

Allen Oberleitner
Andrey Araujo Masiero

Programação orientada a objetos

Dados Internacionais de Catalogação na Publicação (CIP)
(Jeane Passos de Souza - CRB 8ª/6189)

Oberleitner, Allen

Programação orientada a objetos / Allen Oberleitner, Andrey Araujo Masiero. – São Paulo: Editora Senac São Paulo, 2021. (Série Universitária)

Bibliografia.

e-ISBN 978-65-5536-477-4 (ePub/2021)
e-ISBN 978-65-5536-478-1 (PDF/2021)

1. Desenvolvimento de sistemas 2. Linguagem de programação
3. Programação orientada a objetos 4. Software (desenvolvimento) I.
Masiero, Andrey Araujo. II. Título. III. Série.

21-1225t

CDD – 003

005.13

BISAC COM051210

COM051000

Índice para catálogo sistemático

- 1. Desenvolvimento de sistemas 003**
2. Linguagem de programação 005.13

PROGRAMAÇÃO ORIENTADA A OBJETOS

Allen Oberleitner
Andrey Araujo Masiero





Administração Regional do Senac no Estado de São Paulo

Presidente do Conselho Regional

Abram Szajman

Diretor do Departamento Regional

Luiz Francisco de A. Salgado

Superintendente Universitário e de Desenvolvimento

Luiz Carlos Dourado

Editora Senac São Paulo

Conselho Editorial

Luiz Francisco de A. Salgado

Luiz Carlos Dourado

Darcio Sayad Maia

Lucila Mara Sbrana Sciotti

Jeane Passos de Souza

Gerente/Publisher

Jeane Passos de Souza (jpassos@sp.senac.br)

Coordenação Editorial/Prospecção

Luís Américo Tousi Botelho (luis.tbotelho@sp.senac.br)

Dolores Crisci Manzano (dolores.cmanzano@sp.senac.br)

Administrativo

grupoedsadministrativo@sp.senac.br

Comercial

comercial@editorasenacsp.com.br

Acompanhamento Pedagógico

Mônica Rodrigues dos Santos

Designer Educacional

Diogo Maxwell Santos Felizardo

Revisão Técnica

Ana Cristina dos Santos

Preparação e Revisão de Texto

Janaina Lira

Projeto Gráfico

Alexandre Lemes da Silva

Emilia Corrêa Abreu

Capa

Antonio Carlos De Angelis

Editoração Eletrônica

Stephanie dos Reis Baldin

Ilustrações

Stephanie dos Reis Baldin

Imagens

Adobe Stock Photos

E-pub

Ricardo Diana

Proibida a reprodução sem autorização expressa.
Todos os direitos desta edição reservados à

Editora Senac São Paulo

Rua 24 de Maio, 208 – 3º andar

Centro – CEP 01041-000 – São Paulo – SP

Caixa Postal 1120 – CEP 01032-970 – São Paulo – SP

Tel. (11) 2187-4450 – Fax (11) 2187-4486

E-mail: editora@sp.senac.br

Home page: <http://www.livrariasenac.com.br>

© Editora Senac São Paulo, 2021

Sumário

Capítulo 1

Introdução à programação orientada a objetos, 7

- 1 Construindo software, 8
- 2 Programação orientada a objetos × programação estruturada, 10
- 3 Classes e objetos, 13
- Considerações finais, 21
- Referências, 22

Capítulo 2

Encapsulamento e modificadores de acesso, 23

- 1 Protegendo sua classe, 24
- 2 Modificadores de acesso, 25
- 3 Métodos getters e setters, 28
- Considerações finais, 34
- Referências, 35

Capítulo 3

Sobrecarga de métodos, 37

- 1 O recurso "camaleão", 38
- 2 Métodos polimórficos do Java, 40
- 3 Construindo métodos polimórficos, 47
- Considerações finais, 50
- Referências, 51

Capítulo 4

Construtores, 53

- 1 Tudo começa do início, 54
- 2 Utilizando métodos construtores, 56
- 3 Sobrecarga de métodos construtores, 60
- Considerações finais, 66
- Referências, 66

Capítulo 5

Herança, 67

- 1 Entendimento sobre herança, 68
- 2 Construindo a primeira família de objetos, 70
- 3 Protegendo membros da classe, 84
- 4 Polimorfismo, 85
- 5 Classes e métodos abstratos, 88
- 6 Exercícios de fixação, 92
- Considerações finais, 92
- Referências, 93

Capítulo 6

Interface, 95

- 1 Uso da interface em classes, 96
- 2 Interfaces como tipo, 104
- 3 Exercícios de fixação, 107
- Considerações finais, 108
- Referências, 108

Capítulo 7

Constantes, atributo estático e método estático, 109

- 1 Constantes, 110
- 2 Atributos e métodos estáticos, 115
- Considerações finais, 120
- Referências, 120

Capítulo 8

Genéricos, 121

- 1 Parâmetro de tipo, 124
- 2 Métodos genéricos, 125
- 3 Classes genéricas, 130
- 4 Exercícios de fixação, 133
- Considerações finais, 134
- Referências, 134

Sobre os autores, 137

Capítulo 1

Introdução à programação orientada a objetos

Construir um software não é apenas sair programando ou construindo telas desordenadamente. É preciso um projeto de software para entendermos o que vamos construir e se o produto gerado pelo nosso desenvolvimento vai, realmente, satisfazer às necessidades dos usuários do sistema.

O desenvolvimento do software é uma das etapas do processo de construção de software e é responsabilidade dos desenvolvedores de sistema. Porém, não basta apenas saber programar ou conhecer alguma linguagem de programação para executar essa tarefa. Um bom profissional de desenvolvimento de software deve saber ler diagramas de sistema e interpretar seu conteúdo em código e comandos executáveis.

É aqui que entra a programação orientada a objetos. Uma linguagem orientada a objetos possui recursos para o desenvolvedor de software poder implementar todos os comandos e instruções necessários para a construção do sistema. Dois desses recursos são as classes e os objetos, elementos principais do paradigma orientado a objetos.

Neste capítulo, você compreenderá esse paradigma, capacitando-se a ler e interpretar um diagrama de classes, bem como a desenvolver o código necessário, em Java, para criar essa classe. Faça uma boa leitura e desfrute do conhecimento que será adquirido.

1 Construindo software

Construir um software não é simplesmente programar linhas e linhas de comandos e instruções lógicas seguindo a sintaxe e a semântica de uma linguagem de programação e um paradigma de programação.

Desenvolver um sistema computacional envolve um projeto de software. É como o projeto de uma casa, em que entendemos as necessidades dos moradores e tomamos decisões como o lado em que o sol nasce, a quantidade de janelas e cômodos, se terá varanda, quintal, etc. Após essas definições, desenhamos a planta da casa e construímos maquetes até a aprovação dos moradores. Projeto aprovado, começamos a construir a casa, levantar as paredes e fazer o telhado, as fundações e o acabamento, de acordo com os gostos dos novos moradores.

Essa é a engenharia envolvida no processo de construção de uma casa. No caso de um sistema de computador, temos a engenharia de

software que, segundo Sommerville (2011), é uma disciplina de engenharia cujo foco está em todos os aspectos da produção de software, desde os estágios iniciais da especificação do sistema até sua manutenção, quando o sistema já está sendo usado. Esses aspectos são chamados de etapas no processo de construção de software.

De acordo com Pressman (2011), as etapas do processo de construção de software envolvem: comunicação, planejamento, modelagem, construção e entrega. Neste livro, vamos nos concentrar na construção do software, fazendo uso da linguagem de programação Java, cujo paradigma é orientado a objetos. Entretanto, todo desenvolvedor de software deve conseguir ler e interpretar todas as etapas do processo de construção de um software, principalmente a etapa de modelagem, onde se encontram importantes diagramas do sistema (como se fosse a planta da casa), como o diagrama de classes.



PARA SABER MAIS

Paradigma de programação é uma forma de classificar as linguagens de programação de acordo com sua estrutura e suas funcionalidades. O paradigma é a forma como um desenvolvedor vê a execução do programa e as técnicas utilizadas pela linguagem de programação. A orientação a objetos, por exemplo, é o paradigma da linguagem de programação Java.

De acordo com Sommerville (2011), os diagramas de classe são usados no desenvolvimento de um modelo de sistema orientado a objetos para mostrar as classes de um sistema e as associações entre elas.

Porém, você pode estar se perguntando: afinal, o que é uma classe? Essa é uma ótima pergunta, primordial para o prosseguimento da nossa compreensão sobre orientação a objetos.

2 Programação orientada a objetos x programação estruturada

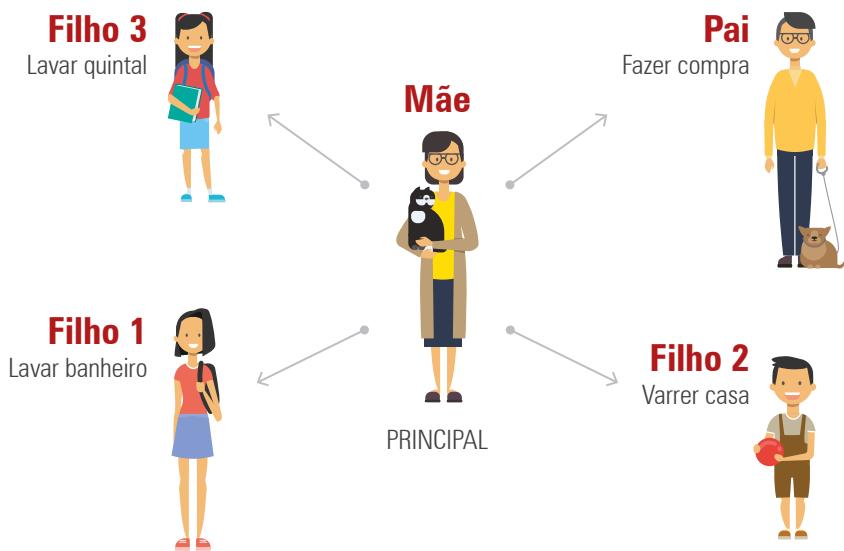
Antes de entender o que é uma classe, temos que compreender o paradigma orientado a objetos e, para isso, faremos uma comparação com o paradigma estruturado.

A programação estruturada trouxe um grande avanço para a maneira de programar (até então era feita de maneira linear, linha a linha, seguindo uma lógica de programação sequencial). Ela é mais interativa e potencializa a solução de problemas, pois trabalha com a divisão desse problema em problemas menores. Esse é o conceito de modularização, processo no qual, de acordo com Oberleitner e Silva (2020), dividimos nosso programa em partes ou módulos independentes que executam tarefas específicas.

Conforme Puga e Risset (2009), as principais estruturas do paradigma estruturado são: sequência, decisão e iteração. Hoje, essas estruturas podem parecer triviais, mas esses elementos revolucionaram a forma de programar na época. Além de possibilitar a construção de softwares mais complexos, o paradigma estruturado trouxe vantagens em relação a custo de projeto e tempo de construção.

Para entender a programação estruturada, imagine que, por exemplo, uma mãe vá distribuir os afazeres da casa entre os membros da família. Se o problema for limpar a casa e cozinar, ela encarrega um dos filhos de lavar o banheiro, o segundo de varrer a casa, o terceiro de lavar o quintal e o pai de comprar a comida. Enquanto isso, ela orquestra tudo para que todos executem suas tarefas e, dessa forma, o problema seja solucionado. Nesse exemplo, a mãe é o módulo principal do nosso programa e os filhos e o pai são os módulos das atividades que foram divididas (figura 1).

Figura 1 – Representação de modularização



No paradigma orientado a objetos, temos o objeto como elemento principal. Aliás, a programação orientada a objetos surge da necessidade de termos uma linguagem de programação cujo código fosse escrito o mais próximo possível da maneira como os seres humanos interagem. Os objetos representam qualquer “coisa” do mundo real: um carro, uma cadeira, um animal e até mesmo uma pessoa.

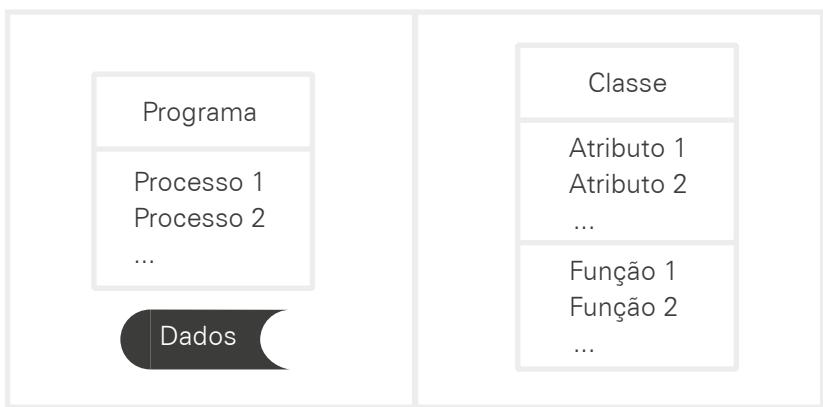
De acordo com Coelho (2012), a programação orientada a objetos é baseada nos seguintes conceitos: objetos, classes, atributos, métodos, encapsulamento e herança. Podemos, ainda, destacar alguns outros, como construtores, polimorfismo e interface. Nesta obra, veremos todos esses elementos, de forma a compreender a programação orientada a objetos.

No nosso exemplo anterior, as pessoas seriam objetos no sistema, e os afazeres domésticos poderiam ser métodos de uma classe chamada CASA, por exemplo. Daqui a pouco falaremos mais sobre classes

e objetos, mas neste momento é importante saber que os objetos desempenham tarefas em nosso sistema e as classes geram os objetos de que precisamos, como moldes.

Basicamente, a diferença entre os paradigmas estruturado e orientado a objetos é que no estruturado estamos pensando em modularização e rotinas como tomada de decisão; já no orientado a objetos, nossa preocupação é entender os objetos que fazem parte do nosso programa, quais suas características e quais funções (métodos) eles desempenham, ou seja, como os objetos interagem entre si (figura 2). É importante frisar que os dois paradigmas são bastante diferentes e a forma como programamos em cada um muda completamente.

Figura 2 – Paradigma estruturado x paradigma orientado a objetos



Fonte: adaptado de Puga e Risset (2009, p. 26).

Como nosso foco é a programação orientada a objetos, devemos entender o que são as classes e os objetos e como vamos utilizar esses conceitos em nossos sistemas. Não se esqueça que devemos saber ler os diagramas (principalmente o diagrama de classes), interpretá-los e transformá-los em código de computador por meio de uma linguagem de programação, no nosso caso, a linguagem Java.

De acordo com Puga e Risset (2009), a utilização dos recursos de uma linguagem orientada a objetos traz muitos benefícios, como reúso de código, aumento de produtividade, segurança (em razão do encapsulamento de dados), robustez dos programas e facilidade de desenvolvimento e manutenção das aplicações.

3 Classes e objetos

Como dissemos, os objetos são os elementos principais no paradigma orientado a objetos e, neste tópico, vamos nos concentrar nele.

Conforme Barnes e Kolling (2010), ao escrever um programa de computador em uma linguagem orientada a objetos, você criará um modelo de alguma parte do mundo. As partes das quais o modelo é construído são os objetos que aparecem no domínio do problema.

Segundo Horstmann e Cornell (2010), os objetos possuem três características-chave:

1. O comportamento do objeto: o que você pode fazer com esse objeto ou quais métodos pode aplicar a ele?
2. O estado do objeto: como o objeto reage quando você aplica esses métodos?
3. A identidade do objeto: como o objeto é diferenciado de outros que poderiam ter o mesmo comportamento ou estado?

Por exemplo, imagine o seu boletim de notas. Cada aluno tem seu próprio boletim, porém cada um tem uma identidade única, mesmo que apresente exatamente as mesmas notas. O boletim possui um comportamento de podermos adicionar notas, excluir notas ou publicá-lo. O seu estado indica o que podemos fazer com ele, em determinados momentos. Não podemos excluir uma nota se ela não existe no sistema, por exemplo.

Se temos um boletim, provavelmente teremos um professor alimentando as notas. O professor é um objeto no nosso sistema, assim como o aluno que recebe essas notas. Dessa forma, vamos definindo quem são nossos objetos dentro do nosso problema.

Entretanto, para um objeto existir no nosso sistema e executar todas as suas funções – possuir comportamento, estado e identidade –, temos que criar esse objeto. Como criamos um objeto, afinal? Essa é uma pergunta que será respondida com o conceito de classe. De acordo com Horstmann e Cornell (2010), uma classe é o modelo ou o esquema a partir do qual os objetos são criados.

Imagine que você e sua família queiram fazer ovos de Páscoa de chocolate na casa de vocês. Do que vão precisar para essa tarefa?

- Ingredientes: chocolate, granulado, amendoim e bombons.
- Fôrma de plástico para o ovo de Páscoa.

Nossa fôrma se comportará como uma classe, pois vai possibilitar a criação de vários ovos de Páscoa a partir dela. E os ovos terão características como tamanho, peso e tipo de chocolate (branco, preto, amargo, ao leite), bem como se terá granulado, amendoim e bombons dentro. Além disso, existem etapas para fazer um ovo de Páscoa:

1. derreter o chocolate;
2. passar uma camada de chocolate nas duas partes da fôrma;
3. resfriar;
4. adicionar o recheio.

Note que são várias atividades ou funções que temos ou podemos executar. Nossa classe vai fornecer essas funcionalidades para que os objetos possam executar. Portanto, precisamos de uma classe, com

várias características (atributos) e várias funcionalidades (métodos) para fornecer aos objetos que ela vai criar. A essa “criação” damos o nome de instância, ou seja, um objeto é instanciado de uma classe.

Ficou complicado? Vamos, então, construir o diagrama de classes voltando ao exemplo do boletim do aluno, com seus atributos e métodos. Em seguida, vamos começar a usar nossos primeiros comandos Java para construir nossa classe e instanciar nossos objetos.

