

É interessante reconhecer que um `ArrayList` não é um `array`! Essa confusão é bastante comum, mas na verdade `ArrayList` usa um `array` internamente para armazenar os dados, mas isso está bem *encapsulado*. Em nenhum momento precisaremos recuperar seu `array` interno para fazer operações mais complexas, não temos acesso a ele e nem deveríamos.

O `ArrayList` resolve todos os problemas que comentamos do `array`. Utilizá-lo, com certeza, será uma melhor opção. Mas existem outros tipos de lista que não utilizam necessariamente um `array` internamente. Um exemplo é a `LinkedList` (lista ligada); enquanto a `ArrayList` é mais rápida para fazer pesquisas (iterar), a `LinkedList` é mais rápida para inserir e remover elementos de suas pontas.

Cada estrutura tem suas vantagens, vale a pena conhecer as diferenças entre elas para tomar uma boa decisão no momento de utilizar uma lista.

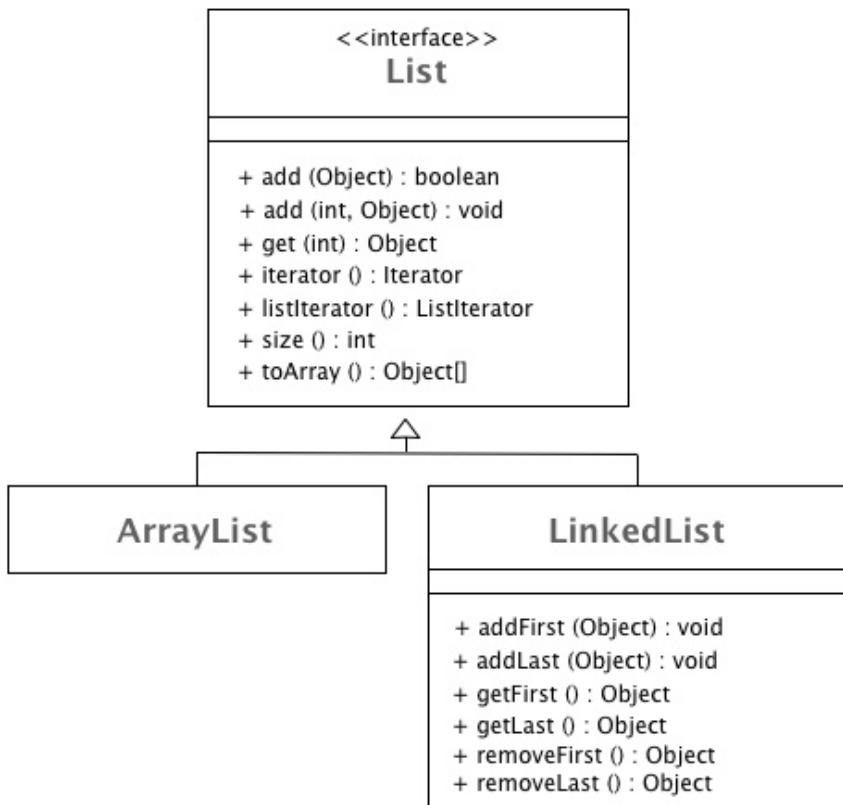
O que todas elas têm em comum é a interface `List`, do pacote `java.util`. Para não deixar nosso código dependendo de um tipo de lista

específico (**acoplamento**), já que podemos a qualquer momento mudá-las, é uma boa prática programar voltado para a interface. Repare como fica nossa classe CarrinhoDeCompras:

```
public class CarrinhoDeCompras {  
  
    private double total;  
    private List<Produto> produtos;  
  
    public CarrinhoDeCompras() {  
        this.produtos = new ArrayList<Produto>();  
    }  
  
    public void adiciona(Produto produto) {  
        this.produtos.add(produto);  
    }  
  
    public void remove(int posicao) {  
        this.produtos.remove(posicao);  
    }  
  
    public double getTotal() {  
        return total;  
    }  
  
    public List<Produto> getProdutos() {  
        return produtos;  
    }  
}
```

Note que, agora que o tipo `ArrayList` aparece apenas no momento de instanciar a lista, em todos os demais pontos nos referimos a ela como uma `List<Produto>`. Isso torna nosso código muito mais flexível para mudanças. Para utilizar um `LinkedList`, bastaria mudar o tipo em que estamos dando `new` sem causar nenhum efeito colateral indesejado ao restante do código.

A imagem a seguir mostra a interface `List` e algumas de suas implementações:



Para finalizar nossa migração de `array` para `List`, precisamos corrigir o erro de compilação da classe `RegistroDeVendas`, no momento em que estamos iterando pelos itens do carrinho:

```
Produto[] produtos = carrinho.getProdutos();

for (Produto produto : produtos) {
    System.out.println(produto);
}
```

A única mudança necessária será no tipo de retorno do método `getProdutos`. Basta mudar de `Produto[]` para `List<Produto>` e todo o restante continuará igual.

```
List<Produto> produtos = carrinho.getProdutos();  
  
for (Produto produto : produtos) {  
    System.out.println(produto);  
}
```

Isso significa que o `enhanced-for` também funciona com qualquer tipo de `List` (na verdade, com qualquer `Iterable`, como veremos mais à frente).

11.2 ORDENANDO NOSSA LIST DE PRODUTOS

Ao imprimir os valores dessa lista, você perceberá que serão exibidos na mesma ordem em que foram inseridos. Mas é fato que, sempre que estamos trabalhando com listas, entre outros tipos de coleções, é natural a necessidade de exibir seus dados de forma ordenada seguindo algum critério.

Até o Java 7, podíamos utilizar o método estático `sort`, presente na classe `java.util.Collections`:

```
List<String> nomes = new ArrayList<>();  
nomes.add("Rodrigo Turini");  
nomes.add("Adriano Almeida");  
nomes.add("Paulo Silveira");  
  
Collections.sort(nomes);  
  
System.out.println(nomes);
```

Executando esse código, o resultado será:

```
[Adriano Almeida, Paulo Silveira, Rodrigo Turini]
```

É importante perceber que o método `sort` efetivamente modificou a estrutura interna da lista, neste caso deixando-a em ordem alfabética.

A classe `Collections` possui diversos outros métodos que podem ser muito úteis quando você está trabalhando com coleções, principalmente com Java 7 ou menor. Logo veremos que o Java 8 introduziu diversos recursos para tornar seu trabalho com coleções ainda mais prático e funcional.

