Desbravando Java e Orientação a Objetos

Um guia para o iniciante da linguagem

Casa do Código

RODRIGO TURINI

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

http://www.caelum.com.br/apostila-java-orientacao-objetos/apendice-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

. A !

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, MeuPrimeiroPrograma . Em Java, o nome de uma classe sempre se inicia com letra maiúscula e, quando necessário, as palavras seguintes também têm sua case alterado. Dessa forma asse nome de classe vira •EsseNomeDeClasse . Essa abordagem é bastante conhecida como CamelCase

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

!

" # # $% &

$ ' (' $ $ ') ' $*$!+

,

,

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

#

Nesse exemplo, o resultado impresso será:

% & *

A mensagem de erro será:

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

. T $ % 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 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

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:

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

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

Você pode e deve usar e abusar desse recurso!

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 cará 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`.

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.

. A -

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, criticas 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/

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

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 (número com ponto utuante). Logo, conheceremos os outros tipos existentes, mas é importante reconhecer desde já que toda variável em Java precisara ter um.

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

```
    ! "#$#%
! "#$#%
```

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:

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:

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

```
! "#$ " %

!& '( ) ' %
*
*
```

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

```
( ) ++
```

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

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

Declarar da seguinte forma:

. C

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.

!

" # $% %&
' $% %&

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

```
!           "   # $
```

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:

```
$          %   #   &    '

   $       #           ()*        (   '

      # $                +,
      # $              +,

      # $
```

Mesmo arredondando o valor dos livros para um número inteiro (neste caso,), 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:

```
# $                +,
      -
```

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

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

. A

NossaCalculadoraDeEstoque precisa de uma nova funcionalidade. Se o valor total de livros for menor que 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:

```
                    !"#"$
%           !"#"$

                          &     %
```

!

"

"

"

"

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

Passamos uma condição como argumento fazendo uma comparação entre o valor da variávelsoma com o valor . Essa condição vai resultar em um valor true quando verdadeira, oufalse caso contrário. Esse tipo de condição é conhecido comexpressão booleana. Seu resultado sempre será do tipo boolean :

$

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 signi ca atribuição) e, por m,!= (diferente).

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

```
% %       & %  #  ' (

    %    % ) #            *+,         *

      #        ) - ) .       $  / /
      #        ) 0"      $  / /

      #             $   ) - ) .          ) 0"
```

!

" #$ %

!

"

"

"

"

O '

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

```
&      '      $
    '   # '%
     '      $
"
     '      $
"
```

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

```
&      '      $ '   # '% (     )
```

Sem dúvida, ca bem mais enxuto, não acha? Performaticamente, não há nenhuma vantagem signi cante 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.

!

O while é bastante parecido com if . A grande diferença é que, enquanto sua expressão booleana for true , seu código continuará sendo executado. Neste exemplo estamos criando uma variável total 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 livros, já que a condição contador < 35 e que a cada iteração incrementamos ao valor contador .

Nosso código completo fica da seguinte forma:

" # $ %

" & '() '

!

```
                                                    !
        "                   #$ %
                                                    !
        "

        "
    "
"
```

Ao executá-lo a saída será:

```
                % &'        (
                   !
```

L (

É muito comum esquecermos de modi car 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 in nito, uma vez que a condição de looping sempre será verdadeira. Se quiser testar, adicione um comentário na linha que incrementa o valor do contador e execute o código:

```
)*      +  ,        -
    ,       .  / 0 .       $  & &
          $         .  / 0 .
    11 +  ,  $ +  ,
"
```

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

```
    $         .  / 0 .
```

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

Podemos utilizar essa técnica com os seguintes operadores:

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

O código do while agora fica da seguinte forma:

!

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

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

!"!

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:

```
#               $       % &  '

    #           (               )   *+,         *


                        !"!
```

!

" #$ %

!

"

"

"

"

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

Neste caso, apenas os números de a serão impressos.

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

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:

O valor . 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.

U (

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.

. C

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.

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

!

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`.