Para a classe *Boletim*, teremos como características (atributos):

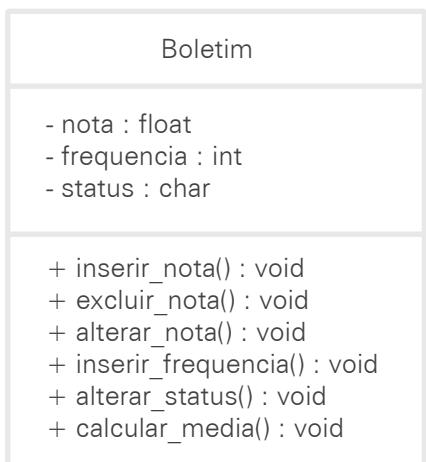
1. Nota: local onde o professor atribui uma nota de uma prova ou atividade.
2. Frequência: local onde o professor relata as faltas de um aluno.
3. Status: estado que informa se o aluno está aprovado ou reprovado em determinada disciplina.

Como funcionalidades (métodos), podemos ter:

1. Inserir nota
2. Alterar nota
3. Excluir nota
4. Calcular média
5. Inserir frequência
6. Alterar status

Estamos prontos para construir nossa classe *Boletim*. Vamos utilizar o diagrama de classes para representá-la, conforme a figura 3.

Figura 3 – Classe *Boletim* construída de acordo com a ferramenta UML



Não se preocupe agora com alguns símbolos, como os sinais de mais (+) e menos (-), nem com os valores passados como parâmetros nos métodos. Ainda estudaremos todos esses conceitos no decorrer do livro. Ao que devemos nos atentar agora é o nome da classe *Boletim*, os atributos *nota*, *frequencia* e *status*, além dos métodos *inserir_nota*, *excluir_nota*, *alterar_nota*, *inserir_frequencia*, *alterar_status* e *calcular_media*.



PARA SABER MAIS

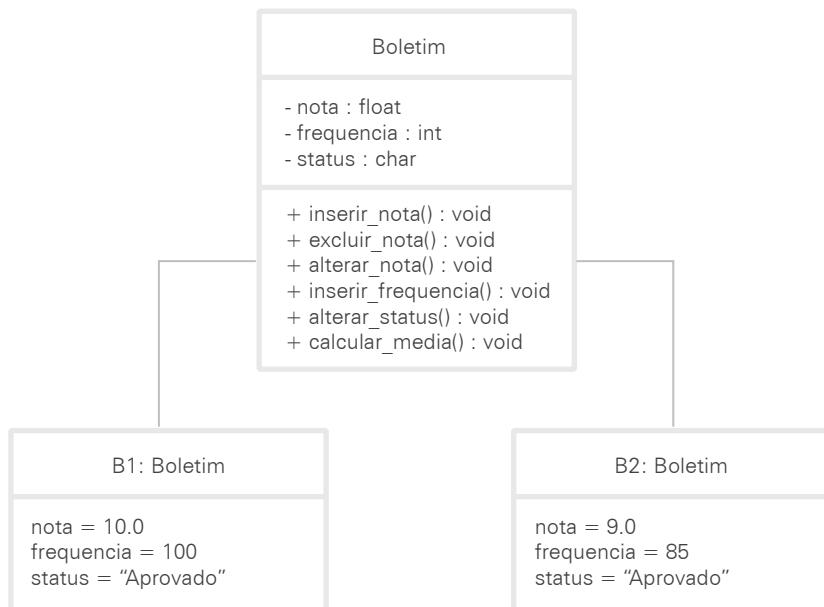
A Unified Modeling Language (UML) é uma linguagem de notação-padrão utilizada para elaborar, escrever e comunicar um projeto de software. Em tradução livre, UML significa linguagem de modelagem unificada. Ela se baseia em conceitos de orientação a objetos e utiliza conceitos de diagramação para representar nosso sistema de computador.

Para mais informações, consulte a obra *Princípios de análise e projetos de sistemas com UML*, de Eduardo Bezerra (2015).

Mas você pode estar se perguntando: entendi o que é uma classe, porém o que é um objeto? A partir de uma classe, podemos criar tantos objetos quanto desejarmos. Por exemplo, podemos instanciar o objeto *B1* (que seria o seu boletim) e *B2* (que seria o boletim do seu amigo de sala). Cada um seria um boletim diferente, com comportamento e estado próprios.

A diferença de um objeto para uma classe está nas atribuições de valores aos seus atributos. Enquanto numa classe nós temos a declaração dos atributos, nos objetos temos os valores desses atributos. Por exemplo, seu boletim pode ter nota = 10, frequência = 100% e status = aprovado; o boletim do seu amigo, nota = 9, frequência = 85% e status = aprovado, e assim por diante. Cada boletim é único, com os valores atribuídos de acordo com aquele objeto. Já a classe permanece a mesma, com os mesmos atributos e os mesmos métodos (figura 4).

Figura 4 – Instâncias de objetos do tipo Boletim



Com o diagrama de classe em mãos, estamos prontos para construir nossa classe e efetuar as instâncias com os comandos Java.



IMPORTANTE

Lembre-se: todos os comandos Java são escritos dentro de uma classe.
Se for a classe principal, esta terá um método chamado *main()*.

Para criar uma classe em Java, devemos usar o modificador de acesso *public* seguido da palavra reservada *class*. Depois, basta colocar o nome da classe, com a primeira letra maiúscula, e abrir e fechar chaves. Todos os comandos da classe devem estar entre o par de chaves {}. Além disso, essa classe será criada dentro de um arquivo *.java* que possui o mesmo nome da classe. Não se preocupe com os modificadores de acesso, vamos estudá-los no capítulo 2.

Dessa forma, nossa classe *Boletim* ficará assim:

```
public class Boletim {  
  
    float nota;  
    int frequencia;  
    String status;  
  
    public void inserir_nota(float nota) {  
  
    }  
  
    public void excluir_nota() {  
  
    }  
  
    public void alterar_nota(float nova_nota) {  
  
    }  
}
```

(cont.)

```
public void inserir_frequencia(boolean freq) {  
}  
  
public void alterar_status() {  
}  
  
public void calcular_media() {  
}  
}
```

É importante frisar que os métodos dessa classe não possuem nenhum comando ainda. Portanto, eles não possuem nenhuma funcionalidade prática. Mas, conforme avançarmos nos capítulos, traremos mais conceitos e conteúdos, e logo você conseguirá desenvolver funcionalidades a eles.

Agora, precisamos instanciar dois objetos (*B1* e *B2*) e inserir valores aos atributos, conforme o exemplo anterior. Para efetuar a instância, também temos palavras reservadas do Java. Nosso objeto *B1* é do tipo *Boletim*, portanto vamos declarar uma variável chamada *B1*, que será do tipo *Boletim* e armazenará uma referência do objeto, ou seja, *B1* guarda o endereço de memória do objeto *Boletim*, e não um valor, como acontece nas variáveis de tipos primitivos. Porém, para iniciar esse objeto, devemos usar a palavra reservada *new* e chamar o construtor da classe *Boletim* (o construtor tem o mesmo nome da sua classe). Novamente, não vamos nos preocupar com isso agora. Estudaremos os construtores no capítulo 4.

```
Boletim B1 = new Boletim();
```

Como dissemos, todos os comandos em Java devem ser escritos dentro de classes. Para criar as instâncias, vamos construir a classe principal chamada *Faculdade* e um método principal chamado *main()* do nosso programa. As palavras reservadas também existem aqui.

E assim ficará nossa classe principal com duas instâncias (*B1* e *B2*) e com atribuições de valores aos atributos da classe *Boletim*:

```
public class Faculdade {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
  
        Boletim B1 = new Boletim();  
        Boletim B2 = new Boletim();  
  
        B1.inserir_nota(10);  
        B1.inserir_frequencia(true);  
        B1.alterar_status();  
  
        B2.nota = 9;  
        B2.frequencia = 85;  
        B2.status = "Aprovado";  
  
    }  
}
```

Note que fizemos as instâncias de *B1* e *B2* e, em seguida, inserimos valores aos seus atributos de duas formas diferentes. O objeto *B1* chamou seus métodos e passou, por parâmetros, os valores que ele queria inserir. Já o objeto *B2* chamou, diretamente, as variáveis que tiveram seus valores atribuídos sem a ação dos métodos. Se o “rodarmos”, esse sistema vai funcionar, porém o objeto *B2* está infringindo um conceito importante do paradigma orientado a objetos: o encapsulamento.

Como não introduzimos esse conceito ainda, o sistema vai funcionar assim mesmo. Entretanto, a forma correta de fazer atribuições de valores aos atributos é por passagem de parâmetros, como fez o

objeto *B1*. Estudaremos o encapsulamento no capítulo 2 e lá arrumaremos esse código.

Por enquanto, tudo o que precisávamos saber de programação orientada a objetos foi apresentado. Por ora, entre no ambiente Java e familiarize-se com os comandos, as classes, os atributos e os métodos. A cada capítulo, vamos inserir mais conceitos e nos aprofundar mais nesse maravilhoso mundo que é a programação orientada a objetos.

Para finalizarmos este capítulo, propomos um desafio: construa as classes *Aluno* e *Professor* usando alguma ferramenta UML e desenvolva seus códigos no Java. Pense nos atributos e métodos para cada objeto e também no comportamento, no estado e na identidade de cada um deles.

Considerações finais

Neste capítulo, vimos a importância de um projeto para a construção de software e que a construção (ou implementação) do nosso sistema é uma das etapas desse processo. Uma etapa que tem como responsáveis por ela os desenvolvedores de sistemas.

Evidenciamos as características do paradigma orientado a objetos e como uma linguagem orientada a objetos pode nos ajudar na etapa de construção do sistema. Trouxemos a importância de um desenvolvedor de software estar apto a ler e interpretar um diagrama de classes e transformá-lo em código executável. Além disso, compararamos os paradigmas estruturado e orientado a objetos, destacando seus principais elementos, como sequência, decisão, iteração e modularização (paradigma estruturado) e classes, objetos, atributos, métodos, encapsulamento e herança (paradigma orientado a objetos).

Por fim, construímos uma classe utilizando recursos UML. Desenvolvemos essa classe por meio de comandos e instruções da linguagem de programação orientada a objetos Java.

Referências

BARNES, David J; KOLLING, Michael. **Programação orientada a objetos com Java**: uma introdução prática usando o BlueJ. São Paulo: Pearson Prentice Hall, 2010.

BEZERRA, Eduardo. **Princípios de análise e projetos de sistemas com UML**. 3. ed. Rio de Janeiro: Elsevier, 2015.

COELHO, Alex. **Java com orientação a objetos**. Rio de Janeiro: Ciência Moderna Ltda, 2012.

HORSTMANN, Cay S.; CORNELL, Garry. **Core Java, volume 1**: fundamentos. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

OBERLEITNER, Allen; SILVA, Luís C. S. **Desenvolvimento de sistemas**. 1. ed. São Paulo: Senac São Paulo, 2020. (Série Universitária).

PRESSMAN, Roger. **Engenharia de software**: uma abordagem profissional. 7. ed. São Paulo: McGraw-Hill, 2011.

PUGA, Sandra; RISSET, Gerson. **Lógica de programação e estruturas de dados com aplicação em Java**. 2. ed. São Paulo: Pearson, 2009.

SOMMERVILLE, Ian. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

Capítulo 2

Encapsulamento e modificadores de acesso

A segurança é um dos elementos mais importantes quando nos referimos a programas de computadores. Pensando nisso, o paradigma orientado a objetos possui um recurso muito interessante e bastante útil para proteger suas classes, métodos e atributos de acessos indesejáveis, chamado encapsulamento. Além disso, o encapsulamento mantém a integridade do código.

Neste capítulo, vamos estudar os modificadores de acesso *public* e *private* e compreender como os objetos conseguem acesso aos elementos encapsulados. Além disso, vamos aprender como os métodos de manipulação de atributos *getters* e *setters* funcionam. Eles são também chamados de métodos de acesso e métodos modificadores, respectivamente.

Vamos finalizar o capítulo com exemplos e codificando em Java nossa classe completa e encapsulada.

1 Protegendo sua classe

O paradigma orientado a objetos nos traz muito recursos importantes e vamos começar a fazer uso deles.

Como apresentamos no primeiro capítulo, seus principais elementos são os objetos e as classes. As classes são compostas por atributos e métodos, sendo os atributos variáveis e os métodos as funcionalidades do objeto que será instanciado. Quando um objeto é criado (ou instanciado), ele pode usufruir de todos os métodos da classe, assim como obter todas as características dela.

No entanto, nosso objeto está fora dessa classe, ou seja, ele foi criado em alguma outra classe do nosso projeto, e para usufruir dos seus métodos e atributos, o objeto deve ter acesso a eles. Porém, por motivos de segurança, não é qualquer um que pode ter acesso aos elementos de uma classe.

Neste momento, você pode estar se perguntando: mas o objeto pode ter acesso irrestrito aos elementos da sua classe, não é mesmo? A resposta é: não. E o motivo é simples: não queremos que o objeto tenha acesso a elementos que possam “danificar” a classe, por exemplo. Além disso, os objetos só executam tarefas no nosso sistema por meio dos métodos, que são as ações da classe.

O paradigma orientado a objetos possui um recurso que protege os elementos de uma classe. Esse recurso é chamado de encapsulamento. É obrigatório indicarmos o nível de acesso dos objetos a elementos como classes, atributos e métodos no momento em que os criamos.

Lembra-se de como criamos a classe *Boletim* e seus métodos?

```
public class Boletim {  
  
    float nota;  
    int frequencia;  
    String status;  
  
    public void inserir_nota(float nota) {  
  
    }  
}
```

Note que inserimos a palavra reservada *public* na criação da classe *Boletim* e do método *Inserir Nota()*. Em relação aos atributos, não colocamos nenhuma palavra, mas, como dito no capítulo anterior, iríamos corrigir esse código agora.



PARA SABER MAIS

Segundo o dicionário Aurélio (FERREIRA, 2010), “encapsular” é incluir ou proteger alguma coisa em uma cápsula.

Aliás, em nosso diagrama de classe, havia sinais de menos (-) nos atributos. Vamos entender para que servem esses sinais e como utilizar o recurso de encapsulamento no paradigma orientado a objetos.

2 Modificadores de acesso

De acordo com Barnes e Kolling (2010), encapsulamento é a técnica que faz com que detalhes internos do funcionamento dos métodos de uma classe permaneçam ocultos para os objetos.

Segundo Deitel e Deitel (2010), existem níveis de proteção às classes e seus elementos. Esses níveis de acesso são indicados pelos

chamados modificadores de acesso. Vamos trabalhar, neste capítulo, com dois tipos de modificadores de acesso: *public* e *private*.



PARA SABER MAIS

Além dos modificadores de acesso *public* e *private*, existem outros como *protected* e *internal*.

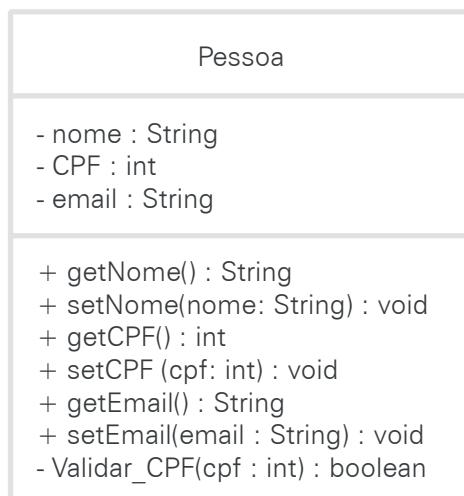
O modificador de acesso *protected* só permite acesso aos elementos da própria classe (como o *private*), porém também permite acesso aos das classes filhas, ou seja, classes que herdam dessa classe (falaremos sobre herança no capítulo 5).

O modificador de acesso *internal* permite acesso somente dentro do projeto atual.

De acordo com Horstmann (2019), o modificador de acesso *public* permite o acesso de qualquer objeto ao código interno da classe. É a forma menos segura de acesso. Já o modificador de acesso *private* permite o acesso somente dentro da classe em que o elemento foi criado. Como bom hábito de programação e por recomendação da engenharia de software, vamos declarar todas as variáveis de uma classe como *private* e todos os métodos como *public*, assim como a própria classe.

Dessa forma, quando construirmos nosso diagrama de classe UML, devemos indicar, por símbolos, o encapsulamento dos elementos de uma classe. A figura 1 apresenta o diagrama de classe UML *Pessoa*, com seus atributos encapsulados como privados e seus métodos encapsulados como públicos. Isso significa que um objeto instanciado dessa classe só poderá ter acesso aos atributos por meio dos métodos. O conjunto de métodos públicos de uma classe é também chamado de interface da classe, pois esta é a única maneira pela qual você se comunica com os objetos dessa classe.

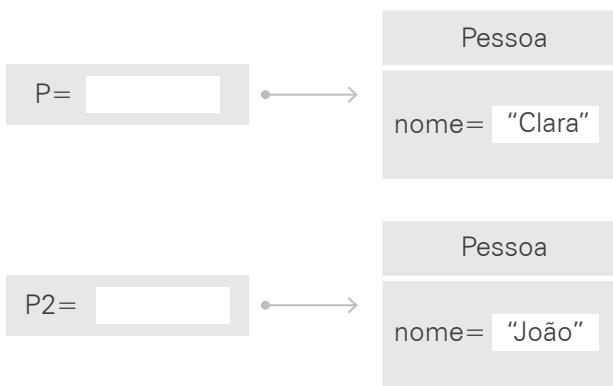
Figura 1 – Diagrama da classe Pessoa



Note que, ao criarmos um objeto *P* do tipo *Pessoa*, ele poderá ter acesso e modificar os atributos *Nome*, *CPF* e *Email* somente por meio dos métodos públicos que criamos. Se não tivéssemos criado nenhum método que manipulasse o *Nome*, por exemplo, o objeto *P* jamais teria acesso a essa informação.

Vamos supor que criássemos também o objeto *P2* do tipo *Pessoa*. Esse objeto também poderá modificar seus atributos por meio dos métodos, mas eles não são os mesmos do objeto *P*. Cada objeto, segundo Deitel e Deitel (2010), possui sua própria cópia de um atributo, chamada “campos de instância” ou “variáveis de instância”. Em outras palavras, quando declaramos a variável *Nome* na classe *Pessoa*, estamos reservando um espaço de memória para essa variável. Entretanto, quando instanciamos os objetos *P* e *P2*, cada um possui o atributo *Nome*, mas em endereços diferentes de memória, referenciando, dessa forma, suas variáveis (figura 2). Assim, nenhum objeto modifica diretamente a variável da classe *Pessoa*, e isso é garantido pelo encapsulamento privado.