Alguns outros exemplos de métodos da classe `Collections` são: o `reverse`, para inverter a ordem dos elementos da lista; o `copy`, para copiar os elementos de uma lista para outra; e também o `emptyList`, que retornará uma lista vazia. Você pode ver os diversos métodos dessa classe em sua detalhada documentação:

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

O `sort` em nossa `List<Strings>` utilizou como critério de ordenação a ordem alfabética. Mas qual seria o resultado do seguinte código?

```
List<Produto> produtos = new ArrayList<>();  
  
// populando a lista com alguns produtos  
  
Collections.sort(produtos);
```

Esse código não compila! Observe a saída:

```
The method sort(List<T>) in the type Collections  
is not applicable for the arguments (List<Produto>)
```

Faz sentido, afinal qual seria o critério de ordenação? Precisamos ensinar ao método `sort` qual elemento deve vir antes de qual, ou seja, como ordenar elementos do tipo `Produto`.

O método `sort` precisa receber uma lista de elementos comparáveis, que possuam um método específico ensinando-o como deve comparar tais elementos. Para garantir que os elementos passados tenham esse método, mais uma vez a API do Java fez uso de uma interface! Todo elemento ordenável deve implementar a interface `java.lang.Comparable`, que possui o método `compareTo`.

Para que nosso código compile, podemos dizer que todo `Produto` implementa `Comparable`, mas repare no código a seguir:

```
public interface Produto extends Comparable<Produto> {  
  
    double getValor();  
}
```

Pode parecer um pouco estranho na primeira vez que você olhar, afinal estamos utilizando `extends`, e não o `implements`. Já percebeu por quê? Como `Produto` é uma interface, ela apenas herdará as obrigações da interface `Comparable`. Já as classes que a implementarem agora além da obrigação de escrever seu método `getValor` terão também que implementar `compareTo`, o que logo veremos.

Apenas um último detalhe antes de partirmos para a implementação do método: você notou que o `Comparable` também faz uso do *generics* do Java 5? Se não tinha notado, dê uma boa olhada na assinatura da interface, que tem um `Comparable<Produto>`. Com isso, eliminamos a necessidade de fazer o casting de `Object`, que é o tipo do parâmetro de seu método `int compareTo(Object)`.

Pronto, finalmente podemos partir para a implementação, que poderá ser feita na classe abstrata `Produto` e, portanto, replicada a suas subclasses `LivroFisico`, `Ebook` e `MiniLivro`. O critério de comparação será seu valor:

```
public abstract class Livro implements Produto {  
  
    // código omitido  
  
    @Override  
    public int compareTo(Produto outro) {  
  
        if (this.getValor() < outro.getValor()) {  
            return -1;  
        }  
        if (this.getValor() > outro.getValor()) {  
            return 1;  
        }  
        return 0;  
    }  
}
```

O mesmo pode ser feito na classe `Revista`, que não é filha de `Livro` mas também é um `Produto` que deve ser ordenado por valor. O código pareceu confuso? Não se preocupe, vamos entendê-lo um pouco melhor.

Note que o método `compareTo` retorna um `int`. Devemos retornar o (zero) se o objeto comparado (parâmetro `outro`) for igual a este objeto (`this`), um número negativo se este objeto for menor (deva vir antes quando ordenado) que o objeto passado, ou um número positivo caso seja maior (deva vir depois).

Execute o seguinte código para ver o resultado em prática:

```
public class OrdenandoComJava {  
  
    public static void main(String[] args) {  
  
        Autor autor = new Autor();  
        autor.setNome("Rodrigo Turini");  
  
        LivroFisico fisico = new LivroFisico(autor);  
        fisico.setNome("Java 8 Prático");  
        fisico.setValor(59.90);  
  
        Ebook ebook = new Ebook(autor);  
        ebook.setNome("Java 8 Prático");  
        ebook.setValor(29.90);  
  
        List<Produto> produtos = Arrays.asList(fisico, ebook);  
  
        Collections.sort(produtos);  
  
        for (Produto produto : produtos) {  
            System.out.println(produto.getValor());  
        }  
    }  
}
```

A saída será:

```
29.9  
59.9
```

OUTRAS IMPLEMENTAÇÕES

Como estamos comparando apenas o valor do Produto, poderíamos simplificar nossa implementação retornando apenas a subtração de um valor pelo outro. Repare como nosso código fica mais simples:

```
public abstract class Livro implements Produto {  
  
    // código omitido  
  
    @Override  
    public int compareTo(Produto outro) {  
        return this.getValor() - outro.getValor();  
    }  
}
```

Mais simples, não acha? Alternativamente, também poderíamos utilizar o método `Integer.compare` passando os dois valores como parâmetro. Experimente utilizar alguma dessas estratégias para deixar seu código mais simples, essa é uma prática bastante comum.

11.3 GERENCIANDO CUPONS DE DESCONTO

Agora que nossa classe `CarrinhoDeCompras` já está bem resolvida, podemos seguir para uma outra necessidade de nosso projeto. Precisamos agora gerenciar cupons de desconto promocionais.

Como ponto de partida, podemos criar a classe `GerenciadorDeCupons` que possui internamente uma lista com alguns cupons já cadastrados. Repare:

```
public class GerenciadorDeCupons {  
  
    private List<String> cupons;  
  
    public GerenciadorDeCupons() {
```

```
    this.cupons = Arrays.asList("CUP74", "CUP158",
        "CUP14", "CUP52", "CUP21", "CUP221", "CUP91",
        "CUP327", "CUP410", "CUP275", "CUP484", "CUP207",
        "CUP96", "CUP119", "CUP174", "CUP291", "CUP1",
        "CUP115", "CUP222", "CUP272");
    }
}
```

Note que setamos o método estático `asList` da classe `java.util.Arrays` para criar nossa lista. Poderíamos criar um `ArrayList` e chamar seu método `add` para cada elemento, mas é comum utilizar esse novo método para simplificar esse processo. O método `asList` é uma fábrica (*factory*) de `List` e, com certeza, será muito útil em seu dia a dia.

Nosso próximo passo será criar um método para validar os cupons passados pelo cliente. Ele retornará um `boolean` indicando se o cupom passado está ou não presente em nossa `List<String>` `cupons`. Vamos chamá-lo de `validaCupom`:

```
public boolean validaCupom(String cupom) {
    return this.cupons.contains(cupom);
}
```

Note que utilizamos o já conhecido método `contains` para fazer essa consulta. Bem simples, não acha?