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:

Neste caso, a saída será:

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:

```
                    !     "

                              #$%        #   "

          !              &  '   !
                &
                   &           (               # #
                 & )* *+
                &        *, - )-../)+-0.-.
```

```
     1
1
```

Execute a classe para ver a saída! Deverá ser:

```
(                # #
)* *
*, - )-../)+-0.-.
```

! " #
$
%& &'
&(!)!%)**+%'),*)*

-
-
-
-

. " /0

%& &'
&(!)!%)**+%')++)'

-
-
-
-

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).

. C "

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 linhas de código:

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 livros. Para cada livro, teremos linhas de código imprimindo os seus atributos. Ou seja, teremos 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 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:

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

```
! "
```

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:

```
# $%                    $
```

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:

```
&
```

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

```
                    !"

                  #
                $   % & '
                     (
)* *+
                *,%-%)-../)+-0.-.



                     #
                    1       &      23

                 )* *+
                     *,%-%)-../)+-//-+
```

! " #
$$
%

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

```
& ' ( )
                * *
 +, ,
! "  ,-'$'+$../+0$1.$.
$$
```

```
23*     ( *
   4            *
 +, ,
! "  ,-'$'+$../+0$//$0
$$
```

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

```
                    !                   "
                      #$ !                      "
                %     !                    "
                & ' !                     "
                ((
        )
)
```

Mas repare que todas essas novas informações pertencem ao livro e não necessariamente ao livro. Sendo o autor do livro 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:

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. Nosso código fica assim:

```
! "                    #
    $% "                    #
&   "                    #
' (!"                    #
))
```

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

```
*   +
*     ,     -
*                .
*     /01 234 567 /8
```

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

```
*
```

! "

#

$ %&' ()* +,- %.

/ , 0 1

2

)- -.

-+,3,)3**&).3(*3*

4

0

#

$ %&' ()* +,- %.

5 0 67

)- -.

-+,3,)3**&).3&&3.

4

8

8

```
                                    !
                                        "
                                #$% &'( )*+ #,

                    $
            $                       !
            $                           "
            $                #$% &'( )*+ #,

    -
-
```

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

```
                    $
    .             /    0          1
-
    .             2   13  4    5          3
-
```

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

```
2   13  4    5          3
```

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

Achou complicado? Não se preocupe, durante a leitura faremos muitas atribuições e relacionamentos entre objetos. A nal, 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:

```
! "                    #


$
%&'                    (
))
*
```

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:

```
$
%&'                    (
```

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

```
#              +

       ,
       ,
       ,   (


                  $
```

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 de nidos, 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:

```
!
"
    #$
%
& '"                    (

))
```

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

```
!
"    *    +    ,
    #$    "                - -
%      ./ /
& '"   /0+)+.)112.3)41)1
!
"     5    -   6
7          -       8          (
    92: 4.1 0+/ 93
))
!
"     ;<-          -    #$
    #$                      -
%      ./ /
& '"   /0+)+.)112.3)22)3
!
```

```
" "     #            $% "  $&'(       & )  *

+              , $ -  +   %).
          ,     .

/ "    "   $" $%01     "    0              2          ).

          !,                  3    .

/ "    "   $" $%01       #   $"   0              2          ).
4
```

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

```
      !,                  3    .
```

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

```
1     "
1       #   $"
```

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 livros e eu aplique desconto em todos eles. A nossa lógica de aplicar desconto vai car repetida e espalhada por todo o código.

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

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

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

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 `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:

E a classe `Livro` completa:

! " #

$$

%

&

$' (&

%

%

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:

```
)  *
      +  ,

%
```

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

```
#  *
   --  .  ) /  . 0
%
```

Mas como saber se `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

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

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:

```
  !                "
  # $% !                "    # #
& !                 "
' ( !                "

)


**
```

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

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:

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:

```
!    " #    $    %    $

      %   &%'   &
      %   &%'   & "
   "
      %   &%'   &

      "          (
           "              "
```

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

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

```
!       "   #
$       "   #
%       "   #
&       "   #
```

Como já é esperado, a saída será:

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 de nido.