Figura 2 – Instância de objetos



Fonte: adaptado de Horstmann e Cornell (2010, p. 142).



PARA PENSAR

Voltando à classe `Pessoa` representada no nosso diagrama de classes, podem surgir algumas perguntas:

- O que são esses métodos *getters* e *setters*?
- Por que o método `Validar_CPF()` está com o sinal de menos (-)?
- Por que alguns métodos possuem a palavra `void` e outros possuem o tipo da variável?

Essas são excelentes perguntas a que vamos responder a seguir.

3 Métodos *getters* e *setters*

Basicamente, os métodos *getters* e *setters* são funções que manipulam as variáveis que estão encapsuladas como privadas. Segundo Horstmann e Cornell (2010), há uma diferença conceitual entre o método *getters* e o método *setters*. O *getters* só consulta o estado do objeto

e informa sobre isso, ou seja, esse método acessa os campos de instância e é chamado de método de acesso. O setters modifica o estado do objeto, ou seja, altera os campos de instância e é chamado de método modificador.

Isso responde a duas de nossas perguntas anteriores. Como os métodos de acesso consultam uma informação e a retorna, precisamos indicar qual é o tipo de retorno dessa informação. Isso é indicado pelo tipo da variável retornada pelos métodos getters. No caso do `getCPF()`, por exemplo, o método vai consultar o valor da variável `CPF` e retornar um número inteiro (representado pelo `int`).

```
getCPF() : int
```

Já os métodos modificadores, que alteram o campo de instância, precisam da informação para alterar os campos da instância ou os atributos do objeto. Essa informação é passada por parâmetro dentro dos parênteses () do método. Como esse método não retorna nenhuma informação, indicamos com a palavra `void`. Por exemplo, o método `setNome()` vai modificar a variável `nome`, por isso precisa saber qual nome deverá atribuir. Essa informação está indicada dentro dos parênteses do método.

```
setNome(nome : String) : void
```

Para uma melhor compreensão dos métodos `getters` e `setters`, vamos desenvolver a classe `Pessoa`. Em seguida, faremos uma instância dessa classe e atribuiremos alguns valores aos nossos atributos. Faremos, também, a leitura desses atributos com os métodos de acesso.

Antes, porém, falta responder a mais uma pergunta: por que o método `Validar_CPF()` está com o sinal de menos (-)? Na verdade,

construímos esse método para mostrar que podemos ter métodos encapsulados como privados e que o objeto não pode acessá-lo.

Então, quem pode?

Os métodos da mesma classe. Isso mesmo! Veja o método `setCPF(cpfo int) : void`. Como sabemos, esse é um método modificador e está atribuindo o valor passado por parâmetro (`cpfo int`) ao campo de instância `CPF` do objeto. Porém, antes de atribuirmos esse valor, o método `setCPF()` pode pedir para um outro método verificar se o CPF informado é válido. Então, nesse momento, o método `setCPF()` chama, dentro dele, o método `Validar_CPF()`, passa por parâmetro o valor do CPF e espera como retorno um booleano, ou seja, um valor falso ou verdadeiro.

```
private boolean Validar_CPF(int cpf) {  
    //API com o sistema da Receita Federal  
    return true;  
}
```

Consegue entender por que, nesse caso, o método `Validar_CPF()` está com acesso privado? O objeto, que foi instanciado fora da classe `Pessoa`, nem precisa saber que ele existe, nem como ele funciona, pois quem precisa verificar se o CPF é válido ou não é o método `setCPF()`.



PARA SABER MAIS

O método `Validar_CPF()` vai consultar o número do CPF no site da Receita Federal, por exemplo. Isso é possível por causa da integração do nosso sistema com o sistema da Receita Federal por Application Programming Interface (API).

Para saber mais, recomendamos a leitura da obra *Ajax, rich internet applications e desenvolvimento web para programadores*, de Paul J. Deitel e Harvey M. Deitel (2008).

Agora, vamos desenvolver, em Java, a classe Pessoa, instanciar o objeto *P* e utilizar os métodos getters e setters.

```
public class Pessoa {  
  
    private String nome;  
    private int CPF;  
    private String email;  
  
    public int getCPF() {  
        return CPF;  
    }  
  
    public void setCPF(int cPF) {  
        if (Validar_CPF(cPF))  
            CPF = cPF;  
    }  
  
    public String getEmail() {  
        return email;  
    }  
  
    public void setEmail(String email) {  
        this.email = email;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome (String nome) {  
        this.nome = nome;  
    }  
  
    private boolean Validar_CPF(int cpf) {  
        //API com o sistema da Receita Federal  
        return true;  
    }  
}
```

Podemos fazer algumas observações em relação a esse código:

- Sempre que você passar por parâmetro uma variável com o mesmo nome da declarada na classe (*email*, por exemplo), use a palavra reservada *this* para fazer referência ao atributo da classe.
Exemplo:

```
this.nome = nome;
```

- Como o método *Validar_CPF()* retorna um *boolean*, podemos efetuar a tomada de decisão *if* no método *setCPF()* desta maneira:

```
if (Validar_CPF(cPF))
    CPF = cPF;
```

- Todos os métodos de acesso (*get*) retornam valor e todos os métodos modificadores (*set*) possuem parâmetros.
- O método *Validar_CPF()* é privado, podendo ser utilizado apenas pela própria classe.

Uma vez desenvolvida a classe *Pessoa*, vamos desenvolver a classe *Principal*, instanciar o objeto *P* e chamar os métodos *getters* e *setters*:

```
public class Principal {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
  
        Pessoa P = new Pessoa();  
  
        P.setNome("Albert");  
        P.setCPF(111222333);  
        P.setEmail("albert@senac.com.br");  
  
        System.out.println("O CPF inserido foi: " + P.getCPF());  
        System.out.println("O nome inserido foi: " +  
P.getNome());  
        System.out.println("O email inserido foi: "  
+ P.getEmail());  
    }  
  
}
```

Note que, nesse código, instanciamos o objeto *P*, inserimos os valores para *Nome*, *CPF* e *Email*, por meio dos métodos modificadores (*setters*), passando os parâmetros necessários, e imprimimos, na tela (pelo comando *System.out.println*), os valores das variáveis de instância do nosso objeto por meio dos métodos de acesso (*getters*) concatenados com textos (entre aspas).

Vamos verificar o resultado ao rodarmos nosso código:

```
0 CPF inserido foi: 111222333  
0 nome inserido foi: Albert  
0 email inserido foi: albert@senac.com.br
```

Note que resolvemos o problema que “deixamos” no final do capítulo anterior, em que o objeto *B2* atribuiu valores diretamente aos atributos da classe *Boletim*.

Por falar em boletim, para você treinar os conceitos deste capítulo, encapsule os atributos e os métodos da classe *Boletim* conforme diagrama de classes e faça os *getters* e os *setters* das classes *Boletim*, *Aluno* e *Professor*. Além disso, tente chamar métodos encapsulados como privados com o objeto instanciado para verificar o que acontece.

Treine bastante e bons estudos!

Considerações finais

Encapsulamento é um dos principais componentes do paradigma orientado a objetos e deve ser usado tanto como boas práticas de desenvolvimento como por recomendação da engenharia de software.

Para encapsular (proteger) nossas classes, métodos e atributos, utilizamos os modificadores de acesso *public* e *private*. Para que os atributos de uma classe não sejam accidentalmente modificados, encapsulamos todos eles como privados. Dessa maneira, os objetos só podem ter acesso às variáveis de instância por meio dos métodos *getters* e *setters*.

Os métodos *getters*, também chamados de métodos de acesso, são utilizados para verificar o estado de uma variável de instância e retornar seu valor. Por isso, devem possuir um tipo de retorno. Os métodos *setters*, também chamados de métodos modificadores, alteram os valores de uma variável de instância e, por isso, devem possuir informações passadas por parâmetro.

Os objetos, por sua vez, só conseguem ter acesso aos métodos públicos de uma classe, que são também chamados de interface da classe.

Referências

- BARNES, David J.; KOLLING, Michael. **Programação orientada a objetos com Java**: uma introdução prática usando o BlueJ. São Paulo: Pearson Prentice Hall, 2010.
- DEITEL, Paul J.; DEITEL, Harvey M. **Ajax, rich internet applications e desenvolvimento web para programadores**. 1. ed. São Paulo: Pearson Prentice Hall, 2008. (Série do Desenvolvedor).
- DEITEL, Paul J.; DEITEL, Harvey M. **Java**: como programar. 8. ed. São Paulo: Pearson Prentice Hall, 2010.
- ENCAPSULAR. In: FERREIRA, Aurélio B. H. **Dicionário Aurélio da língua portuguesa**. 5. ed. Curitiba: Positivo, 2010.
- HORSTMANN, Cay S. **Conceitos de computação em Java**. Porto Alegre: Bookman, 2019.
- HORSTMANN, Cay S.; CORNELL, Gary. **Core Java, volume 1**: fundamentos. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

Capítulo 3

Sobrecarga de métodos

O paradigma orientado a objetos possui algumas grandes vantagens em relação ao paradigma estruturado, que são: reúso de código, facilidade de manutenção e extensibilidade, ou seja, capacidade de reescrita e modificações de classes com pouco impacto ao sistema.

Neste capítulo, vamos estudar um recurso extensível chamado polimorfismo. Focaremos o polimorfismo estático, também chamado de sobrecarga. Vamos relembrar o conceito de assinatura de método e verificar alguns métodos polimórficos do Java.

Finalizamos com um exemplo prático e um desafio para você construir seus próprios métodos polimórficos. Vamos nessa?

1 O recurso “camaleão”

No paradigma orientado a objetos existe um recurso muito útil chamado polimorfismo, que permite muitas vantagens como extensibilidade. De acordo com Deitel e Deitel (2010), com o polimorfismo podemos projetar e implementar sistemas que são facilmente extensíveis, isso significa que novas classes podem ser adicionadas com pouca ou nenhuma modificação às partes gerais do programa.

A palavra “polimorfismo” vem do grego e quer dizer “muitas formas” (*poli* = muito, *morfos* = formas). Daí a nossa analogia com o camaleão, que também assume várias formas de cores para se camuflar e se proteger de predadores.

Em linguagem orientada a objetos (como a do Java), o polimorfismo permite ao desenvolvedor usar o mesmo elemento de várias formas diferentes, ou seja, um objeto pode se comportar de maneiras diferentes ao receber uma mensagem.

Basicamente, existem quatro tipos de polimorfismo:

1. Estático ou sobrecarga
2. Dinâmico ou sobrescrita
3. De inclusão
4. Paramétrico

Entretanto, nesta obra, serão abordados apenas dois tipos: sobrecarga e sobrescrita.

O polimorfismo estático (sobrecarga) acontece quando temos o mesmo método implementado várias vezes na mesma classe. Segundo Horstmann e Cornell (2010), essa capacidade é chamada de sobrecarga ou clonagem. O sobreengamento ocorre se vários métodos tiverem o

mesmo nome, mas parâmetros diferentes. Isso é o que chamamos de assinatura do método que, de acordo com Barnes e Kolling (2010), é o cabeçalho do método. Por exemplo:

```
public void setNome (String nome) {  
}
```

O cabeçalho desse método é a sua assinatura e ele é composto de:

- Modificador de acesso (no caso, o *public*).
- Tipo de retorno (no caso, *void*, ou seja, esse método não retorna nada).
- Nome do método (no caso, *setNome*).
- Parâmetros (no caso, *String nome*). Porém, nem todos os métodos precisam de parâmetros.



PARA SABER MAIS

O polimorfismo dinâmico acontece na herança, quando a subclasse sobrepõe o método original. Nesse caso, o método escolhido se dá em tempo de execução, e não em tempo de compilação. A escolha de qual método será chamado depende do tipo do objeto que recebe a mensagem. Estudaremos o polimorfismo dinâmico no capítulo 5.

Nesse caso, podemos fazer um questionamento: posso ter dois métodos com o mesmo nome? A resposta é: sim. Desde que esse método tenha tipos de parâmetros ou quantidade de parâmetros diferentes.

Seguindo o exemplo anterior, podemos ter os seguintes métodos:

```
1     public void setNome (String nome) {  
2         }  
  
2     public void setNome (String nome, String sobrenome)  
3         {  
4             }  
  
3     public void setNome (double nome) {  
4         }
```

Note que os três métodos possuem o mesmo nome, entretanto cada um tem parâmetros diferentes. O método 1 recebe uma *string* como parâmetro; já o método 2 recebe duas *strings* como parâmetros e o método 3 recebe um *double*. Dessa forma, podemos afirmar que o método *setNome* é um método polimórfico.

A seguir, vamos conhecer um método polimórfico do Java.

2 Métodos polimórficos do Java

Existem várias centenas desses métodos em linguagens orientadas a objetos. O que vamos demonstrar é uma instrução de entrada e saída de dados que trabalha com interfaces e frames (veremos esses conceitos no capítulo 6).

Para a demonstração do método polimórfico, vamos fazer uso de uma classe chamada *JOptionPane*, que faz parte do pacote Swing do javax, e do método polimórfico *showMessageDialog*.



PARA SABER MAIS

Assim como a linguagem de programação Java, um pacote – ou *package*, no paradigma orientado a objetos – é um conjunto de classes localizadas na mesma estrutura hierárquica de diretórios.

Para saber mais sobre o pacote Swing do Java, uma sugestão é a leitura do capítulo 9 da obra *Core Java, volume 1: fundamentos*, de Cay Horstmann e Gary Cornell (2010).

Esse método pode ser demonstrado de várias formas, isto é, ele possui várias assinaturas diferentes. Em nossos exemplos, vamos demonstrar três formas para que você, além de aprender o recurso, utilize em seus projetos.

1. A primeira forma recebe dois parâmetros e é escrita com a seguinte sintaxe:

```
JOptionPane.showMessageDialog(parentComponent, message);
```

Em que:

- *parentComponent*: determina o frame no qual o diálogo vai aparecer. Como ainda não sabemos os parâmetros de interfaces e frames, vamos utilizar o valor *null*. Dessa forma, o próprio Java cria uma caixa de diálogo padrão.
- *message*: é a mensagem que vai aparecer no display. Deve ser colocada entre aspas.

Segue o código Java:

```
import javax.swing.JOptionPane;  
  
public class Principal {
```

(cont.)

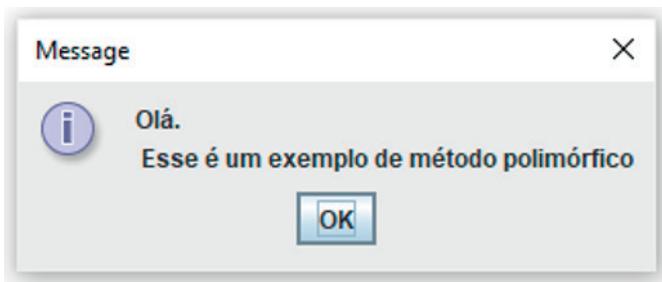
```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
  
    JOptionPane.showMessageDialog(null, "Olá. \n"  
        Esse é um exemplo de método polimórfico");  
  
}  
}
```

Note que, para utilizarmos essa classe, precisamos importar seu pacote:

```
import javax.swing.JOptionPane;
```

Ao rodar o código acima, o Java produz, como resultado, uma caixa de diálogo padrão, localizada no centro da tela e de tamanho fixo (fruto do valor *null* inserido no método), com a mensagem, também inserida no método como parâmetro.

Figura 1 – Exemplo de método polimórfico com dois parâmetros



Se quisermos uma caixa diferente, personalizada, de tamanho diferente, com mais botões, outras cores e posições diferentes em relação à tela, precisaremos configurar um frame e inserir como parâmetro em *parentComponent*.

2. A segunda forma recebe quatro parâmetros e é escrita com a seguinte sintaxe:

```
JOptionPane.showMessageDialog(parentComponent, message,  
title, messageType);
```

Em que:

- *parentComponent*: determina o frame no qual o diálogo vai aparecer. Como ainda não sabemos os parâmetros de interfaces e frames, vamos utilizar o valor *null*. Dessa forma, o próprio Java cria uma caixa de diálogo padrão.
- *message*: é a mensagem que vai aparecer no display. Deve ser colocada entre aspas.
- *title*: é o texto do título da caixa de diálogo. Note que, no exemplo anterior, como não utilizamos esse parâmetro, o Java inseriu a palavra *Message*.
- *messageType*: define o estilo da mensagem. Dependendo do valor atribuído, o gerenciador do Java cria a caixa mais adequada. Esses valores são predefinidos:
 - ERROR_MESSAGE
 - INFORMATION_MESSAGE
 - WARNING_MESSAGE
 - QUESTION_MESSAGE
 - PLAIN_MESSAGE

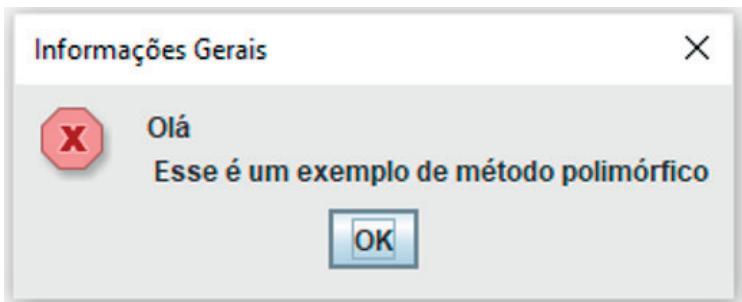
No nosso exemplo, vamos utilizar o estilo *error_message* com *title*: "Informações Gerais". Segue o código Java:

```
import javax.swing.JOptionPane;
```

```
public class Principal {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
  
        JOptionPane.showMessageDialog(null, "Olá  
\n Esse é um exemplo de método polimórfico", "Informações  
Gerais", JOptionPane.ERROR_MESSAGE);  
  
    }  
}
```

Quando rodamos esse código, obtemos o seguinte resultado:

Figura 2 – Exemplo de método polimórfico com quatro parâmetros



Note que o texto “Informações Gerais” aparece como título da caixa de diálogo e que foi criada uma figura vermelha com um X, resultado do parâmetro de `error_message`. Como sugestão, crie outros diálogos, utilizando outros estilos de mensagens e verifique os resultados.