Já estamos prontos para testar esse código. Vamos criar uma nova classe chamada `ConsultaDeDescontos`, que deve utilizar esse novo método, o `validaCupom`. Repare:

```
public class ConsultaDeDescontos {

    public static void main(String[] args) {

        GerenciadorDeCupons gerenciador =
            new GerenciadorDeCupons();

        if(gerenciador.validaCupom("CUP1234")){
            System.out.println("Cupom de desconto valido.");
        }
    }
}
```

```
        // aplica o desconto desse cupom
    } else {
        System.out.println("Esse cupom não existe.");
    }
}
```

Rode esse código para ver o resultado. Como o cupom não existe em nossa lista, a saída será:

Esse cupom não existe.

Trabalhar com uma `List` aqui resolve o problema, mas ela não é a estrutura ideal para fazer esse trabalho. Normalmente, trabalhamos com uma lista quando seus elementos internos podem repetir e sua ordem tem alguma importância, mas note que não é esse o caso de nossos cupons. Um cupom é único, não há repetições e, além disso, a ordem deles não importa, afinal estamos utilizando esses dados apenas para consulta.

Em casos como este, podemos utilizar uma outra estrutura de dados bastante interessante, um conjunto (`java.util.Set`).

`java.util.Set`

Um conjunto (ou `Set` em Java) funciona de forma parecida com os conjuntos da matemática. Ela é uma coleção, assim como a `List`, mas em que não há repetição de seus dados internos. Além disso, sua ordem não é necessariamente a ordem em que os valores foram inseridos, isso pode variar bastante de cada implementação.

Assim como a `List`, o `Set` é apenas uma interface. Para utilizar um `Set`, precisamos instanciar alguma de suas implementações, como a `HashSet`, que é uma das implementações mais usadas. Observe como fica nosso código da classe `GerenciadorDeCupons`:

```
public class GerenciadorDeCupons {

    private Set<String> cupons;

    public GerenciadorDeCupons() {
```

```
this.cupons = new HashSet<String>();  
  
cupons.addAll(Arrays.asList("CUP74", "CUP158",  
    "CUP14", "CUP52", "CUP21", "CUP221", "CUP91",  
    "CUP327", "CUP410", "CUP275", "CUP484", "CUP207",  
    "CUP96", "CUP119", "CUP174", "CUP291", "CUP1",  
    "CUP115", "CUP222", "CUP272"));  
}  
  
public boolean validaCupom(String cupom) {  
    return this.cupons.contains(cupom);  
}  
}
```

Note que utilizamos o método `addAll` para facilitar essa mudança. Mas é natural adicionar elementos no `Set` chamando seu método `add`, assim como na `List`. Outro detalhe importante é que continuamos programando voltados para a interface, isto é, apesar de dar `new` em `HasSet`, o tipo do atributo `cupons` é um `Set`. Essa é e sempre será uma excelente prática, posto que diminui o acoplamento de nossa classe e facilita evoluções no futuro.

Nosso código continua funcionando assim como antes. Rode a classe `ConsultaDeDescontos` para confirmar! Aparentemente, tudo está igual, mas na verdade algumas coisas mudaram.

O primeiro ponto importante a se perceber é que, mesmo que você adicione mais 500 mil cupons iguais nesse conjunto, ao mandar imprimir o seu tamanho (`size`) ele ainda será 20. Você pode rodar o código a seguir para confirmar isso:

```
HashSet<String> set = new HashSet<String>();  
set.add("Não há repetição em Conjuntos");  
System.out.println(set.size());
```

Qual foi o valor da saída? Exatamente 1, ele ignorou as repetições. Para ajudar a saber quando você está inserindo uma repetição em um `Set`, seu

método `add` retorna um `boolean`, que será `false` nesses casos.

Outra vantagem muito importante do `Set` é que existem implementações, como o próprio `HashSet` que já estamos utilizando, que apresentam uma performance muito melhor que a da `List` para fazer consultas (com o método `contains`, por exemplo).

Em um conjunto tão pequeno como o nosso, não haverá diferença notável, mas se quiser perceber a diferença, execute o seguinte teste:

```
public class TestandoPerformance {  
  
    public static void main(String[] args) {  
  
        List<String> colecao = new ArrayList<String>();  
  
        for (int i = 0; i < 100000; i++) {  
            colecao.add("Item"+i);  
        }  
  
        long inicio = System.currentTimeMillis();  
  
        for (int i = 0; i < 100000; i++) {  
            colecao.contains("Item"+i);  
        }  
  
        long fim = System.currentTimeMillis();  
        long tempo = fim - inicio;  
  
        System.out.println("Demorou "+ tempo + " MS para executar");  
    }  
}
```

O resultado pode variar bastante de acordo com o hardware, mas utilizando o `List` em minha máquina, a saída foi:

```
Demorou 31216 MS para executar
```

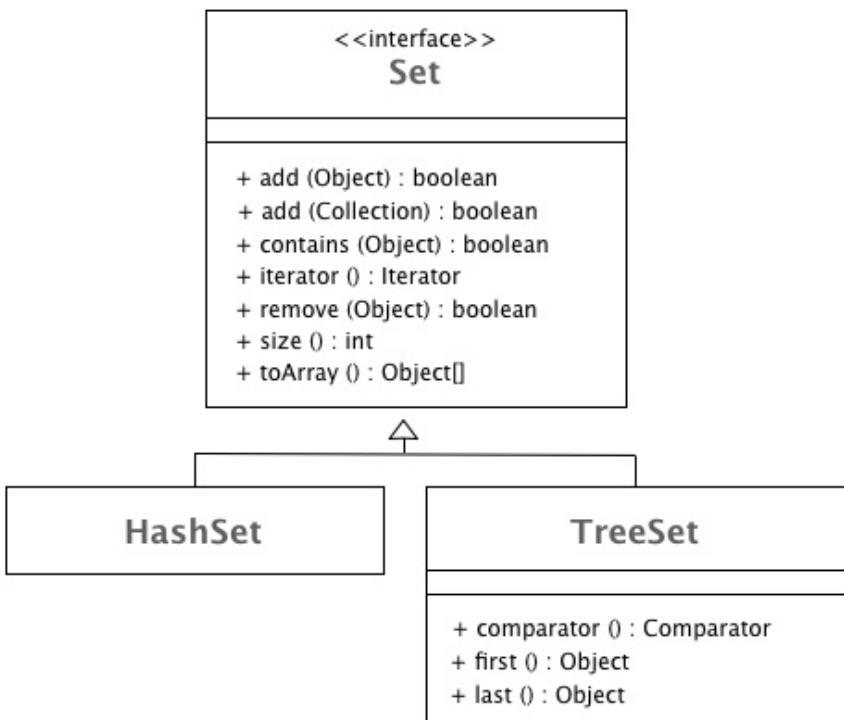
Agora mude para um `HashSet` para perceber a diferença. O mesmo código com `Set` apresentou o seguinte resultado:

Demorou 42 MS para executar

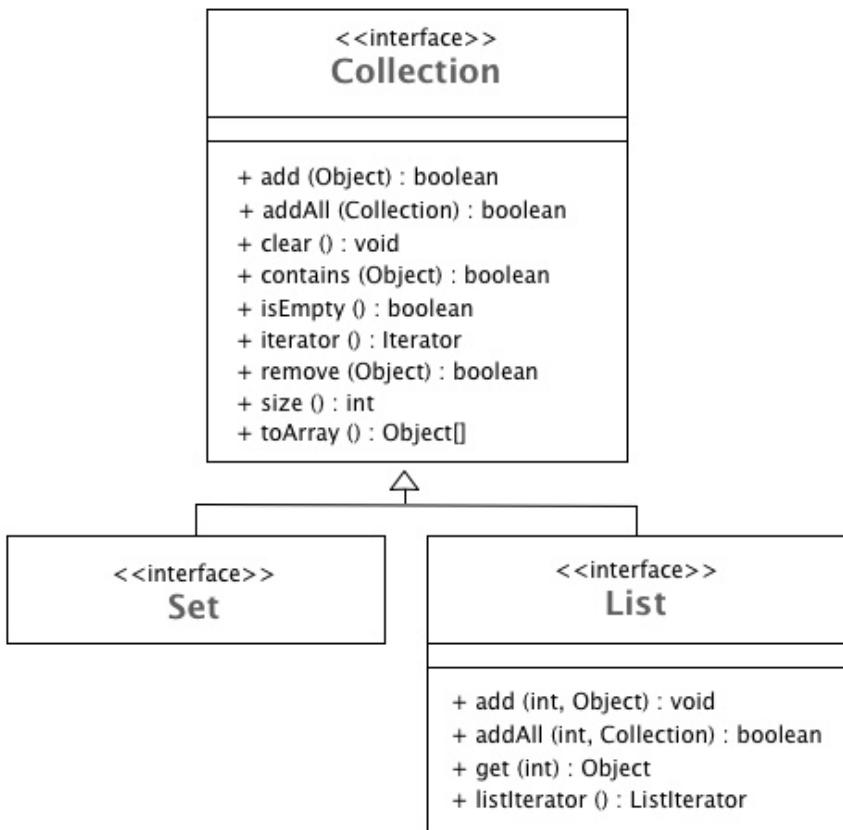
Impressionante! O segredo do ganho para consultas com o `HashSet` está no uso do `hashCode` e `equals`. Você pode ler mais sobre esses métodos em:

<http://blog.caelum.com.br/ensinando-que-e-o-hashcode/>

A imagem a seguir ilustra a interface `Set` e algumas de suas implementações:



Todas essas estruturas têm como base a interface `Collection`. Nela, estão definidos os métodos essenciais para quando estamos trabalhando com coleções. Você pode ver alguns deles na imagem a seguir:



11.4 JAVA.UTIL.MAP

Nosso GerenciadorDeCupons já está quase completo. O próximo passo será aplicar o desconto específico para determinado cupom que foi validado. Mas como vincular um cupom único com o seu valor de desconto?

Essa é uma necessidade muito comum: vincular um objeto a uma chave única, para conseguir buscar esse valor rapidamente. Poderíamos resolver isso com uma lista, mas já vimos que essa não é a melhor estrutura quando queremos fazer buscas.

Para casos como esse você pode utilizar uma outra estrutura de dados muito útil, um Mapa (`java.util.Map`). É muito comum confundir isso, afinal ela sempre anda de mão dadas com as estruturas da API de `Collection`, mas um `Map` não estende a interface `Collection`. Essa estrutura também é bastante conhecida como dicionário em outras linguagens.

Um mapa é composto por um conjunto (`Set`) de chaves associadas a um objeto valor. Sua sintaxe pode parecer um pouco intimidadora no começo, mas não há nada de tão complicado. Veja como podemos criar um `HashMap`, que é sua implementação mais comum:

```
Map<String, Double> mapa = new HashMap<>();
```

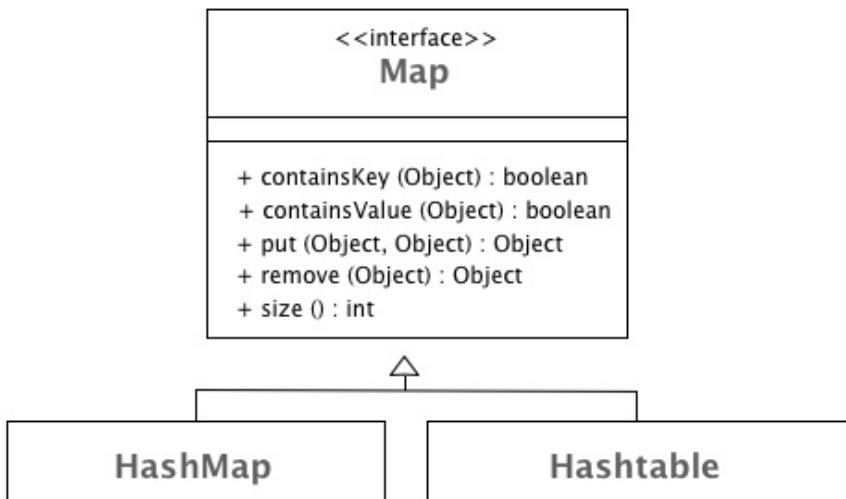
Note que agora precisamos passar os dois tipos genéricos: chave e valor. Nesse caso, declaramos a chave como `String` para um valor do tipo `Double`. Podemos inserir valores neste mapa utilizando seu método `put`, como a seguir:

```
mapa.put("CUP982", 5.99);
```