!

C ( )/ '

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

" " #
$

!
% &' % (#
!

Com isso, não seria necessário escrever em nosso método main , mas repare que estamos perdendo a exibilidade de customizar nossas mensagens. Será que todos que forem utilizar esse método querem mostrar a mensagem Desconto não pode ser maior do que? 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 especí cas em seus moldes, além de não sobrecarregá-los com comportamentos que não deveriam ser de sua responsabilidade.

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

") #

!" # ! $

% & ' ' (
!')
* + ,!
-
!" # !
$
-
-
-

Ao executá-la, teremos o resultado:

" #
" # + (

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

" #
') * + ,

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:

. ' '

!" # ! $

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

O problema é que o atributo valor da classe Livro pode ser acessado diretamente e modi cado 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 modi cados independente das condições veri - cadas em seus métodos.

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

```
!  "
```

```
##  $
```

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

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 .

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 .

Sua implementação será bem simples, observe:

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

!

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:

E como esse método não recebe parâmetros, o uso dos será opcional.

Agora podemos imprimir dessa forma:

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:

```
                    ! "
```

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:

```
                    ! "
#           $    % &
                '    !
```

Repare que o método lenght foi usado para retornar o tamanho de uma String . Além desse há diversos outros métodos úteis de nidos 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 deveria ser isolado em um método, assim evitamos repetição de código possibilitando o reúso

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:

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.

E : GGAS A

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`. Selecionando 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!

http://blog.caelum.com.br/ / / /nao-aprender-oo-getters-e-setters/

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

Precisamos fazer o mesmo para nossa classe Aluno :

!

!

E, por m, será necessário modi car a classe CadastroDeLivros para que deixe de acessar os atributos diretamente. Seu código nal deve car parecido com:

```
                                !

"                       #   "
        $   %&          '    %
            %             (           %
          %)*+ ,-. /01 )2%

                        #
        $   %3     0  4 5    %
              %$                      %
        6    -1 12
        7   %1/080-8..*-28,.8.%

        "
```

Repare que ele cou 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:

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

!!

!!

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 caria assim:

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:

```
! !
```

Esse código não compilará, a nal, quando alguém zer 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 car assim:

```
" #
```

! "
$ %&' ()* +,- %.

/ /
0 , 1 2
3
4)- -.
5 -+,6,)6**&).6(*6*

3 7

1
! "
$ %&' ()* +,- %.

/ / /
/ /8 1 9:
/ 3 #
/ 4)- -.
/ 5 -+,6,)6**&).6&&6.

/ 3 7
;
;

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

```
    !    " # $
%    &'                    ( (
)     *+ +
```

```
*       + ,  -.                 / 0
            1             2
            1       3           3         2
4

*       + ,  -/ 0
4
```

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:

```
*       + ,  -.                 / 0
       -/2
            1             2
4

*       + ,  -/ 0
            1       3           3         2
4
```

```
! !
```

Note que em nosso construtor já de nimos um valor padrão para o atributo impresso . Todo Livro que não tenha o valor de impresso de nido 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 .

```
                " " !
#               $ %
        #

                    &
```

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

```
                " " !
#               $ %
        #
     #    '       ((                  $ )*
        #
```

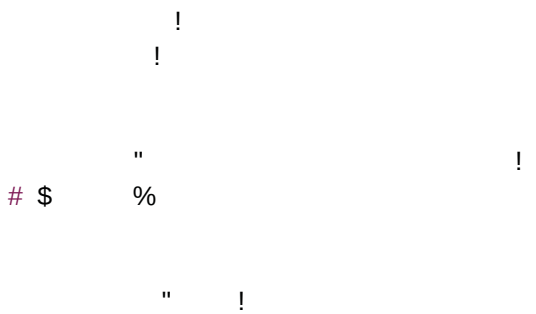
Agora nosso código funciona como esperado, mas está um pouco mais verboso e difícil de entender. O grande problema aqui é que, a cada método cuja regra seja diferente, teremos que colocar um novo if parecido com esse, isso sem contar que novos tipos de Livro podem aparecer e tornar nosso código ainda 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 Objeto! Podemos criar uma classe Ebook definindo os atributos e comportamentos específicos desse novo tipo.