3. A terceira forma de representar nosso método polimórfico recebe cinco parâmetros e é escrita com a seguinte sintaxe:

```
JOptionPane.showMessageDialog(parentComponent, message,  
title, messageType, icon);
```

Em que:

- *parentComponent*: determina o frame no qual o diálogo vai aparecer. Como ainda não sabemos os parâmetros de interfaces e frames, vamos utilizar o valor *null*. Dessa forma, o próprio Java cria uma caixa de diálogo padrão.
- *message*: é a mensagem que vai aparecer no display. Deve ser colocada entre aspas.
- *title*: é o texto do título da caixa de diálogo.
- *messageType*: define o estilo da mensagem. Dependendo do valor atribuído, o gerenciador do Java cria a caixa mais adequada. Esses valores são predefinidos:
 - ERROR_MESSAGE
 - INFORMATION_MESSAGE
 - WARNING_MESSAGE
 - QUESTION_MESSAGE
 - PLAIN_MESSAGE
- *icon*: um ícone será criado na caixa de diálogo para ajudar o usuário a identificar que tipo de mensagem está sendo exibido. Um ícone é uma interface também e pode ser obtido de várias maneiras como por um lable, um frame ou uma imagem PNG do seu computador.

Neste exemplo, vamos utilizar um emoji de celular. Basta salvar a figura em uma pasta do seu computador e utilizar um objeto do tipo *Icon*, passando a localização da figura como parâmetro.



PARA PENSAR

Nossa intenção, neste capítulo, não é falar de interface, frame, lable ou importação de imagem ou ícone. Estamos apenas utilizando o código para mostrar e evidenciar o polimorfismo de método.

Caso você se interesse pelo assunto, sugerimos analisar o que fizemos e tentar reproduzir na sua máquina. Pense no que foi feito e nos conceitos envolvidos, como instância de objeto e passagem de parâmetro.

Segue o código Java:

```
import javax.swing.Icon;
import javax.swing.JOptionPane;

public class Principal {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Icon icone = new javax.swing.ImageIcon
            ("C:\\\\Users\\\\55119\\\\Pictures/icon.png");

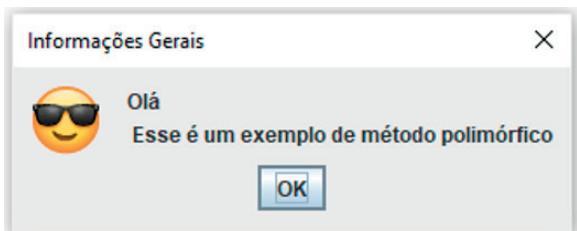
        JOptionPane.showMessageDialog(null, "Olá \n
Esse é um exemplo de método polimórfico", "Informações
Gerais", JOptionPane.ERROR_MESSAGE, icone);

    }

}
```

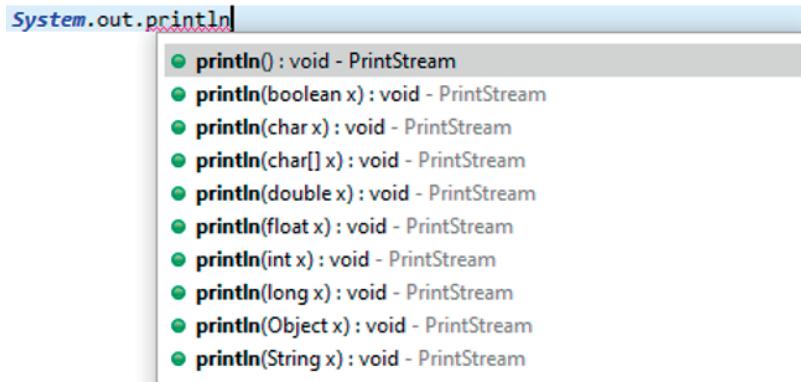
Note que, para esse exemplo, tivemos de importar a classe *Icon*. Criamos um objeto chamado *icone* e o inicializamos com o método *ImageIcon* passando, como parâmetro, o endereço da nossa figura. Quando rodamos o código, o resultado é este:

Figura 3 – Exemplo de método polimórfico com cinco parâmetros



Outro exemplo de método polimórfico que você já conhece é o `println()`. Esse método possui dez assinaturas diferentes, conforme a figura 4. Para utilizá-lo, basta escolher o tipo de parâmetro que você deseja.

Figura 4 – Assinaturas do método polimórfico `println()`



Agora que sabemos sobre polimorfismo de método, que tal construirmos nossos próprios métodos polimórficos?

3 Construindo métodos polimórficos

Lembra-se da nossa classe `Boletim`? Essa classe possui o método `calcular_media()`. Vamos supor a seguinte situação: precisamos calcular a média dos alunos, porém temos alunos de graduação e pós-graduação e as regras para o cálculo da média são diferentes para os dois tipos de alunos:

Para os alunos de graduação, o cálculo da média é dado pela fórmula:

$$MS = (AM * 0,3) + (AC * 0,2) + (AS * 0,5)$$

Em que:

AM = avaliação multidisciplinar (vale 30% da média);

AC = avaliação continuada (vale 20% da média);

AS = avaliação semestral (vale 50% da média);

MS = média semestral.

Já para os alunos de pós-graduação, o cálculo da média é dado pela seguinte fórmula:

$$MM = (PM * 0,4) + (AS * 0,6)$$

Em que:

PM = projeto multidisciplinar (vale 40% da média);

AS = avaliação semestral (vale 60% da média);

MM = média modular.

Sabendo das regras ou, em engenharia de software, conhecendo os requisitos, vamos criar nossos métodos *calcular_nota()* e passar, como parâmetros, três variáveis (quando o cálculo diz respeito aos alunos de graduação) e duas variáveis (quando nos referimos aos alunos de pós-graduação).

```
public double calcular_media(double AM, double AC, double AS) {  
    double media = AM * 0.3 + AC * 0.2 + AS * 0.5;  
    return media;  
}  
  
public double calcular_media(double PM, double AS) {  
    double media = PM * 0.4 + AS * 0.6;  
    return media;  
}
```



IMPORTANTE

Se você construir um método com a mesma assinatura, na mesma classe, o Java avisa sobre o erro dizendo que existe um método duplicado na sua classe.

Uma vez criados nossos métodos, vamos fazer duas instâncias de *Boletim* e passar os parâmetros corretos. Como já aprendemos a usar os métodos de entrada e saída *showMessageDialog()* da classe *JOptionPane*, vamos utilizá-los.

```
import javax.swing.JOptionPane;

public class Faculdade {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Boletim B1 = new Boletim();
        Boletim B2 = new Boletim();

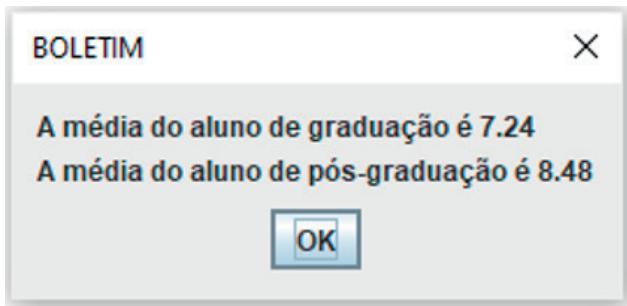
        double m_grad = B1.calcular_media(9.5, 4.7,
6.9);
        double m_pos = B2.calcular_media(7.4, 9.2);

        JOptionPane.showMessageDialog(null, "A média
do aluno de graduação é " + m_grad + "\nA média do aluno
de pós-graduação é " + m_pos, "BOLETIM", JOptionPane.
PLAIN_MESSAGE);

    }
}
```

Ao ser rodado, esse código produz o seguinte resultado:

Figura 5 – Painel de saída de dados – Boletim



Você acabou de criar seus primeiros métodos polimórficos. No capítulo 5, falaremos sobre polimorfismo dinâmico construído em classes diferentes e executados com o uso de herança.

Por enquanto, o seu desafio é reproduzir o exemplo acima. Acabe de criar a classe *Boletim* e execute o código, verificando os resultados. Tente também introduzir um ícone à caixa de diálogo. Quanto mais você treinar, melhor ficará.

Considerações finais

Neste capítulo, estudamos o conceito de polimorfismo que, na tradução, significa “muitas formas”. Ou seja, apresentamos várias formas de utilizar um método de uma classe.

Existem quatro tipos de polimorfismo: estático (ou sobrecarga), dinâmico (ou sobrescrita), inclusão e paramétrico. Nossa foco foi a sobre-carga de métodos e, para que isso aconteça, no Java, precisamos nos atentar à assinatura do método, ou seja, a seu cabeçalho. Uma sobre-carga ocorre quando temos vários métodos com o mesmo nome, porém com parâmetros diferentes (ou pelos tipos de parâmetros ou pela quantidade de parâmetros).

Além disso, vimos um método polimórfico do Java chamado `showMessageDialog()` da classe `JOptionPane` e do pacote Swing. Para utilizar esses exemplos, tivemos que importar essa classe desse pacote. Vimos também como passar os parâmetros de forma correta para cada situação e aprendemos uma curiosidade que foi a inserção de um objeto `Icon`.

Por último, construímos um método polimórfico `calcular_media()` e testamos com dois objetos, exibindo o resultado na caixa de diálogo do `JOptionPane`. O método `calcular_media()` teve como assinatura quantidades diferentes de parâmetros.

Referências

BARNES, David J.; KOLLING, Michael. **Programação orientada a objetos com Java**: uma introdução prática usando o BlueJ. São Paulo: Pearson Prentice Hall, 2010.

DEITEL, Paul; DEITEL, Harvey. **Java**: como programar. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

HORSTMANN, Cay S.; CORNELL, Gary. **Core Java, volume 1**: fundamentos. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

Capítulo 4

Construtores

Um dos principais conceitos do paradigma orientado a objetos é a instância de objetos, que são variáveis cujo tipo é uma classe. Ao instanciarmos um objeto, devemos chamar seu método construtor para que algumas condições iniciais sejam cumpridas.

O construtor de uma classe sempre é chamado obrigatoriamente na inicialização do objeto e pode ser um método vazio e sem parâmetros.

Neste capítulo, vamos estudar os conceitos dos métodos construtores, como criá-los e utilizá-los em nosso sistema. Além disso, vamos verificar como funcionam as sobrecargas de métodos construtores.

Por fim, vamos demonstrar vários exemplos de utilização dos construtores e verificar uma forma interessante e bastante útil de aplicação de sobrecarga de construtores que facilita a manutenção do nosso código.

1 Tudo começa do início

Como sabemos, dois dos conceitos mais importantes no paradigma orientado a objetos são as classes e os objetos. Aprendemos nos capítulos anteriores a como construir uma classe e como instanciar um objeto, além de utilizar alguns métodos, como *getters* e *setters*.

No entanto, antes mesmo de um objeto chamar explicitamente um método de uma classe (como *B1.calcular_media(9.5, 4.7, 6.9)*), na criação e inicialização desse objeto já aconteceu uma chamada a uma instrução específica: o construtor.

De acordo com Horstmann e Cornell (2010), o construtor inicializa o objeto de uma forma específica, conforme a necessidade inicial desse objeto. Ou seja, podemos criar um objeto com alguns valores iniciais, “settando” alguns valores em suas variáveis de instância.

Por exemplo, imagine a seguinte situação: você está com fome e resolve fritar um ovo para saciá-la. Por mais que se saiba fritar um ovo, a pergunta a ser feita é: por onde você começaria? Vamos a algumas possibilidades:

1. verificar se tem ovos;
2. verificar se tem uma frigideira;
3. verificar se tem gás no fogão.

Não existe, nas possibilidades acima, uma maneira correta ou melhor de começar a fritar o ovo. Todas as condições iniciais estão corretas e são válidas. É exatamente isso o que faremos com nosso objeto ao instanciá-lo: dar uma condição inicial para começar nossas tarefas com uma condição mais adequada ao momento.

Vamos relembrar, entretanto, o que é instanciar um objeto e como o inicializamos. No capítulo 3, instanciamos o objeto *B1* (do tipo *Boletim*) da seguinte maneira:

```
Boletim B1 = new Boletim();
```

Vamos interpretar o que está ocorrendo nessa linha de comandos. Na verdade, temos duas instruções diferentes acontecendo e vamos separá-las para explicar onde está ocorrendo a inicialização do objeto.

1

```
Boletim B1;
```

Nessa instrução, estamos declarando uma “variável” chamada *B1*, porém o tipo dessa variável é o nome de uma classe. Isso significa que *B1* é um objeto da classe *Boletim* e, portanto, uma instância da classe *Boletim*, cujos comportamento e estado são definidos pela sua classe. Note que apenas criamos o objeto, e o Java já exige que esse objeto seja inicializado para começarmos o seu uso.

2

```
B1 = new Boletim();
```

Nessa instrução, estamos inicializando o objeto *B1* com um método especial chamado construtor. Essa instrução é chamada somente quando um novo objeto é criado e gera a condição inicial para nosso objeto, de acordo com nossa necessidade.



PARA PENSAR

Uma pergunta que você pode estar se fazendo agora é: mas eu não criei esse método construtor, como ele está sendo usado para iniciar meu objeto?

Vamos responder a essa pergunta no próximo tópico.

2 Utilizando métodos construtores

Não criamos, ainda, nenhum construtor, mas já conseguimos inicializar alguns objetos ao longo deste livro. Como isso é possível? O compilador Java cria um método construtor quando nós não o fazemos, chamado de construtor-padrão.

Uma das características que um método construtor deve ter é possuir o mesmo nome de sua classe e não possuir tipo de retorno em sua assinatura. Quando dizemos “não possui retorno”, não significa que é *void*. Simplesmente essa instrução não possui nenhum retorno e isso diferencia um construtor de um método. No corpo desse método, quando criado pelo compilador, não existe nenhum comando, e o Java inicializa as variáveis de instância do objeto com 0 (zero) para as variáveis numéricas, *false* para as booleanas e *NULL* para as variáveis de texto.



PARA SABER MAIS

Um compilador é um programa de computador escrito em alguma linguagem de programação utilizado para escrevermos nossos programas e algoritmos. O compilador verifica as regras de programação da linguagem que estamos utilizando, por exemplo, sintaxe e semântica, e traduz as instruções em linguagem de máquina para que o processador possa executar nosso código.

O método construtor da classe *Boletim*, criada pelo compilador, foi esse:

```
public Boletim() {  
}
```

Note que esse método está encapsulado com o modificador de acesso *public* e não possui tipo de retorno. Além disso, deve ter o mesmo nome da sua classe e pode ou não receber parâmetros e possuir instruções em seu corpo. Do jeito que esse método foi criado, as variáveis de instâncias, ao executarmos o comando *B1 = new Boletim();* considerando as variáveis da classe *Boletim*, demonstrada abaixo, foram:

```
public class Boletim {  
    private float nota;  
    private int frequencia;  
    private String status;  
}  
]  
nota = 0;  
frequencia = 0;  
status = NULL;
```

Ou seja, se nós quiséssemos obter o mesmo resultado criando nosso método construtor, bastaria escrever, logo após as declarações das variáveis na classe *Boletim*, o código *public Boletim(){}*, ficando desta maneira:

```
public class Boletim {  
    private float nota;  
    private int frequencia;  
    private String status;  
  
    public Boletim() {  
    }  
}
```

Mas e se nossa intenção fosse inicializar essas variáveis com valores específicos? Por exemplo, todos os alunos começarem com nota 10, frequência 100% e status APROVADO. Conforme os seus desempenhos, eles poderiam ir perdendo a nota, da mesma maneira que, conforme as suas faltas, eles poderiam ir perdendo frequência. Dessa forma, no final do período letivo, saberíamos seu *Status*. Vamos passar o *status* por parâmetro criando o seguinte método construtor:

```
public Boletim(String st) {  
    nota = 10;  
    frequencia = 100;  
    status = st;  
}
```

Repare que passamos o parâmetro *Status* apenas para diferenciar do construtor anterior, mas poderíamos não ter passado nenhum parâmetro para esse método. Entretanto, poderíamos enviar todos os valores, indicados pelo usuário, por exemplo. Imagine que o usuário decida os valores com que o objeto deve começar. Para isso, devemos criar o método construtor passando todos esses parâmetros e os atribuindo às variáveis da classe. Nossa método construtor ficaria assim:

```
public Boletim(float n, int f, String s) {  
    nota = n;  
    frequencia = f;  
    status = s;  
}
```

No método *MAIN()* da classe principal, nossa chamada ao método construtor ficaria assim:

```
float nota;  
int frequencia;  
String status;  
  
Scanner ler = new Scanner (System.in);  
  
System.out.println("Insira a nota de início");  
nota = ler.nextFloat();  
System.out.println("Insira a frequência de início");  
freq = ler.nextInt();  
System.out.println("Insira o status de início");  
st = ler.next();  
  
Boletim B1;  
B1 = new Boletim(nota, freq, st);
```

Repare alguns novos comandos nesse código:

1 Scanner ler = new Scanner (System.in);

Esse comando é uma instrução de entrada de dados usada no Java. A classe `Scanner` deve ser importada para sua utilização (`import java.util.Scanner;`). `Ler` é um objeto do tipo `Scanner` e foi inicializado com a função `System.in`, passada por parâmetro em seu método construtor.

2 nota = ler.nextFloat("Insira a nota de início");

O objeto `ler` chamou o método `nextFloat` que lê, via teclado, um dado do tipo `float` e o atribui à variável `nota`.

3 freq = ler.nextInt("Insira a frequência de início");

O objeto `ler` chamou o método `nextInt` que lê, via teclado, um dado do tipo `Int` e o atribui à variável `freq`.

4 st = ler.next("Insira o status de início");

O objeto `ler` chamou o método `next` que lê, via teclado, um dado do tipo `String` e o atribui à variável `st`.