Simples, não é? E para recuperar seus valores, podemos utilizar seu método `get`. Passando a chave (`String`) como parâmetro, ele nos retornará o seu valor associado (`Double`). Repare:

```
Double valor = mapa.get("CUP982");
System.out.println(valor);
```

A saída será `5.99`. Operações de busca e consulta também são muito performáticas nessa estrutura. A imagem a seguir demonstra suas principais implementações:



Agora que conhecemos um pouco dessa estrutura, podemos utilizá-la em nosso `GerenciadorDeCupons`:

```
public class GerenciadorDeCupons {

    private Map<String, Double> cupons;

    public GerenciadorDeCupons() {

        this.cupons = new HashMap<>();

        cupons.put("CUP74", 10.0);
        cupons.put("CUP158", 15.00);
        cupons.put("CUP14", 5.99);
        cupons.put("CUP52", 20.00);

        // ...
    }

    public boolean validaCupom(String cupom) {
```

```
        return this.cupons.containsKey(cupom);
    }
}
```

Observe que agora utilizamos o método `containsKey` para verificar se o cupom passado existe. Também poderíamos utilizar seu método `containsValue` para verificar a existência de algum de seus valores.

Para deixar nosso método `validaCupom` um pouco mais interessante, podemos utilizar o método `get` para já retornar o valor do desconto que deve ser aplicado caso a chave passada exista:

```
public Double validaCupom(String cupom) {
    return this.cupons.get(cupom);
}
```

Podemos agora modificar nossa classe `ConsultaDeDescontos` para recuperar esse valor ou, caso nulo, exibir a mensagem dizendo que o cupom não é válido.

```
Double desconto = gerenciador.validaCupom("CUP74");

if(desconto != null){
    System.out.println("Cupom de desconto valido.");
    System.out.println("Valor "+ desconto);
} else {
    System.out.println("Esse cupom não existe.");
}
```

Execute a classe mais uma vez. A saída deve se parecer com:

```
Cupom de desconto valido.
Valor 10.0
```

Excelente! Agora podemos facilmente realizar operações com esse valor associado ao desconto.

CAPÍTULO 12

Streams e novidades do Java 8

A API de Collections é bastante robusta, sem dúvidas. Há muito o que explorar em suas diversas estruturas e soluções. Mas o fato é que estamos falando de uma API gigante e com mais de 15 anos, que com certeza possui algumas limitações e características que não poderiam ser evoluídas sem que houvesse uma sobrecarga ou ainda quebra de compatibilidade (veremos no próximo capítulo o quanto isso é importante para o Java).

Por esse e outros motivos, o Java 8 introduziu várias novidades que nos trazem uma forma bem mais interessante e funcional de trabalhar com nossas coleções. Veremos algumas delas durante este capítulo.

12.1 ORDENANDO COM JAVA 8

Vimos que uma maneira de ordenar a `List<Produto>` era tornando seus elementos comparáveis, implementando `java.lang.Comparable`. Mas e

se você tiver mais de um critério de ordenação? E se você quiser em alguns momentos comparar um `Livro` pelo seu `valor` e em outros pelo seu `nome`?

Até o Java 7, uma forma de fazer isso seria criando uma nova classe, um comparador de `Livro` por nome. Essa classe precisava implementar a interface `java.util.Comparator`, algo como:

```
public class ComparadorPorNome implements Comparator<Livro>{

    @Override
    public int compare(Livro l1, Livro l2) {
        return l1.getNome().compareTo(l2.getNome());
    }
}
```

O método `sort` da `Collection` tem uma sobrecarga que recebe um `Comparator` como parâmetro, portanto agora basta fazer:

```
Collections.sort(livros, new ComparadorPorNome());
```

Vamos ver esse código na prática! Crie a classe `NovidadesDoJava8` e adicione o seguinte trecho para testar:

```
public class NovidadesDoJava8 {

    public static void main(String[] args) {

        Autor autor = new Autor();
        autor.setNome("Rodrigo Turini");

        Livro javaoo = new LivroFisico(autor);
        javaoo.setNome("Java 0.0.");

        Livro java8 = new LivroFisico(autor);
        java8.setNome("Java 8 Prático");

        Livro ruby = new LivroFisico(autor);
        ruby.setNome("Livro de Ruby");

        List<Livro> livros = Arrays.asList(javaoo, java8);
```

```
Collections.sort(livros, new ComparadorPorNome());  
  
    for (Livro livro : livros) {  
        System.out.println(livro.getNome());  
    }  
}  
}
```

Agora execute o código e repare que a saída será:

```
Java 8 Prático  
Java 0.0.  
Livro de Ruby
```

Classes anônimas e o lambda

O que pode incomodar bastante dessa solução é que, para cada novo critério de ordenação, precisaríamos criar uma nova classe como a `ComparadorPorNome`. Ainda que seu código seja bem pequeno e não seja reutilizado em nenhum outro lugar.

Uma alternativa bastante utilizada até o Java 7 eram as conhecidas classes anônimas. Você pode dar `new` em uma interface, mas terá que implementar seus métodos ali mesmo, na mesma instrução. Podemos fazer isso com a interface `Comparator`. No lugar de criar a classe `ComparadorPorNome`, podemos fazer:

```
Collections.sort(livros, new Comparator<Livro>() {  
    @Override  
    public int compare(Livro l1, Livro l2) {  
        return l1.getNome().compareTo(l2.getNome());  
    }  
});
```

Isso mesmo, é o mesmo código da classe `ComparadorPorNome`, mas declarado em uma única instrução. Esse recurso é conhecido como classe anônima, pois afinal essa classe não tem nem mesmo um nome! Achou a sintaxe estranha? Feia? Difícil de ler? Bem... eu concordo! Isso polui bastante o nosso código.

Para simplificar esse processo, o Java 8 introduziu algumas mudanças, uma das quais foi adicionar o método `sort` na própria interface `List`! Pode parecer uma mudança simples, mas agora podemos fazer:

```
livros.sort(new Comparator<Livro>() {
    @Override
    public int compare(Livro l1, Livro l2) {
        return l1.getNome().compareTo(l2.getNome());
    }
});
```

Para não quebrar a compatibilidade da interface `List`, esse método `sort` é um método concreto, com código dentro! Lembra do nome desse recurso? No capítulo de interface brevemente comentei sobre os tais `default methods`, eis um bom exemplo:

```
default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
}
```

Isso mesmo, esse método `defalt` da `List` apenas delega a chamada para o método `sort` de `Collections`, que já fazia bem este trabalho. Mas isso ainda não resolve todos os problemas de ordenação, não é? Então vamos para outra novidade.