Nosso código já está um pouco mais interessante, a nal 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 dizer na declaração da classe que ela é uma Livro, que é uma extensão dessa classe:



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:

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

H 1

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 demais o acoplamento entre suas classes.

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

```
! "  #  $
% & ' "
! " (
)*
++  & "
```

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

```
@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 (`super()`).

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. A nal, um atributo `private` só pode ser acessado pela própria classe, nem mesmo as classes lhas (`super()`) 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 modi car a visibilidade dos atributos da classe `Livro` para `protected`, que é um meio termo entre `public` e `private`.

```
@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.

```
@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 `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 . para . , 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.

. R L F

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

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:

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.

```
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");
```

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

Repare que criamos um `LivroFisico` e um `Ebook` com o mesmo `Autor`. Nosso próximo passo será adicionar esses dois elementos em um `CarrinhoDeCompras`. O código deve car 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 `LivroFisico`s e também `Ebook`s, 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 `Livro`s (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`:

```
Adicionado: LivroFisico@4f23c55
Adicionado: Ebook@21a33b44
```

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.

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

Mas, como vimos, também podemos dizer que `Ebook` é do tipo `Livro`, a nal 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 car 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 *cast*, 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

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:

```
Adicionado: LivroFisico@4f23c55
Adicionado: Ebook@21a33b44
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:

```
Adicionado: LivroFisico@4f23c55
Adicionado: Ebook@21a33b44
Total89.8
```

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:

```
Adicionado: LivroFisico@4f23c55
aplicando desconto no Livro
Adicionado: Ebook@21a33b44
Total 80.21600000000001
```

http://www.artima.com/intv/gosling P.html

```
            = 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");
```

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

A nal, 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 modi cador `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 nal.

```
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

```
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 não instanciar

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.

Mini livros

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);
    }
}
```

```
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 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 de nirmos isso, ele já poderia ter um desconto de até 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 caria 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 caria 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 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 car assim:

```
public class MiniLivro extends Livro {

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

```
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. A nal, 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 lhas 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 car 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

```
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 `Livro`s, adicionar `Revista`s. 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 caram 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 polimorsmo 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 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 polimorsmo. A hierarquia de nossas classes seria:

```
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 . da linguagem, também não pode ter nenhum método concreto, ou seja, com implementação. Veremos que, a partir do Java . , isso mudou um pouco.

Podemos agora fazer com que todas as classes que identificamos 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

```
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

```
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:

```
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 `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

Adicionado: LivroFisico@4f23c557
```

```java
@FunctionalInterface
public interface Promocional {

    boolean aplicaDescontoDe(double porcentagem);

    default boolean aplicaDescontoDe10Porcento() {
        return aplicaDescontoDe(0.1);
    }
}
```

```
@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 ?

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

```
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`

```
    .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 se um `import` for por alguma razão você precisar usar duas classes com o mesmo nome dentro do mesmo arquivo. Nesse caso, você poderá fazer `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, ela não tem um pacote de nido. 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.

http://www.oracle.com/technetwork/java/codeconventions- .
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 as classes devem ser públicas para que quem visíveis entre os diferentes pacotes. Logo falaremos mais sobre os modificadores de visibilidade e entenderemos a fundo as suas

```
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.

```
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 `import`s 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 (quick fix), 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);
```