5 B1 = new Boletim(nota, freq, st);

O objeto `B1` foi inicializado com os valores passados por parâmetro no método construtor.

A partir desse momento, o objeto `B1` está criado e inicializado, podendo ser usado no nosso sistema. Mas e se quisermos ter os três

métodos construtores na nossa classe e, a cada instância de objetos do tipo *Boletim*, inicializar de formas diferentes, dependendo da situação?

Essa alternativa é perfeitamente possível e vamos mostrar como fazer isso.

3 Sobrecarga de métodos construtores

Além do construtor-padrão (criado pelo compilador) ou do método construtor criado por nós, programadores, podemos ter vários métodos construtores na mesma classe. Segundo Deitel e Deitel (2010), a sobrecarga de construtores permite que os objetos sejam inicializados de diferentes maneiras. Para que a sobrecarga aconteça, basta criar vários construtores com assinaturas diferentes.



IMPORTANTE

Lembre-se de que o compilador diferencia assinaturas de métodos pelo número e pelos tipos de parâmetros em cada assinatura.

Vamos criar os três métodos construtores do nosso exemplo anterior na mesma classe e utilizá-los para inicializar três objetos. Lembrando que, para usarmos sobrecarga de construtores, temos de criá-los com assinaturas diferentes.

Vejamos como ficará nossa classe *Boletim* (sem considerar os métodos *getters* e *setters*):

```
public class Boletim {  
  
    public Boletim(float n, int f, String s) {
```

(cont.)

```

        nota = n;
        frequencia = f;
        status = s;
    }

    public Boletim (String st) {
        nota = 10;
        frequencia = 100;
        status = st;
    }

    public Boletim() {
    }
}

```

Repare que temos três métodos construtores com assinaturas diferentes, caracterizando a sobrecarga. Agora, vamos conferir a classe principal com o método *main()* instanciando os objetos e usando os métodos construtores adequadamente:

```

import java.util.Scanner;

public class Principal {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        float nota;
        int freq;
        String st;

        Scanner ler = new Scanner (System.in);

        System.out.println("Insira a nota de
início");
        nota = ler.nextFloat();
        System.out.println("Insira a frequência de
início");
    }
}

```

(cont.)

```

        freq = ler.nextInt();
        System.out.println("Insira o status de
início");
        st = ler.next();

        Boletim B1 = new Boletim(nota, freq, st);
        Boletim B2 = new Boletim();
        Boletim B3 = new Boletim("APROVADO");
    }
}

```

Observe as três formas diferentes de inicialização dos objetos *B1*, *B2* e *B3*. O objeto *B1* passou três parâmetros inseridos pelo usuário, via teclado, com o auxílio do objeto *ler* do tipo *Scanner*. O objeto *B2* chama o construtor vazio (padrão) e terá suas variáveis de instância zeradas (variáveis numéricas) e com o valor *NULL* (variável de texto). E o objeto *B3* está chamando o construtor, que passa apenas um parâmetro (*status*) e atribui o valor *10* à nota e *100* à frequência.



PARA SABER MAIS

O Java permite criar métodos com o mesmo nome da classe e que não são os construtores. Para que isso seja possível, basta o método ter um tipo de retorno. O Java determina quais são os construtores localizando os métodos que têm o mesmo nome da classe e não especificam um tipo de retorno.

Vamos fazer agora uma utilização interessante dos construtores em sobrecarga, reutilizando código e facilitando a manutenção. Vamos construir os métodos *getters* e *setters* na classe *Boletim* e criar um método chamado *setBoletim()*, que atribuirá todos os valores às variáveis de instância dos objetos.

Vejamos a classe *Boletim* completa:

```
public class Boletim {  
    private float nota;  
    private int frequencia;  
    private String status;  
  
    public Boletim(float n, int f, String s) {  
        setBoletim(n,f,s);  
    }  
    public Boletim (String st) {  
        this(10,100,st);  
    }  
  
    public Boletim() {  
        this(0,0,null);  
    }  
  
    public void setBoletim(float n, int f, String st) {  
        setNota(n);  
        setFrequencia(f);  
        setStatus(st);  
    }  
  
    public float getNota() {  
        return nota;  
    }  
    public void setNota(float nota) {  
        this.nota = nota;  
    }  
  
    public int getFrequencia() {  
        return frequencia;  
    }  
  
    public void setFrequencia(int frequencia) {  
        this.frequencia = frequencia;  
    }  
    public String getStatus() {  
        return status;  
    }  
  
    public void setStatus(String status) {  
        this.status = status;  
    }  
}
```

Com a classe construída dessa maneira, o método `setBoletim()` é o que vai atribuir os valores às variáveis de instância dos objetos. Os construtores `Boletim()` e `Boletim(String st)` fazem uso da referência `this`, que só é permitida como a primeira instrução no corpo de um método construtor e é utilizada para invocar o construtor que recebe três parâmetros, que, por sua vez, chama o método `setBoletim()` e passa os parâmetros recebidos. Dessa forma, se houver alguma alteração nas variáveis, basta mudar o método `setBoletim()`, facilitando a manutenção do nosso código.

Vamos inserir o método *imprimir* à classe `Boletim` para imprimir os objetos que foram inicializados com os diferentes construtores e refazer nossa classe principal para chamar o método *imprimir* em cada objeto. Verifiquemos, então, o método *imprimir()*:

```
// inserir esse método na classe Boletim
public void imprimir() {
    System.out.println(getNota() + "\n" +
    getFrequencia() + "\n" + getStatus() + "\n");
}
```

E nossa classe principal ficará assim:

```
import java.util.Scanner;

public class Principal {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        float nota;
        int freq;
        String st;
        Scanner ler = new Scanner (System.in);
```

(cont.)

```

System.out.println("Insira a nota de início");
    nota = ler.nextFloat();
    System.out.println("Insira a frequência de
início");
    freq = ler.nextInt();
    System.out.println("Insira o status de
início");
    st = ler.next();

    Boletim B1 = new Boletim(nota, freq, st);
    Boletim B2 = new Boletim();
    Boletim B3 = new Boletim("APROVADO");

    System.out.println("Uso do construtor com 3
parâmetros: ");
    B1.imprimir();
    System.out.println("Uso do construtor com
nenhum parâmetro: ");
    B2.imprimir();
    System.out.println("Uso do construtor com 2
parâmetros: ");
    B3.imprimir();

}
}

```

Ao rodar esse código, teremos o seguinte resultado:

```

Uso do construtor com 3 parâmetros:
5.0
75
REPROVADO

Uso do construtor com nenhum parâmetro:
0.0
0
null

Uso do construtor com 2 parâmetros:
10.0
100
APROVADO

```

Você acabou de estudar as formas de criar métodos construtores e como podemos utilizar vários desses métodos, em sobrecarga, para inicializar nossos objetos instanciados.

Considerações finais

Neste capítulo, abordamos o conceito de método construtor, importante recurso do paradigma orientado a objetos.

Sempre que criamos um objeto temos que, obrigatoriamente, inicializá-lo, ou seja, criar uma situação inicial para o objeto. Se não criamos o método construtor, o compilador Java cria um método-padrão, sem nenhum parâmetro e sem nenhuma instrução em seu corpo. Dessa forma, as variáveis numéricas são inicializadas com valor zero, as variáveis booleanas com valor falso e as variáveis textuais com valor null.

Porém, podemos criar quantos métodos construtores quisermos inicializando os objetos instanciados com vários cenários diferentes. Quando uma classe possui mais de um método construtor, temos sobrecarga de construtores e eles são diferenciados pelas assinaturas.

Por fim, demos um exemplo usando três métodos construtores, instanciando três objetos do tipo *Boletim* e imprimindo seus resultados.

Referências

DEITEL, Paul; DEITEL, Harvey. **Java**: como programar. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

HORSTMANN, Cay S.; CORNELL, Gary. **Core Java, volume 1**: fundamentos. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

Capítulo 5

Herança

Ao trabalhar com projetos orientados a objetos, talvez a maior dificuldade que exista para os desenvolvedores é reutilizar de maneira inteligente o código já produzido (GAMMA et al., 2009). Uma das maneiras que desenvolvedores de softwares encontram para reutilizar o código, dentro dos conceitos de programação orientada a objetos, é o uso da técnica de herança. A herança constitui em criar classes a partir de classes existentes, mantendo a possibilidade de aprimorar suas habilidades (DEITEL; DEITEL, 2010).

1 Entendimento sobre herança

Uma forma de entender o comportamento de herança em um código é pensar na herança genética entre pessoas de determinada família. Os pais sempre deixam para os filhos algumas características, que acabam configurando algumas aparências e comportamentos dentro da família. Entretanto, os filhos não são exatamente a cópia dos pais, eles possuem novas habilidades que podem aprender ao longo do tempo e ainda aprimorar alguma já existente, em outras palavras, que foi herdada.

O conceito de herança aplicado ao desenvolvimento de software é interessante, pois possibilita economizar tempo de construção, já que serão criados objetos com base nos existentes, que estão testados e já têm uma qualidade alta de boas práticas (DEITEL; DEITEL, 2010). A classe que serve de base para a criação de objetos é chamada de superclasse, mas também pode ser reconhecida na literatura como geral ou classe mãe. As classes que herdam as características da superclasse são chamadas de subclasses ou, ainda, de específicas ou classes filhas (HORSTMANN; CORNELL, 2010).

Subclasses têm a capacidade de se tornarem superclasses a partir do momento que servem de base para a criação de futuros objetos. Elas ainda podem agregar novos comportamentos ou modificar os herdados da superclasse, assim como suas características, para melhor solucionar o problema proposto (DEITEL; DEITEL, 2010). Em virtude desse aperfeiçoamento da subclasse em relação à superclasse, é comum encontrar na literatura a denominação “especialização”, quando tratamos do termo herança.

A combinação obtida a partir da derivação das classes dentro de um sistema gera uma hierarquia, de maneira bem parecida com uma árvore genealógica. Analisando a estrutura hierárquica dentro das famílias de objetos no Java, podemos concluir que as superclasses são as classes mais gerais (genéricas) e as subclasses são aquelas mais

específicas (especializadas) dentro de nosso sistema (DEITEL; DEITEL, 2010). Assim, podemos encontrar superclasses diretas, que são aquelas diretamente ligadas com suas subclasses, e indiretas, que, em termos mais simples, seriam as classes netas, em que existe uma classe intermediária. Essa relação hierárquica entre as classes define o que conhecemos como herança, formando uma família de classes em nosso sistema (DEITEL; DEITEL, 2010; HORSTMANN; CORNELL, 2010).

Caso você esteja pensando que porque foi mencionada uma árvore genealógica pode existir então a herança “genética” de mais de uma classe diretamente, isso não ocorre, pois o Java permite apenas uma única herança. Algumas linguagens como C++ dão suporte ao que chamamos de herança múltipla, em que pode existir um pai e uma mãe para uma classe filha (DEITEL; DEITEL, 2010).

Depois de compreender o que é a herança e como é o relacionamento hierárquico entre as classes, é importante entender como, durante a modelagem, podemos identificar o tipo de relacionamento a ser aplicado entre elas. E identificar isso é muito simples. Imagine que você tenha duas classes: a classe A e a classe B. Ao analisar a classe B, você diz que ela é um objeto da classe A; e este é um relacionamento de herança em que A é a superclasse e B, a subclasse. A figura 1 apresenta o conceito de herança por meio de um diagrama UML.

Figura 1 – Representação UML de herança (generalização)



Se, por acaso, você iniciar a análise e disser que B tem um objeto de A, então é apenas uma composição simples entre objetos, e não herança. A figura 2 apresenta a representação de composição entre as classes A e B por meio de um diagrama UML.

Figura 2 – Representação UML de composição



Mas qual é o objetivo de todas essas relações entre objetos? Bem, alguns especialistas alegam que existirá um momento no desenvolvimento de software em que a criação de novos softwares será feita completamente com base em componentes reutilizáveis de software. Isso tornará o tempo de construção extremamente rápido e efetivo, mas isso ainda é apenas um desejo (DEITEL; DEITEL, 2010; HORSTMANN; CORNELL, 2010). Ao longo deste capítulo, discutiremos mais sobre herança e veremos na prática como trabalhar com esse relacionamento entre objetos.

2 Construindo a primeira família de objetos

Existem diversos exemplos para trabalharmos com herança em nossos estudos. Os mais utilizados na literatura estão relacionados à hierarquia empresarial em um sistema de folha de pagamento, que, por ser um exemplo conhecido para todos, torna-se bem didático. Mas antes de abordarmos esse exemplo, vamos pensar em um para o produto de financiamento de uma instituição financeira. É possível dizer que podemos ter uma classe *CasaPropriaFinanciamento*, que não deixa de ser uma especialização de uma classe *Financiamento*, assim como *CarroFinanciamento* e *MelhoriaEmpresaFinanciamento*. Todos os

exemplos são especializações de um financiamento mais comum, do que podemos concluir que *Financiamento* pode ser qualquer tipo de financiamento existente em uma instituição financeira. Portanto, é um financiamento genérico.

Vamos pensar em alguns exemplos e relacioná-los no quadro 1, apresentado a seguir.

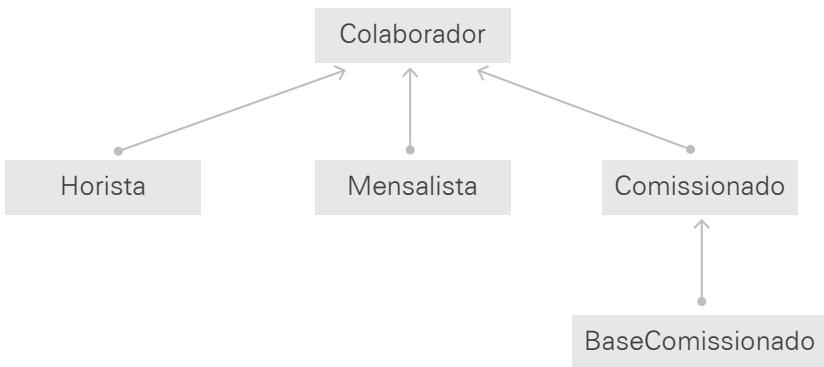
Quadro 1 – Exemplos de superclasses e subclasses com relação de herança

SUPERCLASSES (GENÉRICAS)	SUBCLASSES (ESPECIALIZADAS)
Estudante	Graduacao, PosGraduacao
PosGraduacao	LatoSensu, StrictoSensu
Formato	Circulo, Quadrilatero, Triangulo, Cubo, Esfera
Financiamento	Casa, Carro, Empresarial
ContaBancaria	ContaCorrente, ContaPoupanca, ContaInvestimento, ContaSalario
Colaborador	Horista, Mensalista, Comissionado, BaseComissionado

Perceba que, no quadro 1, é possível identificar diversos relacionamentos que nos permitem trabalhar na estrutura de herança. Com isso, podemos afirmar que todo objeto na coluna *subclasses* é um objeto pertencente à coluna *superclasses*, dentro de sua própria linha. Assim como uma família de seres vivos, as superclasses podem ter diversas subclasses, sendo a representação-base de qualquer outra classe associada.

Agora que os exemplos foram dados, vamos focar a linha do objeto *Colaborador* como superclasse, apresentado na última linha do quadro 1. A figura 3 apresenta a estrutura de árvore hierárquica formada a partir dessas informações.

Figura 3 – Diagrama hierárquico das classes relacionadas ao *Colaborador* de uma empresa



Observe na figura 3 que existe uma hierarquia de relacionamento entre as classes, em que as classes *Horista*, *Mensalista* e *Comissionado* derivam (herdam) diretamente da classe *Colaborador*. Indiretamente, existe a herança da classe *BaseComissionado*, que não deixa de ser um relacionamento “é um” com *Colaborador*. Porém, nesse caso existe a classe *Comissionado* entre as duas, tornando o relacionamento “é um” de *BaseComissionado* indireto com *Colaborador* e direto com *Comissionado*.

Após estabelecer os pontos principais do relacionamento entre os objetos propostos, é importante entender o que deverá haver entre cada uma das classes existentes. *Colaborador* será uma classe simples que possui como características o nome e o departamento do colaborador. Ambos os atributos são do tipo *String*. A classe *Horista* representa os colaboradores da empresa que recebem de acordo com as horas trabalhadas, por exemplo, trabalhadores com contrato de pessoa jurídica ou prestadores de serviços.

Os trabalhadores que recebem um valor fixo pela jornada de trabalho são representados por meio da classe *Mensalista*. Seguindo para a próxima classe da hierarquia, encontramos a classe *Comissionado*. Essa

classe pode ser aplicada a quaisquer colaboradores que trabalham por comissão, como os profissionais da área comercial da empresa. Por fim, a classe *BaseComissionada*, que recebe um valor mínimo e ainda um valor a mais das comissões das vendas, por exemplo.

Com o escopo das classes definido, é hora de colocar a mão no código e efetuar a implementação de cada uma delas, começando pela classe *Colaborador*.

```
1  public class Colaborador {
2      private String nome;
3      private String departamento;
4
5      public Colaborador(String nome, String departamento) {
6          this.nome = nome;
7          this.departamento = departamento;
8      }
9
10     public String getNome() {
11         return this.nome;
12     }
13
14     public String getDepartamento() {
15         return this.departamento;
16     }
17
18     @Override
19     public String toString() {
20         return String.format("Colaborador: %s\nDepartamento: %s",
21                             this.nome, this.departamento);
22     }
23 }
```

Foram definidos dois atributos privados do tipo *String* que representam o nome e o departamento do colaborador (linhas 2 e 3). O construtor da classe foi declarado, recebendo dois parâmetros que representam o nome e o departamento a serem inseridos nos atributos da classe (veja o código entre as linhas 5 e 8). No intervalo das linhas 10 e 16, estão implementados os métodos *getters* referentes a cada atributo da classe. Por fim, entre as linhas 18 e 22, temos a implementação de um método *toString*, que deve formatar a maneira como o objeto *Colaborador* deve ser representado em um texto. A anotação *@Override* presente na linha 18 e a estrutura do método *toString* serão discutidas em detalhes mais adiante neste capítulo.