Também no capítulo de interfaces, vimos brevemente que o Java 8 introduziu o conceito de **interfaces funcionais**. Em poucas palavras, uma interface com um único método abstrato pode ser considerada funcional, como é o caso da interface `Comparator`, que possui apenas o método `compare`. Com isso, ganhamos algumas possibilidades, como transformar aquela classe anônima em uma **expressão lambda**! Uma primeira forma de fazer isso seria:

```
livros.sort((Livro l1, Livro l2) -> {
    return l1.getNome().compareTo(l2.getNome());
});
```

Note que a diferença desse código para a classe anônima é que deixamos apenas os parâmetros do método `compare`, seguidos do operador `->` e sua

implementação. Como o método `sort` espera receber um `Comparator`, o compilador já infere que esse é o tipo da expressão lambda.

Podemos simplificar ainda mais esse código, retirando o tipo `Livro` dos parâmetros, as chaves, o `return` e até mesmo o ponto e vírgula. Repare:

```
livros.sort(  
    (l1, l2) -> l1.getNome().compareTo(l2.getNome())  
);
```

Esse é um conceito um pouco mais avançado e, com certeza, pode parecer bastante estranho ou mesmo assustador no início. O código `(l1, l2) -> l1.getNome().compareTo(l2.getNome())` resulta em uma instância de `Comparator` que devolve `l1.getNome().compareTo(l2.getNome())`. Não há necessidade de declarar o tipo (`Livro`) dos parâmetros, o próprio compilador sabe inferir isso para você. Não há necessidade da palavra `return` e nem mesmo de chaves, já que temos uma única instrução após o operador `->`.

Para tornar o código ainda mais enxuto e um pouco mais fluente, o método `default comparing` foi adicionado na interface `Comparator`. Ele é uma fábrica (`factory`) de `Comparator`, tudo que precisamos fazer é passar uma expressão lambda com o critério de comparação como a seguir:

```
livros.sort(comparing(l -> l.getNome()));
```

Note que, como há apenas um parâmetro nesse lambda, não precisamos passá-lo dentro de parênteses. Rode o código para ver o resultado! O que acha?

Simplificando ainda mais com `method reference`

A expressão lambda `l -> l.getNome()` apenas diz ao `comparing` qual método deverá ser utilizado em sua comparação. Ele está apenas referenciando um método, neste caso o `getNome`. Em casos como este, podemos ainda fazer:

```
livros.sort(comparing(Livro::getNome));
```

Esse é um outro recurso do Java 8, conhecido como `method reference`. Note que sua sintaxe é o nome da classe seguido de `:>:` e o nome do método. Esse recurso pode ser utilizado sempre que temos uma expressão lambda como a `l -> l.getNome()`, que apenas delega a chamada de um método.

12.2 FOREACH DO JAVA 8

Outro `default method` que, com total certeza, fará parte de seu dia a dia é o `forEach`, presente na interface `Iterable`. Observe a forma como estamos iterando pela lista de livros utilizando o `enhanced-for`:

```
for (Livro livro : livros) {  
    System.out.println(livro.getNome());  
}
```

No Java 8, agora podemos fazer:

```
livros.forEach(l -> System.out.println(l.getNome()));
```

O método `forEach`, assim como a maior parte dos `default methods`, recebe uma interface funcional como parâmetro. Portanto, note que foi possível utilizar uma expressão lambda para representar nossas intenções.

O lambda `l -> System.out.println(l.getNome())` nada mais diz do que: “para cada livro, chame o método `println` imprimindo seu nome”.

12.3 FILTRANDO LIVROS PELO AUTOR

Além da forma de ordenar e iterar, o Java 8 mudou bastante a forma de fazer algumas operações rotineiras de quando estamos trabalhando com coleções. Quer um exemplo prático?

O trabalho de filtrar uma Collection

Dada uma lista de livros, queremos filtrar apenas os que tenham a palavra `Java` em seu nome. Até o Java 1.7, uma das formas mais tradicionais de se

fazer isso seria criando uma nova lista para o resultado e condicionando os elementos que deveriam ser inseridos:

```
List<Livro> filtrados = new ArrayList<>();  
  
for (Livro livro : livros) {  
    if (livro.getNome().contains("Java")) {  
        filtrados.add(livro);  
    }  
}
```

Para testar, adicione esse código na classe `NovidadesDoJava8` e logo depois mostre os resultados dessa lista, fazendo um simples `for`:

```
for (Livro livro : filtrados) {  
    System.out.println(livro.getNome());  
}
```

Note que a saída será:

```
Java 8 Prático  
Java 0.0.
```

Excelente, resolvemos o problema! Mas dê uma boa olhada em seu código... o que achou? Repare que, além de ser muito verboso, ele exigiu a criação de uma lista intermediária para o resultado. Esse é sem dúvida um código bastante imperativo.

Operações comuns como essa, filtrar elementos de uma coleção, estão agora presentes em uma nova API, conhecida como `Stream`. O `Stream` traz para o Java uma forma mais funcional de trabalhar com as nossas coleções, usando uma interface fluente! Separando as funcionalidades do `Stream` da `Collection`, também ficou mais fácil deixar claro que métodos são mutáveis, evitar problema de conflito de nome de métodos, entre outros.

Stream e o filter

Como criar um `Stream`? Isso é bem simples, um novo método `default` chamado `stream` foi definido na interface `Collection`. Com isso, basta fazer:

```
Stream<Livro> stream = livros.stream();
```

A partir desse nosso `Stream<Livro>` conseguimos utilizar, por exemplo, o método `filter`:

```
livros.stream().filter(l -> l.getNome().contains("Java"));
```

Estamos fazendo a mesma coisa que aquele `for` com um `if` dentro do Java 7, só que agora de uma forma muito mais declarativa!

Não deixe de testar, mude o código da classe `NovidadesDoJava8` e escreva o seguinte `for` para exibir a lista de `livros`:

```
for (Livro livro : livros) {  
    System.out.println(livro.getNome());  
}
```

Rode o código e o resultado será:

```
Java 8 Prático  
Java 0.0.  
Livro de Ruby
```

Ops, esse `Livro de Ruby` não deveria estar no resultado! O filtro não funcionou? Na verdade, funcionou, mas a API de `Streams` é imutável, isso é, não altera sua coleção inicial. Você pode fazer quantas transformações quiser sem se preocupar com efeitos colaterais em sua lista.