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

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

E

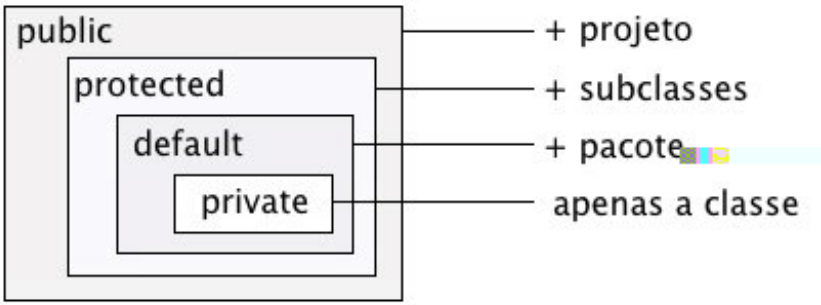
Note que um pacote é representado como uma estrutura de pastas no seu sistema operacional. Por exemplo, a classe `Livro` que está localizada no pacote `br.com.casadocodigo.livraria` está dentro do diretório `br/com/casadocodigo/livraria`, que se encontra dentro da pasta `src` de seu projeto.

. M (

Agora que nossas classes estão organizadas em pacotes, podemos entender um pouco melhor os diferentes modi cadores 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 modi cador 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 modi cador algum). A essa altura, você já pode ter percebido que neste caso apenas as classes do mesmo pacote podem ter acesso aos atributos, construtores, métodos ou classes com a ausência de um modi cador de acesso.


public
protected
default
private
+ projeto
+ subclasses
+ pacote
apenas a classe

```
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 exibilidade de adicionar qualquer classe que implemente essa interface `Produto`. Mas, fora isso, nosso código não está nem um pouco exível! Além de termos um número limitado de produtos, no método `adiciona` precisaríamos fazer várias condições para veri car 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 `if`s todos com as condicionais. Esse código seria bem difícil de manter, quase inviável. Além disso, quantos atributos do tipo `Produto` um `CarrinhoDeCompras` teria? ? ? 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 `array`s! 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 `String`s 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 car 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 signi ca 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 posições. Poderíamos sim receber a quantidade de Produto s no construtor da classe ou criar um método setProdutos, isso a deixaria um pouco mais exível. Mas a princípio, deixaremos o CarrinhoDeCompras assim, limitado a 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. Um detalhe muito importante é que a posição é a segunda posição do array, e não a primeira. Isso acontece pois seus índices vão de (zero) até seu tamanho -, ou seja, como criamos um array de posições, ele vai de a.

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

```
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 situ-

> **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@4f23c557
Adicionando:
    br.com.casadocodigo.livraria.produtos.Ebook@21a33b44
Total 83.81
53.91
29.9
```

O código está funcionado 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@4f23c557
Adicionado:
    br.com.casadocodigo.livraria.produtos.Ebook@21a33b44
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`!

```
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 (lenght) do array para conseguir percorrê-lo, isso evitaria a confusão com <= e que causou o ArrayIndexOutOfBoundsException .

. A ( ) !

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
```

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!");
```

```
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.

```
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@4f23c557
Adicionado:
        br.com.casadocodigo.livraria.produtos.Ebook@21a33b44
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.

```
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 J

Caso você queira executar a mesma ação para dois tipos de Exception diferentes, desde o Java 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.

```
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.

. M E)

As `RuntimeException` s

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.

```
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 signi ca 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");
    }
}
```

```
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 THROWS

É muito comum confundir no início, mas lembre-se que usamos o throw no imperativo quando estamos efetivamente lançando uma exeption. Quando escrito no indicativo, o throws apenas informa ao compilador e a quem mais interessar que determinado método lança uma exception.

```
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 `Exception`s 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`!

http://docs.oracle.com/javase/ /docs/api/java/lang/Exception.html

Repare que logo no inicio deste arquivo temos as suas subclasses diretas (Direct Known Subclasses). Achou que são muitas? E essas são apenas as lhas diretas, ainda temos as lhas 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.

```
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;
```

```
                ++;
        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 Produto s.

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.