Na sequência, é apresentada a classe *Horista*, que possui a quantidade de horas trabalhadas e o valor por hora que o colaborador recebe. Além disso, o método *salario* é inserido para realizar o cálculo do salário. Veja a implementação da classe *Horista* a seguir:

```
1  public class Horista extends Colaborador{
2      private int horasTrabalhadas;
3      private double valorPorHora;
4
5      public Horista(String nome, String departamento, double
valorPorHora) {
6          super(nome, departamento);
7          this.valorPorHora = valorPorHora;
8      }
9
10     public void setHorasTrabalhadas(int horasTrabalhadas) {
11         this.horasTrabalhadas = horasTrabalhadas;
12     }
13
14     public int getHorasTrabalhadas() {
15         return this.horasTrabalhadas;
16     }
17
18     public double getValorPorHora() {
19         return this.valorPorHora;
20     }
21
22     public double salario() {
23         return this.horasTrabalhadas * this.valorPorHora;
24     }
25
26     @Override
27     public String toString() {
28         return String.format("%s\nHoras trabalhadas: %d\n" +
29                             "Valor por hora: R$ %.2f\nSalário: R$ %.2f",
30                             super.toString(), this.horasTrabalhadas,
31                             this.valorPorHora, this.salario());
32     }
33 }
```

Na linha 1 do código da classe *Horista* temos uma novidade: a palavra reservada *extends* seguida do nome da classe *Colaborador*. Isso significa que a classe *Horista* é uma extensão da *Colaborador*. O relacionamento estabelecido “é um”, portanto criou-se uma herança entre as classes *Colaborador* e *Horista*. São definidos dois atributos *horasTrabalhadas* e *valorPorHora*, ambos privados (linhas 2 e 3). No construtor da classe *Horista* (da linha 5 à 8), são esperados três parâmetros: *nome*, *departamento* e *valorPorHora*. Nesse momento, o primeiro pensamento que vem é: “não foi definido nem o atributo *nome* nem o atributo *departamento*, mas, tudo bem, basta incluí-los”. E a resposta para

esse pensamento é não, não existe a necessidade de incluir ambos os atributos, eles existem na superclasse *Colaborador*, e como *Horista* é uma subclasse de *Colaborador*, ela também possui os atributos *nome* e *departamento*.

Para atribuir os valores que foram declarados por meio da classe *Colaborador*, vamos utilizar o construtor. A utilização do construtor é possível apenas pela substituição do construtor-padrão do Java (veja a classe *Colaborador* entre as linhas 5 e 8). Como não existe mais o construtor-padrão, é obrigatório chamar o construtor da classe *Colaborador*. Para realizar essa tarefa, utiliza-se a palavra *super* (referência à superclasse) e, entre parênteses, enviam-se os parâmetros desejados na ordem estabelecida pela declaração na superclasse (linha 6). Entre as linhas 10 e 20 encontram-se os métodos getters e setters da classe.

O método *salario*, declarado entre as linhas 22 e 24, calcula o salário do colaborador horista, que no caso é a multiplicação entre a quantidade de horas trabalhadas pelo valor por hora. A classe *Horista* é finalizada com a implementação do método *toString* entre as linhas 26 e 32. Repare que existe uma chamada do método *toString* da classe *super* na lógica, confira a linha 30. Assim, o resultado já produzido pela superclasse *Colaborador* é reutilizado na subclasse *Horista*, compondo o resultado. A próxima subclasse implementada é a *Mensalista*, confira o código-fonte.

```
1  public class Mensalista extends Colaborador {
2      private double salarioMensal;
3
4      public Mensalista(String nome, String departamento, double salarioMensal) {
5          super(nome, departamento);
6          this.salarioMensal = salarioMensal;
7      }
8
9      public double salario() {
10         return this.salarioMensal;
11     }
12
13     @Override
14     public String toString() {
15         return String.format("%s\nSalário Mensal: R$ %.2f",
16                             super.toString(), this.salario());
17     }
18 }
```

A subclasse *Mensalista* é a mais simples depois da superclasse *Colaborador*. Ela acrescenta apenas o atributo *salarioMensal* (linha 2), que deve ser informado no momento da criação do objeto, utilizando o construtor declarado entre as linhas 4 e 7. Como o cálculo do salário é retornar o salário mensal, não há necessidade de um método *get*, ele foi substituído pelo método *salario* implementado entre as linhas 9 e 11. Por fim, o método *toString* é apresentado entre as linhas 13 e 17. A próxima classe implementada é a subclasse *Comissionado*, confira o código apresentado.

```
1  public class Comissionado extends Colaborador {
2      private double totalVendas;
3      private int comissao;
4
5      public Comissionado(String nome, String departamento, int
comissao) {
6          super(nome, departamento);
7          this.comissao = comissao;
8      }
9
10     public void setTotalVendas(double totalVendas) {
11         this.totalVendas = totalVendas;
12     }
13
14     public double getTotalVendas() {
15         return totalVendas;
16     }
17
18     public int getComissao() {
19         return comissao;
20     }
21
22     public double salario() {
23         return this.totalVendas * (this.comissao / 100.);
24     }
25
26     @Override
27     public String toString() {
28         return String.format("%s\nTotal de Vendas: R$ %.2f\n" +
29                             "Comissão: %d%\nSalário Mensal: R$ %.2f",
30                             super.toString(), this.getTotalVendas(),
31                             this.getComissao(), this.salario());
32     }
33 }
```

A subclasse *Comissionado* segue a mesma linha de implementação da subclasse *Horista*. Possui dois atributos, *totalVendas* e *comissao* (sobre as vendas), declarados de maneira privada (linhas 2 e 3). Possui um construtor com três parâmetros, em que dois são encaminhados para a superclasse, conforme observado entre as linhas 5 e 8. Da linha 10 até a

linha 20 estão declarados os métodos getters e setters da classe. O método `salario` é implementado entre as linhas 22 e 24, onde calcula o valor percentual sobre o total de venda. Um ponto interessante nesse cálculo, apresentado na linha 23, é a necessidade de utilizarmos um ponto após o número 100. O que acontece nesse caso é que o valor armazenado no atributo `comissao` e o valor 100 são do tipo inteiro. O retorno obtido nessa operação é inteiro também, e como é esperado um valor abaixo de 1, esse resultado inteiro seria igual a zero. Colocar o ponto após o número 100 faz com que o tipo do valor seja convertido para `double`, possibilitando agora o retorno de um número decimal preciso. Por fim, a implementação do método `toString` é feita entre as linhas 26 e 32.

A próxima subclasse implementada é a *BaseComissionado*. Essa classe tem relação indireta com *Colaborador* e direta com *Comissionado*. Confira o código dessa implementação a seguir:

```
1  public class BaseComissionado extends Comissionado {
2      private double salarioBase;
3
4      public BaseComissionado(String nome, String departamento, int
5          comissao, double salarioBase) {
6          super(nome, departamento, comissao);
7          this.salarioBase = salarioBase;
8      }
9
10     public double getSalarioBase() {
11         return salarioBase;
12     }
13
14     @Override
15     public double salario() {
16         return this.salarioBase + super.salario();
17     }
18
19     @Override
20     public String toString() {
21         return String.format("%s\nSalário Base: R$ %.2f",
22                             super.toString(), this.salarioBase);
23     }
}
```

Perceba na linha 1 que a extensão não é mais feita com base na classe *Colaborador*, mas agora é realizada com base na superclasse *Comissionado*. Sendo assim, é preciso definir apenas um novo atributo do salário-base na linha 2. O construtor necessita receber quatro parâmetros, os três referentes à superclasse *Comissionado*, que são

repassados por meio da palavra reservada *super* mais o valor do salário-base (linhas 4 a 7). E o método *getSalarioBase* é declarado entre as linhas 9 e 11.

Neste momento, temos a sobreescrita do método *salario* criado na superclasse (linhas de 12 a 16). Esse método precisa adicionar o valor da comissão ao salário-base. Para evitar realizar novamente os cálculos de comissão, utilizamos o método *salario* da superclasse invocada por meio da palavra *super* e o resultado é adicionado sobre o atributo da subclasse *BaseComissionado*. Com as palavras *this* e *super*, é possível determinar exatamente qual implementação você deseja executar em cada tempo do código implementado. Finalizando a subclasse *BaseComissionado*, o método *toString* é apresentado entre as linhas 18 e 22.

Com toda a família de classe implementada, basta fazer uma classe para testar as implementações realizadas até este momento. Confira a classe para teste das classes implementadas.

```
1  public class EmpresaMain {
2
3      public static void main(String... args) {
4          System.out.println("\n|---- Colaborador -----");
5          Colaborador colaborador = new Colaborador("Christian
Cunningham", "Infraestrutura");
6          System.out.println(colaborador);
7
8          System.out.println("\n|---- Horista -----");
9          Horista horista = new Horista("Avery Jordan",
"Desenvolvimento", 75.5);
10         horista.setHorasTrabalhadas(220);
11         System.out.println(horista);
12
13         System.out.println("\n|---- Mensalista -----");
14         Mensalista mensalista = new Mensalista("Taylor Griffin",
"Agile", 8500);
15         System.out.println(mensalista);
16
17         System.out.println("\n|---- Comissionado -----");
18         Comissionado comissionado = new Comissionado("Charlene
Butler", "Comercial", 25);
19         comissionado.setTotalVendas(350000);
20         System.out.println(comissionado);
21
22         System.out.println("\n|---- Base Comissionado -----");
23         BaseComissionado baseComissionado = new
BaseComissionado("Terrance Wallace", "Venda", 5, 3000);
24         baseComissionado.setTotalVendas(350000);
25         System.out.println(baseComissionado);
26     }
27 }
```

O objetivo da implementação realizada para teste foi criar as instâncias de cada uma das classes de maneira independente e popular a todos os atributos. Na sequência, é necessário realizar a exibição das informações por meio do método `toString`, que é invocado de maneira automática ao passar o nome do objeto para operações com valores do tipo `String`. Confira o resultado obtido com a execução do código da classe `EmpresaMain`.

```
|---- Colaborador -----
Colaborador: Christian Cunningham
Departamento: Infraestrutura

|---- Horista -----
Colaborador: Avery Jordan
Departamento: Desenvolvimento
Horas trabalhadas: 220
Valor por hora: R$ 75,50
Salário: R$ 16610,00

|---- Mensalista -----
Colaborador: Taylor Griffin
Departamento: Agile
Salário Mensal: R$ 8500,00

|---- Comissionado -----
Colaborador: Charlene Butler
Departamento: Comercial
Total de Vendas: R$ 350000,00
Comissão: 25%
Salário Mensal: R$ 87500,00

|---- Base Comissionado -----
Colaborador: Terrance Wallace
Departamento: Venda
Total de Vendas: R$ 350000,00
Comissão: 5%
Salário Mensal: R$ 20500,00
Salário Base: R$ 3000,00

Process finished with exit code 0
```

Tendo dado os primeiros passos sobre o conceito de herança, que é fundamental para a programação orientada a objetos, vamos entender um pouco mais sobre a família Java.

2.1 *Object*, a “super” superclasse do Java

Neste momento, é clara a importância do paradigma de programação orientada a objetos. Conseguimos reaproveitar os códigos já produzidos, deixando a construção de um sistema mais eficiente. Uma das maneiras de reaproveitar o código é por meio da extensão de um objeto, que conhecemos como herança. Mas quando não existe a herança, como no caso da classe *Colaborador* que fizemos, como o Java se comporta? Existe uma classe detentora de toda a força que habita em um sistema Java, a classe soberana chamada *Object*.

Quando não declaramos uma herança de maneira explícita, o Java automaticamente assume que será realizada uma herança da classe *Object* (HORSTMANN; CORNELL, 2010). Vamos ver como seria a maneira explícita disso com a classe *Colaborador*. Declaramos a classe *Colaborador* com a seguinte assinatura:

```
public class Colaborador { ... }
```

Porém, de maneira transparente ao desenvolvedor, o Java associa a declaração desta maneira:

```
public class Colaborador extends Object { ... }
```

É assim que o Java nos permite ter acesso a alguns métodos já predefinidos nos objetos que criamos, como no caso do método *toString* que utilizamos para converter nosso objeto em uma apresentação de fácil entendimento ao desenvolvedor/usuário do sistema. Conhecer os métodos já existentes na classe *Object* é fundamental para conseguir manter a estrutura proposta nos pacotes fundamentais do Java e extrair o melhor

desempenho de nossas aplicações (HORSTMANN; CORNELL, 2010). A seguir, discutiremos os métodos básicos da classe *Object* e como realizar a sobreescrita deles em nossas classes.



IMPORTANTE

Os principais métodos da classe *Object* são utilizados em implementações de *Threads* no sistema. A documentação do Java possui a explicação desses métodos e como podem ser utilizados. Para acessar a documentação do Java, procure por “Java docs Object” na internet.

2.2 Método *equals*

O método *equals* é utilizado na comparação entre dois objetos. Ele realiza o teste para validar se ambos são iguais. Quando efetuamos a comparação com o símbolo de dois iguais (==), da mesma forma que fazemos com variáveis do tipo primitivo como *int* ou *double*, a comparação é realizada pelo endereço de memória ao qual aquele objeto foi instanciado. Isso faz com que o resultado seja sempre negativo.

Para solucionar esse problema, é necessário realizar a implementação do método *equals* para aquela classe, e, assim, atender ao objetivo de comparação da igualdade dos objetos. Vamos utilizar a classe *Colaborador* como exemplo e reescrever o método *equals*. A implementação fica da seguinte maneira:

```
1  @Override
2  public boolean equals(Object outro) {
3      if (this == outro) return true;
4      if (outro == null) return false;
5      if (this.getClass() != outro.getClass()) return false;
6
7      Colaborador colaborador = (Colaborador) outro;
8      return nome.equals(colaborador.nome) &&
9             departamento.equals(colaborador.departamento);
10 }
```

Entre as linhas 3 e 5, são realizados testes rápidos para validar se são objetos idênticos, se o objeto recebido não é nulo e se possuem a mesma classe, respectivamente. Passando pelos testes rápidos, precisamos efetuar a coerção do objeto recebido para o tipo correto da classe, assim teremos acesso a todos os atributos da maneira correta (linha 7). Depois, basta comparar se todos os atributos de instância e os do objeto recebido possuem o mesmo valor, conforme a linha 8. Pronto, o método `equals` está sobrescrito e pronto para ser utilizado.

Podemos simplificar essa implementação para classes herdadas. Vamos fazer o exemplo com a classe *Mensalista*. Confira o código:

```
1     @Override
2     public boolean equals(Object outro) {
3         if(!super.equals(outro)) return false;
4         Mensalista mensalista = (Mensalista) outro;
5         return this.salarioMensal == mensalista.salarioMensal;
6     }
```

Ao verificar com o objeto `super`, é possível garantir que estamos falando com um objeto da mesma família (linha 3). Sendo da mesma família, basta fazer a coerção para o tipo correto (linha 4) e comparar os atributos adicionais da classe (linha 5).

2.3 Método `hashCode`

O método `hashCode` gera um valor *hash* que identifica o objeto. Ele funciona como um certificado digital para autenticar a instância de objeto. Esse método é muito utilizado em soluções que envolvam troca de informações entre sistemas. O valor *hash* é um número inteiro gerado com base nas informações da classe. Por exemplo, o algoritmo da classe *String* segue o algoritmo:

```
int hash = 0;
for (int i = 0; i < length(); i++)
    hash = 31 * hash + charAt(i);
```

Vem agora a pergunta de um milhão de dólares: como podemos implementar o método *hashCode* na classe *Colaborador*? Basta combinar valores inteiros com os *hashs* de cada atributo em sua classe, no caso, *Colaborador*. Vamos implementar o método com o seguinte código:

```
1  @Override
2  public int hashCode() {
3      return 7 * this.nome.hashCode()
4          + 11 * this.departamento.hashCode();
5  }
```

Se você já estiver preocupado com o trabalho que terá quando a classe tiver mais atributos, temos uma novidade. A partir da versão 7 do Java, a classe *Objects* (perceba o "s" no final), do pacote *java.util*, possui um método estático *hash*, que você envia para a lista de atributos que deseja utilizar para a construção do valor *hash* e ele retorna o valor pronto para uso. Simplificando a implementação acima, ficamos com o seguinte código:

```
1  @Override
2  public int hashCode() {
3      return Objects.hash(nome, departamento);
4  }
```

Pronto, agora você já sabe implementar o *hashCode* em suas classes.

2.4 Método *toString*

O método *toString*, como já sabemos, serve para representar um objeto em formato de um texto, facilitando a leitura das informações contidas nele. Vamos deixar aqui o exemplo da classe *Colaborador*, fechando a apresentação dos métodos básicos da classe *Object*, que precisamos implementar como boa prática em nossas classes.

```
1  @Override
2  public String toString() {
3      return String.format("Colaborador: %s\nDepartamento: %s",
4          this.nome, this.departamento);
5  }
```

Agora você está preparado para criar sua família de classes em quaisquer sistemas em que for atuar.



IMPORTANTE

É opcional o uso da anotação `@Override` em métodos sobrescritos. Contudo, sempre utilize essa anotação, pois é possível antecipar erros em tempo de compilação e não durante a execução do sistema.

3 Protegendo membros da classe

Quando você conheceu os modificadores de acesso no conceito de encapsulamento, foram apresentados dois tipos: o *public*, que permite que os membros de uma classe (sejam atributos ou métodos) estejam acessíveis diretamente para qualquer classe de um sistema; e o *private*, que faz com que permanecem acessíveis apenas para os membros da própria classe. Contudo, existe um nível intermediário entre os dois modificadores chamado *protected*. Com o modificador de acesso *protected*, todos os membros da classe se tornam diretamente acessíveis para a família e outras classes do mesmo pacote.