Portanto, no lugar de iterar na lista original podemos utilizar o método `forEach` também presente na API de `Streams` da seguinte forma:

```
livros.stream()  
    .filter(l -> l.getNome().contains("Java"))  
    .forEach(l -> System.out.println(l.getNome()));
```

Agora sim, ao executar o código teremos apenas os elementos filtrados:

```
Java 8 Prático  
Java 0.0.
```

Claro, o `filter` é apenas um dos zilhares métodos presentes nessa nova API. Você pode ver a lista completa e alguns exemplos em:

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

Vimos neste capítulo apenas um pouco de muito que o Java 8 introduziu com seus novos recursos e APIs. O seguinte post do blog da Caelum fala um pouco mais sobre essas novidades:

<http://blog.caelum.com.br/o-minimo-que-voce-deve-saber-de-java-8/>

O LIVRO JAVA 8 PRÁTICO

Junto com Paulo Silveira, escrevi o livro *Java 8 Prático: Lambdas, Streams e os novos recursos da linguagem*, que entra a fundo em cada detalhe e sumariza os novos recursos do Java. Se você gostou das novidades, com certeza vai gostar de seu conteúdo:

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

CAPÍTULO 13

Um pouco da história do Java

13.1 ORIGEM DA LINGUAGEM

Já colocamos em prática muitos dos conceitos da orientação a objetos e recursos específicos da linguagem Java, mas para concluir essa primeira parte do estudo é fundamental conhecermos um pouco de sua história.

Acredito que, quando foi iniciado o *Green Team*, um projeto da *Sun Microsystems* cujo objetivo era criar uma plataforma de computação interativa, ninguém imaginou que esse seria o início de uma linguagem que mudou o rumo da história da programação e que atualmente possui mais de 9 milhões de desenvolvedores. Em 1992, foi feita a primeira demonstração desse projeto, mas apenas em 1995 foi anunciado o lançamento oficial da plataforma Java.

Na época, o uso fundamental da linguagem era focado em navegadores web para rodar pequenas aplicações (os tão conhecidos *applets*), tanto que foi inserida no *Netscape Navigator*, que era o principal navegador do momento.

Desde então, a grande ideia da linguagem é você escrever uma aplicação e executá-la nos diferentes dispositivos existentes. Ou seja, portabilidade é uma das características mais fortes e presente desde suas origens. É claro que essa não é a única característica forte da linguagem, que também é robusta, paralelizável, segura e, dentre diversas outras que ainda serão melhor detalhadas, estável.

Algumas das características que tornam o Java muito bem vindo como linguagem são sua estabilidade e retrocompatibilidade. Você pode reparar isso pela numeração de suas versões: repare que apesar da estratégia de marketing de chamá-lo de Java 8, por exemplo, a versão continua sendo 1.8. Ou seja, o 1 sempre está presente indicando que ainda não houve uma quebra de compatibilidade desde seu lançamento.

Podemos, em 2014, com a versão mais recente da linguagem, compilar e executar uma aplicação escrita por volta de 1995 com Java 1.0.2 sem nenhuma dificuldade. Isso nos dá uma boa segurança de que uma atualização de versão não será um problema, por exemplo.

Há uma comunidade conhecida como JCP (*Java Community Process*) cujo objetivo é garantir esse padrão de estabilidade e compatibilidade multiplataforma da tecnologia Java. Você pode ler mais sobre a JCP e suas propostas em:

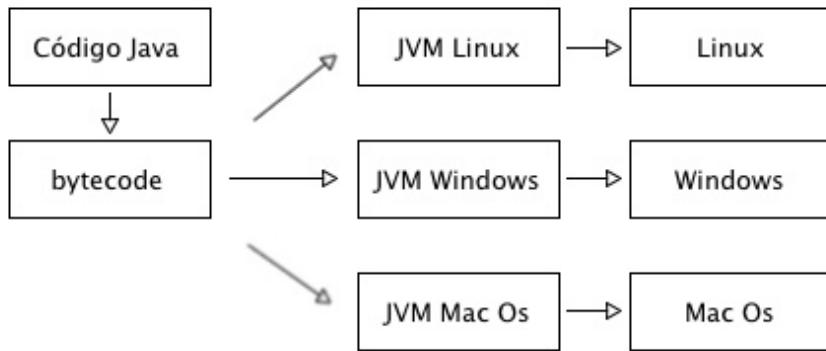
<https://www.jcp.org/en/introduction/overview>

13.2 ESCREVA UMA VEZ, RODE EM QUALQUER LUGAR!

Diferente das demais linguagens, uma aplicação Java pode ser executada em qualquer um dos diferentes sistemas operacionais existentes, como Windows, Linux ou Mac OS. Essa possibilidade abre muitos caminhos e foi um dos principais fatores que tornaram a linguagem tão atraente para o mercado. Exibindo esse benefício, o Java teve como slogan oficial o termo *Write once, run anywhere* (escreva uma vez, rode em qualquer lugar).

A grande chave para essa portabilidade é a máquina virtual, ou JVM (*Java Virtual Machine*). No lugar de instruções nativas para um determinado hardware, após compilado, um código-fonte em Java é traduzido para um formato conhecido como *bytecode*. Esse *bytecode* independe da arquitetura do sistema

em que foi gerado e, portanto, poderá ser executado em qualquer plataforma, contanto que ela tenha uma JVM instalada. A imagem a seguir representa esse processo:



Inicialmente, o desempenho da linguagem foi um fator preocupante, tendo como parâmetro de comparação as demais linguagens compiladas em instruções nativas para uma determinada plataforma. Ao ganhar em portabilidade, a linguagem sofreu um grande impacto por adicionar essa camada intermediária. Em pouco tempo, o desempenho deixou de ser um problema, com introduções e tecnologias que serão melhor detalhadas a seguir nas descrições das diferentes versões da plataforma.

13.3 LINHA DO TEMPO

A linguagem foi, sem dúvida, um caso de sucesso. Em sua primeira versão, tinha cerca de 250 classes e agora conta com mais de 4.200, ricas APIs, poderosos recursos e é executada por mais de 3 bilhões de dispositivos. Para entendermos melhor todo esse sucesso, precisamos conhecer um pouco de sua história, alguns de seus principais acontecimentos e introduções.