```
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`.

```
        [] 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. Nosso código caria 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 `casting`, claro, mas veja como caria nosso código:

```
      [] 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 `casting`s e mais `casting`s, 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 %

Também podemos fazer nossos `casting`s 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@21a33b44
```

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@21a33b44
```

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

```
.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 void toString() {
        System.out.println("Nome: " + nome);
        System.out.println("Descrição: " + descricao);
        System.out.println("Valor: " + valor);
        System.out.println("ISBN: " + isbn);

        if (this.temAutor()) {
            autor.toString();
        }
        System.out.println("--");
    }
}
```

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:

```
            = 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 `Autor`es, 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, a nal, qual a vantagem de usar `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 de nindo 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 pala linha:

```
Autor outro = (Autor) obj;
```

Note que o `equals` recebe `Object` como argumento, portanto a primeira coisa que zemos 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 alterativas 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 estaríamos comparando sua referência. Adiante falaremos mais sobre o problema da comparação de `String`s, mas o importante agora é perceber que estamos delegando a comparação do nome (`String`) para o método `equals` da própria classe `String`. Ele foi sobrescrito 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");
}
```

```
@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());
```

```
= new Autor("Rodrigo");
```

Nem mesmo se for apenas uma `String`, pois todo objeto em Java herda direta 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 ., ele compilará. Isso não signi ca que os tipos primitivos como `int` herdam de `Object` ou são do mesmo tipo que uma referência, esse é apenas um truque do compilador adicionado no Java . para simpli car 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).

```
                = 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

```
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");
```

```
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/ /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 a :

```
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 .

```
= 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 `String`s 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 `String`s. 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 `String`s 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:

```
    .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 sobre uma `String`, eles sempre retornam uma referência nova para o valor transformado.

Agora que já conhecemos essa característica fundamental, podemos usar os seus mais de 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

http://docs.oracle.com/javase/ /docs/api/java/lang/package-summary.html

```
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 `Produto`s que poderá ser adicionada, como:

```
CarrinhoDeCompras carrinho =
    new CarrinhoDeCompras(new Produto[ 10 ]);
```

Mas como saber exatamente se teremos 10, 20, ou `Produto`s 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 arrays.

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 3:

```
                = new ArrayList();
String valor = "conhecendo uma ArrayList";
lista.add(valor);
System.out.println(lista.contains(valor));
lista.remove(valor);
```

```
.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 sobrecarga 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 `Produto`s.

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:

```
<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 Produto s. Há dois grandes ganhos aqui, o primeiro será pela não necessidade de uma casting ao recuperar valores desta lista. Veja:

```
Produto buscado = produtos.get(0);
```

O código ca 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");
```

```
<Produto> produtos = new ArrayList<>();
```

Esse recurso era bastante limitado, funcionava basicamente se a declaração fosse feita na mesma linha. No Java , 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 ca 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, deve valer 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

```
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 uma `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:

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

for (Produto produto : produtos) {
    System.out.println(produto);
}
```

```
    <Produto> produtos = carrinho.getProdutos();

for (Produto produto : produtos) {
    System.out.println(produto);
}
```

Isso signi ca que o enhanced-for também funciona com qualquer tipo de List (na verdade, com qualquer Iterable , como veremos mais à frente).

. O L

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 outro critério.

Até o Java , 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 modi cou a estrutura interna da lista, neste caso deixando-a em ordem alfabética.

http://docs.oracle.com/javase/ /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, a nal 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 especí co 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, a nal 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 de `generics` do Java? 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 `compareTo(Object)`.

Pronto, nalmente podemos partir para a implementação, que poderá

```
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
```

```
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.

. G

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, a nal 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 ca 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 con rmar! Aparentemente, tudo está igual, mas na verdade algumas coisas mudaram.