Os modificadores de acesso *public* e *protected* são mantidos quando se tornam membros da subclasse. Para esses membros (*public* e *protected*), não existe a necessidade de trabalhar com a palavra *super* precedendo o nome do método ou atributo. Utilize membros com modificadores de acesso *protected* apenas quando você tem total controle sobre o desenvolvimento do sistema. Caso contrário, você poderá expor o membro da classe de maneira inadequada e possivelmente violará o conceito de encapsulamento, que é importante para o desenvolvimento do sistema (DEITEL; DEITEL, 2010). Confira os atributos da classe *Colaborador* declarados como *protected*.

```
1     public class Colaborador {  
2         protected String nome;  
3         protected String departamento;  
4     } ...  
5  
6 }
```

Com essa alteração, todas as subclasses de *Colaborador* podem acessar diretamente os atributos *nome* e *departamento* sem a necessidade de serem intermediadas por um método com modificador de acesso do tipo *public*.

4 Polimorfismo

A ideia por trás do polimorfismo é que você possa programar o sistema com objetos que compartilham a mesma superclasse como se eles fossem o mesmo tipo de objeto. O foco dessa técnica é simplificar a implementação do sistema (DEITEL; DEITEL, 2010).

O uso de polimorfismo permite que sistemas sejam facilmente extensíveis, de modo que a inclusão de novas classes impacte pouquíssimos pontos da implementação do sistema em produção. Sendo assim, cada alteração do sistema deve ser feita exclusivamente nos locais que dizem respeito à nova classe. Esse benefício faz com que o ambiente, em tempo de execução, seja capaz de tratar cada especificidade. A chamada do método é determinada pelo tipo da instância, e não pelo tipo declarado. Veja o código a seguir:

```
Colaborador colaborador = new Mensalista("Brennan Sims", "Engenharia",  
15600);  
System.out.println(colaborador.getClass().getName());
```

Confira a saída obtida a partir do código apresentado.

Mensalista

Perceba que é possível declarar a variável como uma superclasse, mas instanciá-la como uma subclasse. E a saída, quando consultamos o tipo do objeto, é justamente a classe subclasse, que nesse caso é *Mensalista*. Apesar disso, os métodos da classe *Mensalista* não estão acessíveis, a não ser que seja feita uma coerção de *Colaborador* para *Mensalista*. Veja o exemplo a seguir, utilizando o método *salario* disponível apenas na subclasse.

```
System.out.println(colaborador.salario());
```

A saída apresenta o erro a seguir:

```
Error:(32, 39) java: cannot find symbol
  symbol:   method salario()
  location: variable colaborador of type Colaborador
```

Para realizar essa operação, fazemos a coerção, pois é necessário converter o objeto *colaborador* para o tipo de instância que possui a implementação do método. Caso contrário, não é possível utilizar os métodos especificados nas subclasses. O código deve ser implementado da seguinte maneira:

```
System.out.println(((Mensalista) colaborador).salario());
```

Você provavelmente está pensando qual seria então a utilidade do polimorfismo. De fato, é muito teórico e pouco prático. No entanto, imagine que você tem uma lista de colaboradores e gostaria de saber quantos mensalistas existem nela. Veja como fica o código implementado:

```
1  public class EmpresaMain {
2
3      public static long contadorMensalistas(List<Colaborador>
4          colaboradores) {
5          return colaboradores
6              .stream()
7              .filter(colaborador ->
8                  colaborador instanceof Mensalista)
9              .count();
10     }
```

(cont.)

```

11     public static void main(String... args) {
12         List<Colaborador> colaboradores = List.of(
13             new Horista("Avery Jordan", "Desenvolvimento", 75.5),
14             new Mensalista("Christian Cunningham",
15                 "Infraestrutura", 7000),
16             new Mensalista("Taylor Griffin", "Agile", 8500),
17             new Comissionado("Charlene Butler", "Comercial", 25),
18             new BaseComissionado("Terrance Wallace", "Venda", 5,
19                 3000),
20             new Mensalista("Brennan Sims", "Engenharia", 15600)
21         );
22         System.out.printf("O total de colaboradores mensalistas é de
23             %d colaboradores", contadorMensalistas(colaboradores));
24     }

```

Note que o método `contadorMensalistas` (linha 3) recebe como parâmetro uma lista de colaboradores. Percorremos a lista dos colaboradores filtrando todos cuja instância seja igual a `Mensalista`; para isso, utilizamos o operador `instanceOf` (linha 7). Após o filtro, efetuamos a contagem por meio do método `count` (linha 8), que retorna um valor inteiro do tipo `long` com a quantidade de elementos que obtiveram o valor de `verdadeira` na condição do filtro. O método `main` (linhas 11 a 22) apenas cria a lista de colaboradores e exibe a saída do método `contadorMensalistas`. Confira o resultado:

O total de colaboradores mensalistas é de 3 colaboradores

Com o polimorfismo, podemos declarar tipos mais gerais por meio da superclasse, mas executar os métodos instanciados pelas subclasses. Essa capacidade que a superclasse possui de receber as instâncias das subclasses facilita não só na hora de armazenarmos objetos com tipos diferentes, como também na comunicação entre métodos. Expandir sistemas com poucas alterações em declarações de métodos e reutilização de código é a maior contribuição da técnica de polimorfismo em orientação a objetos. O próximo passo é trabalhar com o cálculo do salário para todos os colaboradores.

5 Classes e métodos abstratos

Agora que conhecemos o polimorfismo, podemos criar um sistema para gerar a folha de pagamento e conseguir, a partir de uma lista de colaboradores, exibir o salário de cada um e o total da folha de pagamento. Porém, temos um pequeno problema. A classe *Colaborador* não possui o método *salario*, que efetua o cálculo necessário. Fácil! Vamos adicioná-lo na classe *Colaborador*.

```
1  public class Colaborador {  
2      protected String nome;  
3      protected String departamento;  
4  
5      public Colaborador(String nome, String departamento) {  
6          this.nome = nome;  
7          this.departamento = departamento;  
8      }  
9  
10     public double salario() {  
11         return 0.0;  
12     }  
13  
14     // Demais métodos...  
15 }
```

Legal, mas esse método não possui lógica nenhuma de aplicação. Se eu chamar um colaborador, ele precisa retornar um salário, e, se analisarmos bem, não existe a necessidade de criar uma instância de *Colaborador* pura em meu sistema. A lógica sempre dependerá da especificidade das subclasses do sistema. Para solucionar esse problema, existe um conceito chamado método abstrato. Nesse tipo de método é declarada apenas a sua assinatura sem a necessidade de nenhuma implementação. A partir de agora, qualquer subclass de *Colaborador* é forçada a implementar o método abstrato, assim garantimos a existência dele em qualquer classe que herde de *Colaborador*. Vamos conferir como fica a implementação da classe *Colaborador* utilizando o método abstrato para o cálculo do salário.

```
1  public class Colaborador {  
2      protected String nome;  
3      protected String departamento;  
4  
5      public Colaborador(String nome, String departamento) {
```

(cont.)

```
6         this.nome = nome;
7         this.departamento = departamento;
8     }
9
10    public abstract double salario();
11
12    // Demais métodos...
13 }
```

Porém, agora nossa classe ficou desfalcada. Não podemos permitir que seja criada uma instância de *Colaborador*, caso contrário teremos um pedaço a preencher. O que acontecerá com o programa se for chamado o método de *salario* direto de uma instância da classe *Colaborador*? Erro na certa! Mas, para evitar esse tipo de problema, o compilador já não permite o uso de um método abstrato em uma classe não abstrata. Portanto, vamos tornar a classe *Colaborador* abstrata.

```
1   public abstract class Colaborador {
2       protected String nome;
3       protected String departamento;
4
5       public Colaborador(String nome, String departamento) {
6           this.nome = nome;
7           this.departamento = departamento;
8       }
9
10      public abstract double salario();
11
12      // Demais métodos...
13  }
```

A partir desse momento, não é possível criar mais nenhuma instância da classe *Colaborador* no sistema. Isso significa que a linha de código a seguir não é mais permitida.

```
Colaborador c = new Colaborador("C3PO", "Tradução");
```

Caso tente compilar o código acima, o seguinte erro será apresentado:

```
Error:(31, 25) java: Colaborador is abstract; cannot be instantiated
```

Agora podemos criar a classe *folha de pagamento*, que receberá uma lista de colaboradores e imprimirá o relatório para o setor de recursos humanos. A classe *FolhaDePagamento* é implementada a seguir:

```
1  public class FolhaDePagamento {
2
3      private List<Colaborador> colaboradores;
4      private LocalDate hoje;
5
6      public FolhaDePagamento(List<Colaborador> colaboradores) {
7          this.colaboradores = colaboradores;
8          this.hoje = LocalDate.now();
9      }
10
11     public void geraRelatorio() {
12         System.out.println("|" + repeteCaracter('-', 80) + "|");
13         printTitulo();
14         System.out.println("|" + repeteCaracter('-', 80) + "|");
15         printColaboradores();
16         System.out.println("|" + repeteCaracter('-', 80) + "|");
17         printTotalFolha();
18         System.out.println("|" + repeteCaracter('-', 80) + "|");
19     }
20
21     private void printTitulo() {
22         String titulo = String.format("Folha de Pagamento - Mês: %s
Ano: %d",
23             hoje.getMonth().getDisplayName(TextStyle.FULL, new
Locale("pt")),
24             hoje.getYear());
25         System.out.printf("| %-79s|\n", titulo);
26     }
27
28     private void printColaboradores() {
29         System.out.printf("| %-39s| %-37s |\n", "Nome", "Salário");
30         System.out.println("|" + repeteCaracter('-', 80) + "|");
31         colaboradores.forEach(
32             colaborador ->
33                 System.out.printf("| %-39s|%-38s |%n",
34                     colaborador.getName(),
35                     String.format("R$ %.2f",
36                     colaborador.salario())));
37         );
38     }
39
40     private void printTotalFolha() {
41         double total = colaboradores
42             .stream()
43             .mapToDouble(Colaborador::salario)
44             .sum();
45         System.out.printf("| %-40s%-38s |%n",
46             "Total de pagamentos",
47             String.format("R$ %.2f", total));
48     }
49
50     private String repeteCaracter(char c, int tamanho) {
51         char[] caracteres = new char[tamanho];
52         Arrays.fill(caracteres, c);
53         return new String(caracteres);
54     }
55 }
```

Repare que o único objeto que foi necessário para efetuar todo o cálculo da folha de pagamento foi declarado como a superclasse. Caso uma nova subclasse seja criada no sistema, a classe *FolhaDePagamento* não sofrerá nenhuma alteração. Veja como conhecer cada técnica de programação orientada a objetos é importante para evitar retrabalhos desnecessários durante a evolução do sistema. Mas você deve estar se perguntando como isso tudo funciona, então vamos implementar a chamada da folha de pagamento na classe *EmpresaMain*. Confira o código:

```
1  public class EmpresaMain {
2
3      public static void main(String... args) {
4          Horista horista = new Horista("Avery Jordan",
5              "Desenvolvimento", 75.5);
6          horista.setHorasTrabalhadas(220);
7
8          Comissionado comissionado = new Comissionado("Charlene
9              Butler", "Comercial", 25);
10         comissionado.setTotalVendas(350000);
11
12         BaseComissionado baseComissionado = new
13             BaseComissionado("Terrance Wallace", "Venda", 5, 3000);
14         baseComissionado.setTotalVendas(150000);
15
16         List<Colaborador> colaboradores = List.of(
17             horista,
18             new Mensalista("Christian Cunningham",
19                 "Infraestrutura", 7000),
20             new Mensalista("Taylor Griffin", "Agile", 8500),
21             comissionado,
22             baseComissionado,
23             new Mensalista("Brennan Sims", "Engenharia", 15600)
24         );
25     }
26 }
```

E podemos conferir a saída na sequência.

Folha de Pagamento - Mês: julho Ano: 2020	
Nome	
Avery Jordan	R\$ 16610,00
Christian Cunningham	R\$ 7000,00
Taylor Griffin	R\$ 8500,00
Charlene Butler	R\$ 87500,00
Terrance Wallace	R\$ 10500,00
Brennan Sims	R\$ 15600,00
Total de pagamentos	R\$ 145710,00

Agora que você já sabe tudo sobre herança entre classes, polimorfismo e classes e métodos abstratos, é só colocar tudo em prática com alguns exercícios.

6 Exercícios de fixação

Os exercícios a seguir lhe guiarão para colocar em prática tudo o que foi aprendido neste capítulo.

1. Construa um sistema de geometria que possua uma família de formas. As formas podem ser de duas dimensões (quadrado, triângulo, círculo, etc.) ou três dimensões (esfera, cubo, cone, etc.). Deverá existir apenas uma família de classes que conte coletores de todas as formas descritas.
2. Defina as classes abstratas que terão um método para cálculo de área (2D) ou volume (3D). Implemente nas respectivas subclasses os cálculos correspondentes.
3. Utilizando o polimorfismo, crie uma lista de formas e efetue e apresente a saída do cálculo da área ou volume e o nome da forma que foi processada.

Considerações finais

Neste capítulo, entendemos como é importante o conceito de herança para construirmos sistemas cada vez mais extensíveis e que permitem reutilizar o código já existente para aumentar sua capacidade de processamento.

Identificamos outro modificador de acesso chamado *protected*, que está posicionado entre o *public* e o *private*. Vimos que ele é recomendado para utilizarmos dentro de uma família de classes partindo da superclasse. Contudo, se a equipe ou o desenvolvedor não estiverem seguros

do acesso e da manipulação de um atributo *protected*, devemos deixá-lo como privado mesmo.

Foi apresentado o conceito de polimorfismo, em que a superclasse pode abrigar a instância de qualquer uma de suas subclasses. Isso facilita o desenvolvimento, sempre trabalhando de forma geral e deixando os itens específicos para serem implementados por cada uma das subclasses.

Por fim, aprendemos como garantir que podemos chamar um método a partir de uma superclasse e forçar a implementação apenas nas subclasses. Para isso, utilizamos o conceito de classes e métodos abstratos, os quais tornam mais concisa a implementação de um sistema.

Referências

DEITEL, Paul; DEITEL, Harvey. **Java**: como programar. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

GAMMA, Erich et al. **Padrões de projetos**: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2009.

HORSTMANN, Cay S.; CORNELL, Gary. **Core Java, volume 1**: fundamentos. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

Capítulo 6

Interface

Provavelmente, quando você escuta ou lê a palavra “interface”, a primeira coisa que lhe vem à cabeça é a tela do aplicativo que utiliza no celular ou um site que acessa todos os dias. E você está certo. Mas esse é um conceito que chamamos de interface do usuário. O conceito de interface é mais amplo que apenas esse e é aplicado em qualquer sistema com interface gráfica.

Criar uma interface significa definir e padronizar interações entre coisas, pessoas e sistemas, independentemente da combinação entre eles (DEITEL; DEITEL, 2010). Por exemplo, se pensarmos em um volante de um veículo automotor, podemos afirmar que o volante é uma interface entre o motorista e a suspensão do veículo que auxiliará a tarefa de determinar a direção.

Em um veículo, existem diversas outras interfaces que podem ser utilizadas pelo motorista, como ajuste do banco, alavanca de abertura do capô, pedais de freio, acelerador e muitas outras. Perceba que a interface tem a capacidade de especificar quais são as operações realizadas por um sistema, contudo elas não especificam como eles fazem isso (DEITEL; DEITEL, 2010; GOODRICH; TAMASSIA, 2013).

Assim como um veículo, um sistema computacional formado por objetos também precisa de interfaces para facilitar a comunicação entre eles. Uma interface não é exatamente uma classe do sistema, é um conjunto de requisitos a que uma classe, ao implementá-la, deve se adequar para atendê-los (HORSTMANN; CORNELL, 2010).

Uma classe pode implementar quantas interfaces forem necessárias para atingir o objetivo do sistema. Cada interface pode conter diversos métodos sem a necessidade de uma quantidade mínima ou máxima para isso. Além dos métodos, uma interface pode determinar também valores constantes para uso em uma aplicação. É importante deixar claro que variáveis de instância e especificação de implementação dos códigos de cada método não são permitidas na construção de interfaces (HORSTMANN; CORNELL, 2010; GOODRICH; TAMASSIA, 2013).

1 Uso da interface em classes

Em Java, a declaração de interface inicia-se com a palavra *interface*, que substitui a palavra *class*. Observe a seguir:

```
1     interface NomeInterface {  
2 }
```

Todos os métodos declarados dentro de uma interface são automaticamente considerados públicos e abstratos, sem a necessidade de declarar isso de maneira explícita para o compilador. As constantes definidas também são consideradas públicas (DEITEL; DEITEL, 2010).

No entanto, ao implementar uma interface em uma classe, deve-se explicitar o modificador de acesso *public*. Caso contrário, o compilador reclamará que você está alterando para um modificador de acesso mais fraco ao método (HORSTMANN; CORNELL, 2010).



NA PRÁTICA

De acordo com a especificação da linguagem Java, não devemos explicitar informações redundantes como as palavras reservadas *public* e *abstract* para os métodos de uma interface. Como elas são consideradas por padrão na linguagem, podem ser suprimidas. Apesar disso, alguns programadores preferem deixá-las explícitas para facilitar a leitura do código pelos demais membros da equipe.

Voltaremos agora ao exemplo da folha de pagamento para entender o uso das interfaces no sistema. Antes de começar a trabalhar com as interfaces, vamos inserir uma nova classe em nossa hierarquia chamada *Gerente*. Ele terá como atributos uma lista de colaboradores subordinados a ele e um salário-base. Um *Gerente* receberá um percentual sobre as vendas realizadas pelas classes *Comissionado* e *BaseComissionado*. O código inicial da classe *Gerente* é apresentado a seguir:

```
1  public class Gerente extends Colaborador{  
2  
3      private double salarioBase;  
4      private List<Colaborador> colaboradores;  
5  
6      public Gerente(String nome, String departamento,  
7                      double salarioBase, List<Colaborador>  
colaboradores) {  
8          super(nome, departamento);  
9          this.salarioBase = salarioBase;  
10         this.colaboradores = colaboradores;  
11     }  
12  
13     public double getSalarioBase() {  
14         return salarioBase;  
15     }  
16  
17     public List<Colaborador> getColaboradores() {  
18         return colaboradores;  
19     }  
20  
21     private double getBonus() {
```

(cont.)

```

22         List<Colaborador> colaboradores = this.colaboradores
23             .stream()
24             .filter(colaborador -> colaborador instanceof
Comissionado)
25                 .collect(Collectors.toList());
26
27         double total = colaboradores
28             .stream()
29             .mapToDouble(Colaborador::salario)
30             .sum();
31
32         return total * 0.1;
33     }
34
35     @Override
36     public double salario() {
37         return salarioBase + getBonus();
38     }
39
40     @Override
41     public String toString() {
42         return String.format("%s\nSalário: R$ %.2f",
43             super.toString(), this.salario());
44     }
45
46 }

```

O gerente deve ser responsável por gerar sua folha de pagamento. Contudo, é necessário garantir que esse processo seja autenticado, pois gera-se uma despesa para a empresa. Agora, você pode ir direto e criar um método em *gerente* e um campo que represente a senha e, *voilà*, temos tudo sob controle novamente. Porém, nossos colaboradores comissionados também precisam se autenticar no sistema para liberar as entregas dos projetos.