Java 1

Em 1991, a linguagem foi originalmente chamada de *Oak* (carvalho, em inglês) por influência da vista na janela do escritório de seu criador, James Gosling. Somente a partir de sua primeira versão estável, que foi a 1.0.2 em 1995, passou a ser chamada de Java 1.

Existiam diversos bugs nesse momento e a performance era bastante problemática.

Java 1.2

Com mais de 2 milhões de downloads em sua versão atual, em 1998 foi lançado o Java 1.2. Nessa versão, diversos bugs foram corrigidos e a performance foi consideravelmente melhorada. Um dos principais fatores responsáveis por essa melhora foi a introdução do compilador JIT (*Just In Time*) em que a compilação é feita durante a execução do programa (em *runtime*) em vez de antes de sua execução.

Outra introdução muito importante dessa versão foi a API de *Collections*, que será profundamente estudada mais adiante.

JAVA 2

Por uma estratégia de marketing da época, a linguagem passou a se chamar Java 2 (independente de sua versão, por exemplo, a versão 1.3 era conhecida como *Java(TM) 2 Platform, Standard Edition version 1.3*). Apenas em 2004, na versão 1.5, que o 2 deixou de fazer parte do nome da linguagem.

Java 1.3

Em 2000, a mais nova versão do Java foi lançada, o Java 1.3. Alavancando ainda mais sua performance, uma JVM conhecida como *HotSpot* passou a ser a máquina virtual padrão da Sun. Essa JVM combinava interpretação de código e compilação JIT (em tempo de execução).

O seu código Java ainda era inicialmente interpretado, mas durante a execução do programa a JVM passou a identificar os pontos executados com

maior frequência (conhecidos como pontos quentes, ou *HotSpots*) e quando necessário transformava esses códigos em instruções nativas da plataforma que são executadas diretamente no hardware.

Java 1.4

Chegando a quase 3 mil classes, a API foi enriquecida com introduções como expressões regulares (*Regular Expression*), processamento de XML, *Logging* e muito mais em sua versão 1.4, lançada em 2002. Essa foi a primeira versão do Java desenvolvida no âmbito da JCP (*Java Community Process*).

Java 1.5

Sem dúvida uma das versões mais importantes, o Java 1.5 foi lançado em setembro de 2004. Dentre diversas outras, *Generics*, *imports* estáticos, anotações (*metadata*) e *boxing* de tipos primitivos foram algumas de suas novidades.

Essa versão também ficou bastante conhecida por introduzir uma forma mais sucinta de iterar em coleções, o *enhanced for* que já conhecemos.

Uma nova estratégia de marketing surge. Para melhor refletir o nível de maturidade, estabilidade, escalabilidade e segurança da plataforma, o Java 1.5 deixou de ser Java 2 e passou a ser conhecido como Java 5. Mas claro, não estamos falando de uma quebra de compatibilidade, a versão continua sendo 1.5.

Desde então, as versões da linguagem passaram a ser conhecidas pelo seu último número.

Java 1.6

A fim de impulsionar ainda mais a adoção da tecnologia, a Sun tornou o Java *open source*. Ou seja, seu código foi aberto para comunidade com a licença GNU (*General Public License*), a mesma utilizada pelo sistema operacional Linux. Além disso, diversas otimizações no core da plataforma e na JVM marcaram a versão 1.6.

Java 1.7

Quatro anos após sua ultima versão, essa foi a primeira versão após a aquisição da Sun pela Oracle. O Java 7 foi lançado sob uma nova perspectiva, teve como novidades uma nova API de I/O, recursos como o operador diamante (*diamond operator <>*) e o *try-with-resources*.

Java 8

A mais atual versão da plataforma apresentou mudanças impactantes para a linguagem. Seu lançamento aconteceu em março de 2014, depois de 3 anos de muita espera. Diversos recursos foram introduzidos nessa versão, *default methods*, expressões lambdas e *method reference*, que vimos brevemente no capítulo anterior, foram algum deles.

A versão também conta com uma nova API de datas baseada na conhecida biblioteca *Joda Time*, além de uma forma mais funcional de trabalhar com suas coleções com a API de *Stream*.

Vimos que até então seu código que era escrito dessa forma:

```
List<Usuario> usuariosFiltrados = new ArrayList<>();
for(Usuario usuario : usuarios) {
    if(usuario.getPontos() > 100) {
        usuariosFiltrados.add(usuario);
    }
}

Collections.sort(usuariosFiltrados, new Comparator<Usuario>() {
    public int compare(Usuario u1, Usuario u2) {
        return u1.getNome().compareTo(u2.getNome());
    }
});

for(Usuario usuario : usuariosFiltrados) {
    System.out.println(usuario);
}
```

Agora, com Java 8, pode ser escrito de forma muito mais declarativa e funcional:

```
usuarios.stream()
    .filter(u -> u.getPontos() > 100)
    .sorted(comparing(Usuario::getNome))
    .forEach(System.out::println);
```

Você pode ler mais sobre a história da plataforma e ver a *timeline* completa em:

<http://www.java.com/en/javahistory/>

<http://oracle.com.edgesuite.net/timeline/java/>

CAPÍTULO 14

Continuando seus estudos

Não deixe de praticar cada exercício proposto durante o Livro e sempre ir além com novos testes e comentários mais complexos.

Há diversas formas de continuar seus estudos, uma das quais é o GUJ. Não apenas para postar suas dúvidas, mas também respondendo aos demais usuários e sendo um membro ativo da comunidade Java do Brasil. Ensinar é uma das mais efetivas formas de aprender. Espero vê-lo ensinado e aprendendo por lá:

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

Outro caminho natural para continuar seus estudos é o livro *Java 8 Prático: Lambdas, Streams e os novos recursos da linguagem*. Nele, explicamos detalhadamente desde a sintaxe até o uso prático de cada novidade da mais esperada versão do Java.

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

14.1 ENTRE EM CONTATO CONOSCO

Encaminhe suas dúvidas ou crie tópicos para discussão na lista que foi criada especialmente para este livro:

<https://groups.google.com/d/forum/livro-java-oo>

Além de perguntar, você também pode contribuir com sugestões, críticas e melhorias para nosso conteúdo. Todas serão muito mais do que bem-vindas.