O primeiro ponto importante a se perceber é que, mesmo que você adicione mais mil cupons iguais nesse conjunto, ao mandar imprimir o seu tamanho (`size`) ele ainda será . Você pode rodar o código a seguir para con rmar isso:

```
HashSet<String> set = new HashSet<String>();
set.add("Não há repeticão em Conjuntos");
set.add("Não há repeticão em Conjuntos");
set.add("Não há repeticão em Conjuntos");
set.add("Não há repeticão em Conjuntos");
set.add("Não há repeticão em Conjuntos");
System.out.println(set.size());
```

Qual foi o valor da saída? Exatamente, 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 `HasSet` que já estamos utilizando, que apresentam uma performance muito melhor que a da `List` para fazer consultas (com o método `constains`, 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 31125 MS para executar
```

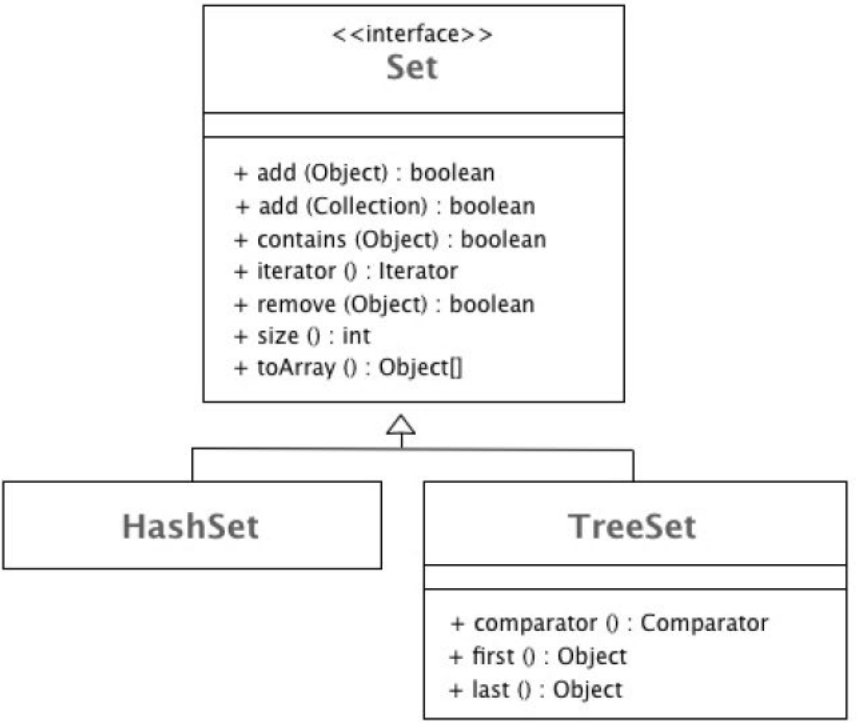
Agora mude para um `HashSet` para perceber a diferença. O mesmo código com `Set` apresentou o seguinte resultado:

Demorou 41 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:

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<>();
```

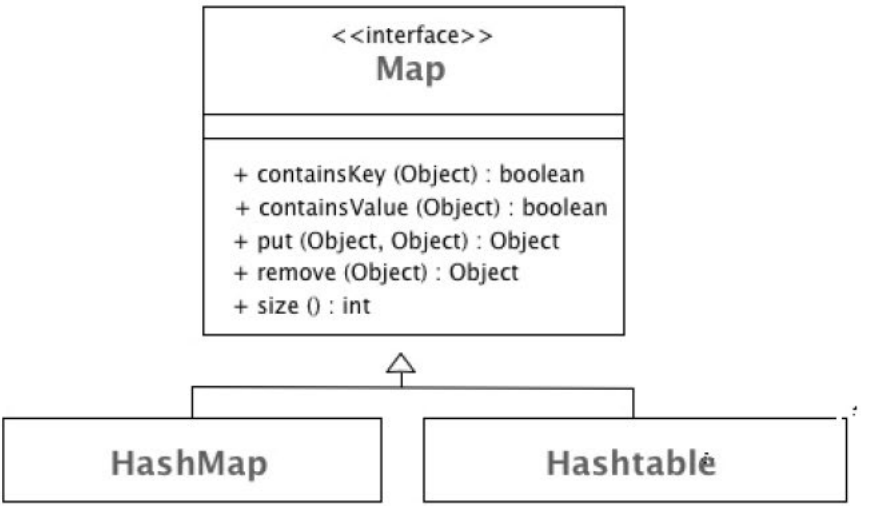
Note que agora precisamos passar os dois tipos genéricos: chave e valor. Nesse caso, declaramos a chave como `String` para um valor do tipo `Double`. Podemos inserir valores neste mapa utilizando seu método `put`, como a seguir:

```
mapa.put("cab123", 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("cab123");
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("cab11", 10.0);
        cupons.put("cab22", 12.0);
        cupons.put("cab33", 13.0);
        cupons.put("cab44", 14.0);

        // ...
    }

    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 valido.

```
Double desconto = gerenciador.validaCupom("cab11");

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.

```
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 código 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 O.O.");

        Livro java8 = new LivroFisico(autor);
        java8.setNome("Java 8 Prático");

        Livro ruby = new LivroFisico(autor);
        ruby.setNome("Livro de Ruby");
```

```
            <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 O.O.
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 eram as conhecidas classes anônimas. Você pode sim dar `new` em uma interface, mas terá que implementar seus métodos ali mesmo, numa 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 numa única instrução. Esse recurso é conhecido como classe anônima, pois a nal essa classe não tem nem mesmo um nome! Achou a sintaxe

```
      .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 capitulo de interface brevemente comentei sobre os `default methods`, heis um bom exemplo:

```
default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
}
```

Isso mesmo, esse método `default` 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 capitulo de interfaces, vimos brevemente que o Java 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 por exemplo 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());
});
```

```
      .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 ínicio. 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 parâmteros, o próprio compilador sabe inferir isso pra 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 nessa `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 o método que 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));
```

```
for (Livro livro : livros) {
    System.out.println(livro.getNome());
}
```

No Java 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 possivel utilizar uma expressão lambda para representar nossas intenções.

O lambda l -> System.out.println(l.getNome()) nada mas diz do que: •para cada livro, chame o método println imprimindo seu nomeŽ.

. F

Além da forma de ordenar e iterar, o Java mudou bastante a forma de fazer algumas operações rotineiras de quando estamos trabalhando com coleções. Quer um exemplo prático?

O trabalho de ltrar uma Collection

Dada uma lista de livros, queremos ltar apenas os que tenham a palavra Java em seu nome. Até o Java . 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, como a seguir:

```
    <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` como a seguir:

```
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 de uma boa olhada em seu código... o que achou? Note que além de ser muito verboso, ele exigiu a criação de uma lista intermediária para o resultado. Esse é sem dúvidas 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 de deixar claro que métodos são mutáveis, evitar problema de confli to 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();
```

```
        .stream().filter(l -> l.getNome().contains("Java"));
```

Estamos fazendo a mesma coisa que aqui, só que agora de uma forma muito mais declarativa! Nós colocamos um `if` dentro do `forEach` para filtrar os livros que contém a palavra "Java".

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 `Stream`s 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 `Stream`s 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 milhares métodos presentes nessa nova API. Você pode ver a lista completa em http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html e alguns exemplos em:

http://blog.caelum.com.br/o-minimo-que-voce-deve-saber-de-java- /

O J P '

Junto com Paulo Silveira escrevi o livro Java Prático: Lambdas, Streams e os novos recursos da linguagem. Java 8 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-java

https://www.jcp.org/en/introduction/overview

. E 23 *

Diferente das demais linguagens, uma aplicação Java pode ser executada em qualquer um dos diferentes sistemas operacionais existentes, como por exemplo Windows, Linux ou Mac OS. Essa possibilidade abre muitos caminhos e foi um dos principais fatores que tornaram a linguagem tão atraente pro mercado. Exibindo esse benefício, o Java teve como slogan o cial o Write once, run anywhere (escreva uma vez, rode em qualquer lugar).

A grande chave para essa portabilidade é a máquina virtual, ou Java Virtual Machine). No lugar de instruções nativas para um determinado hard-

```
    <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 pode ser escrito de forma muito mais declarativa e funcional:

```
        .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/

http://www.guj.com.br

Outro caminho natural para continuar seus estudos é o livro Java 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-java

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.