Repare que existe um requisito se espalhando em muitas classes, e precisamos garantir que todas as classes que devem se autenticar no sistema possuam o método *estahAutenticado*. Com o Java, fazemos isso por meio de uma interface. Em vez de termos implementações de métodos apenas por meio do uso da interface, podemos usar os benefícios de uma linguagem fortemente tipada.¹ Com a interface, o compilador consegue verificar se o método existe ou não em uma classe implementada (HORSTMANN; CORNELL, 2010).

¹ Linguagem fortemente tipada é uma linguagem que define os tipos das variáveis e objetos.

Vamos implementar a interface Autenticavel. Verifique o código:

```
1  public interface Autenticavel {  
2      boolean estahAutenticado(String senhaDigitada);  
3  }  
4  
5 }
```

Veja que a linha 3 possui apenas a assinatura do método que deve ser implementado em quem firmar o contrato com essa interface. Confira a seguir a implementação da interface Autenticavel na classe Gerente:

```
1  public class Gerente  
2      extends Colaborador  
3      implements Autenticavel{  
4  
5      private double salarioBase;  
6      private List<Colaborador> colaboradores;  
7      private String senha;  
8  
9      public Gerente(String nome, String departamento,  
10                      double salarioBase, List<Colaborador>  
colaboradores) {  
11          super(nome, departamento);  
12          this.salarioBase = salarioBase;  
13          this.colaboradores = colaboradores;  
14      }  
15  
16      public String getSenha() {  
17          return senha;  
18      }  
19  
20      public void setSenha(String senha) {  
21          this.senha = senha;  
22      }  
23  
24      // Código ocultado...  
25  
26      @Override  
27      public boolean estahAutenticado(String senhaDigitada) {  
28          return senha.equals(senhaDigitada);  
29      }  
30  }
```

Na classe Gerente, acrescentou-se a palavra *implements* (linha 3) informando a interface que será implementada na classe. Depois, declarou-se um atributo de instância chamado *senha* (linha 7) e seus respectivos métodos *getters* e *setters* (linhas 16 a 22). Por fim, o método *estahAutenticado* foi implementado para validação das senhas (linhas 26 a 30). Como será o gerente quem vai chamar o relatório de folha de pagamento e ele já possui a sua própria lista de colaboradores, vamos

modificar a classe *FolhaDePagamento* para simplificar os cálculos e o relatório da folha de pagamento. Observe o código a seguir:

```
1  public class FolhaDePagamento {  
2  
3      private List<Colaborador> colaboradores;  
4      private LocalDate hoje;  
5  
6      public FolhaDePagamento(Gerente gerente) {  
7          this.colaboradores = new  
ArrayList<Colaborador>(gerente.getColaboradores());  
8          this.colaboradores.add(gerente);  
9          this.hoje = LocalDate.now();  
10     }  
11  
12     // Código ocultado.  
13 }
```

Pronto, com essa pequena alteração no construtor (linhas 6 a 10), ele passa a receber o gerente como parâmetro. Todos os colaboradores de gerente são atribuídos aos colaboradores da folha de pagamento (linha 7). Na sequência, adicionamos o gerente à lista (linha 8). Os demais métodos continuam os mesmos. Vamos observar como fica a classe *EmpresaMain* para testarmos nossa implementação:

```
1  public class EmpresaMain {  
2  
3      public static void main(String... args) {  
4          // Código omitido.  
5          Gerente gerente = new Gerente("Juliana", "Tecnologia", 20000,  
colaboradores);  
6          gerente.setSenha("1234");  
7  
8          try(Scanner scanner = new Scanner(System.in)) {  
9              System.out.print("Informe sua senha:\n> ");  
10             String senha = scanner.next();  
11  
12             if(gerente.estahAutenticado(senha)) {  
13                 FolhaDePagamento folha = new  
FolhaDePagamento(gerente);  
14                 folha.geraRelatorio();  
15             } else {  
16                 System.err.println("Falha ao autenticar colaborador");  
17             }  
18         }  
19     }  
20 }  
21 }
```

Vamos conferir agora o resultado obtido a partir do código apresentado acima:

Informe sua senha:

> 1234

Folha de Pagamento - Mês: julho Ano: 2020		
Nome	Salário	

Antônio		R\$ 16610,00
Helena		R\$ 7000,00
Gloria Maria		R\$ 8500,00
Ana		R\$ 87500,00
Francisco		R\$ 10500,00
João		R\$ 15600,00
Juliana		R\$ 29800,00

Total de pagamentos		R\$ 175510,00

Foi necessário informar uma senha para que a folha de pagamento fosse gerada. Alteramos apenas o construtor da classe e todos os demais métodos já estavam prontos para realizar o cálculo, incluindo o próprio gerente na folha de pagamento de seu departamento. Claro que em um fluxo com interface gráfica apropriada poderíamos melhorar a modelagem do sistema feita até o momento. A questão principal aqui é que você perceba o quanto é importante utilizar os recursos de uma

programação orientada a objetos e que escalar seu código e fazer manutenções fica mais fácil do que algo aplicado sem estrutura e sem uma análise do problema.

Para continuarmos nosso exemplo, imagine que nosso colaborador comissionado precisa realizar a assinatura digital de documentos, e uma assinatura digital precisa sempre ser autenticada, pois foi a regra de negócio colocada pelo cliente. Se estiver pensando: “é só inserir mais um método na interface *Autenticavel* e problema resolvido”. Você tem razão, mas vamos analisar esse cenário em detalhe.

Se apenas inserirmos o método na interface *Autenticavel*, o gerente também terá que implementar o método da assinatura digital, e pela regra de negócio, um gerente não pode ter essa assinatura. Se você pensou que uma classe pode implementar quantas interfaces são necessárias, você acertou. Então cria-se mais uma interface e implementam-se as duas no colaborador comissionado. Ótima solução! Vamos ver como fica a assinatura da classe *Comissionado*:

```
1     public class Comissionado  
2       extends Colaborador  
3         implements AssinaturaDigital, Autenticavel { ... }
```

Problema resolvido! Podemos implementar diversas interfaces em uma classe, basta separarmos o nome delas por vírgula. Contudo, imagine um cenário em que a *AssinaturaDigital* será implementada em outra classe. Já sabemos que é necessário que a *AssinaturaDigital* e a *Autenticavel* andem juntas. Se o desenvolvedor esquecer desse pequeno detalhe, não dará certo.

No entanto, existe um recurso com o qual conseguimos garantir esse vínculo: podemos fazer uma herança também em interfaces.

Com a herança, podemos estender os requisitos solicitados a uma classe por meio da interface. Assim, conseguimos vincular as duas

interfaces sem interferir nas demais classes já implementadas com a interface Autenticavel. Vamos conferir como fica o código da interface AssinaturaDigital:

```
1  public interface AssinaturaDigital extends Autenticavel {
2      long hash();
3  }
```

Utilizamos a palavra reservada `extends` (linha 1) para efetuar a herança entre as duas interfaces, assim como nas classes. Vamos agora vincular a classe `Comissionado` e verificar a nova implementação.

```
1  public class Comissionado
2      extends Colaborador
3      implements AssinaturaDigital {
4
5      private double totalVendas;
6      private int comissao;
7      private int senha;
8
9      public Comissionado(String nome, String departamento, int
comissao) {
10         super(nome, departamento);
11         this.comissao = comissao;
12     }
13
14     public void setTotalVendas(double totalVendas) {
15         this.totalVendas = totalVendas;
16     }
17
18     public double getTotalVendas() {
19         return totalVendas;
20     }
21
22     public int getComissao() {
23         return comissao;
24     }
25
26     public double salario() {
27         return this.totalVendas * (this.comissao / 100.);
28     }
29
30     public int getSenha() {
31         return senha;
32     }
33
34     public void setSenha(int senha) {
35         this.senha = senha;
36     }
37
38     @Override
39     public String toString() {
40         return String.format("%s\nTotal de Vendas: R$ %.2f\n" +
41                             "Comissão: %d%\nSalário Mensal: R$ %.2f",
42                             nome, salario(), comissao, senha);
```

(cont.)

```

42             super.toString(), this.getTotalVendas(),
43             this.getComissao(), this.salario());
44         }
45     }
46     @Override
47     public long hash() {
48         long hash = (senha % 3) * super.hashCode();
49         return hash;
50     }
51 }
52 @Override
53 public boolean estahAutenticado(String senhaDigitada) {
54     return senha == Integer.parseInt(senhaDigitada);
55 }
56 }
```

Na linha 3, efetuamos o vínculo entre a classe *Comissionado* e a interface *AssinaturaDigital*. Precisamos inserir um novo atributo para armazenar a senha (linha 7), com seus métodos getters e setters (linhas 30 a 36). Entre as linhas 46 e 55, implementamos os métodos solicitados pela interface *AssinaturaDigital*, que, inclusive, obriga a implementação do método *estahAutenticado* por causa da herança entre as interfaces. É importante ressaltar que a classe *BaseComissionado*, como subclasse de *Comissionado*, também recebe os métodos implementados a partir da interface. Nesse caso, se necessário, deve-se sobrescrever o método para atender à classe específica.

2 Interfaces como tipo

O uso de interfaces pode ir um pouco além dos recursos em forma de contrato, como visto no capítulo anterior. Sabemos que seu uso como instância direta não é possível por dispor apenas de métodos abstratos. Porém, a interface pode ser utilizada como tipo de declaração de um objeto.

Para ilustrar melhor esse uso, vamos imaginar que gostaríamos de criar um filtro para uma lista de números inteiros. Esse filtro pode executar diversos tipos de testes para obter o resultado desejado. Se pensarmos pelo método tradicional, teremos uma classe com diversos métodos de filtro. Agora imagine que toda vez que precisar de um novo

filtro, você tenha que codificar a classe *filtro* e testar tudo que já estava funcionando. “Impraticável” é a palavra que define essa implementação.

E se você criasse uma interface que seria um predicado da operação e ela tivesse um método que faz justamente esse teste necessário para o filtro? Vamos começar criando a interface *Predicado*, confira a implementação:

```
1  public interface Predicado {  
2      Boolean teste(Integer valor);  
3  }
```

Elá define a assinatura do método *teste* (linha 2), que recebe um valor do tipo inteiro e retorna um booleano como resultado. Na sequência, podemos implementar a classe *Filtro*, que usará *Predicado* como um tipo, e não como um contrato de recurso. Confira o código:

```
1  public class Filtro {  
2  
3      public List<Integer> filtra(List<Integer> numeros, Predicado  
predicado) {  
4          return numeros  
5              .stream()  
6              .filter(numero -> predicado.teste(numero))  
7              .collect(Collectors.toList());  
8  
9      }  
10 }
```

Note que o método *filtra* recebe uma lista de números inteiros e o predicado como parâmetros (linha 3). O predicado é utilizado dentro do método *filter* da biblioteca de *stream* do Java 8 para efetuar o teste necessário (linha 6). Assim, conseguimos utilizar esse mesmo método sem a necessidade de mexer na classe *Filtro*. Ela ficou com um propósito mais geral. A classe *Filtro* pode atender a mais propósitos, mas isso fica para outro capítulo.

Neste momento, vamos focar como o *Predicado* serve para filtrar a lista a partir de qualquer operação desejada e possível para números inteiros. Vamos pensar, a princípio, em uma operação básica de

números: identificar os números pares e ímpares. Vamos começar implementando a classe *Par*.

```
1  public class Par implements Predicado {
2      @Override
3      public Boolean teste(Integer valor) {
4          return 0 == valor % 2;
5      }
6  }
```

A implementação é simples, retorna o valor do resto da divisão por dois, comparado com o número 0 (zero) (linha 4). Agora a classe *Impar*.

```
1  public class Impar implements Predicado {
2      @Override
3      public Boolean teste(Integer valor) {
4          return 1 == valor % 2;
5      }
6  }
```

Agora temos também a classe *Impar*. Ambas implementam a interface *Predicado* e, por consequência, o método *teste*, que é utilizado na classe *Filtro*. Agora, confira a classe *TesteFiltro*, que usará essas classes para filtrar uma lista de números inteiros, separando-os em pares e ímpares. O código da classe *TesteFiltro* é:

```
1  public class TesteFiltro {
2      public static void main(String[] args) {
3          List<Integer> numeros = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9,
10);
4          Filtro filtro = new Filtro();
5          List<Integer> pares = filtro.filtra(numeros, new Par());
6          List<Integer> impares = filtro.filtra(numeros, new Impar());
7
8          System.out.printf("Números pares:%22s\n", pares);
9          System.out.printf("Números ímpares:%20s\n", impares);
10     }
11 }
```

Na linha 3, temos a declaração de uma lista com números de 1 a 10 como base para a operação de filtro. Na sequência, declaramos o objeto *filtro* (linha 4). Na linha 5, utilizamos o objeto *filtro* para invocar o método *filtra*, para o qual passamos a lista de números e uma nova instância do objeto *Par*. Como ele implementa o *Predicado*, podemos passá-lo como

referência para o objeto do tipo *Predicado*. É uma operação que lembra o polimorfismo. O mesmo acontece na linha 6 para os números ímpares. Nas linhas 8 e 9, exibimos o resultado no console. Vamos conferir se funcionou, verificando a saída obtida no console.

Números pares: [2, 4, 6, 8, 10]

Números ímpares: [1, 3, 5, 7, 9]

Funcionou perfeitamente. Agora podemos criar vários outros filtros sem a necessidade de mexer em diversas classes do código. Agora que vimos várias das vantagens do uso de interface, chegou a hora de praticar.

3 Exercícios de fixação

Os exercícios a seguir lhe guiarão para colocar em prática tudo o que foi aprendido ao longo do capítulo.

1. Crie uma classe que implemente um predicado para números primos e teste esse novo filtro.
2. Crie uma classe que implemente um predicado para números múltiplos de 3 e teste esse filtro.
3. Crie uma interface para efetuar a translação de objetos 2D.
4. Crie uma interface para efetuar a translação de objetos 3D.
5. Crie uma interface para efetuar a escala de objetos 2D.
6. Crie uma interface para efetuar a escala de objetos 3D.
7. Crie uma interface para efetuar a rotação de objetos 2D.
8. Crie uma interface para efetuar a rotação de objetos 3D.
9. Teste no conjunto de classes de Forma.

Considerações finais

Neste capítulo, conseguimos validar o uso de interface como um meio de fazer um contrato entre objetos de um sistema. Ela cria a obrigatoriedade de implementar todos os métodos determinados em sua estrutura. O contrato efetuado faz com que o objeto se comprometa em implementar os métodos declarados dentro da interface.

Uma interface só possui métodos *abstratos* e *públicos*, portanto não pode ser instanciada diretamente. Também não existe a necessidade de informar a cada método as palavras reservadas *abstract* e *public*, como uma boa prática do Java, já que é uma informação redundante.

Interfaces também podem ser criadas em uma hierarquia, trabalhando-se com herança para vincular a obrigatoriedade entre suas implementações sem afetar as demais. Agora que você aprendeu mais sobre interfaces, é hora de usá-las no mundo real para que seus projetos sejam cada vez mais escaláveis.

Referências

DEITEL, Paul; DEITEL, Harvey. **Java: como programar**. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados & algoritmos em Java**. Porto Alegre: Bookman, 2013.

HORSTMANN, Cay S.; CORNELL, Gary. **Core Java, volume 1**: fundamentos. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

Capítulo 7

Constantes, atributo estático e método estático

Quando estamos criando um sistema, muitas vezes precisamos compartilhar a mesma informação em diversos trechos de nosso código. Por exemplo, em um sistema bancário, é preciso compartilhar a informação sobre o número dos bancos em vários momentos. Esse número raramente muda e é utilizado em diversos pontos do código. Imagina ter que lembrar todos os locais que uma informação constante é utilizada? Para melhorar esse cenário, em programação, temos um recurso chamado *constante*, que pode ser aplicado nos níveis variável, método ou classe em Java (DEITEL; DEITEL, 2010; HORSTMANN; CORNELL, 2010).

Além disso, também pode existir uma situação em que queremos compartilhar informações entre as variadas instâncias de nossa classe, como o número da conta corrente. Sendo esse número sequencial e atribuído de maneira automática pelo sistema, precisamos garantir que uma classe compartilhe a informação do último número de conta gerado. Esse compartilhamento permitirá que a informação gerada não seja repetida em nenhuma das instâncias da classe.

Podemos necessitar compartilhar métodos para cálculos frequentes, e disponibilizar acesso a esses métodos por meio da criação de um objeto pode gerar um consumo de memória maior do que o necessário ao sistema. Um bom exemplo é o uso dos métodos de cálculos matemáticos, como os da classe *Math* pertencente ao pacote *java.lang*. Ambos os cenários, do compartilhamento da informação e dos cálculos matemáticos, são facilmente contornados por meio do uso de métodos estáticos (DEITEL; DEITEL, 2010; HORSTMANN; CORNELL, 2010).

No decorrer deste capítulo, discutiremos esses assuntos e demonstraremos como o Java implementa as constantes e os atributos e os métodos estáticos. Vamos começar falando sobre as constantes e suas possibilidades.

1 Constantes

Vamos construir a classe *Banco*. Ela deve ter as seguintes características: o número do banco, que não pode ser alterado durante a execução do programa; o nome do banco; e o nome de seu presidente. A declaração inicial da classe, pelo que conhecemos até o momento, é a seguinte.

```
1  public class Banco {  
2      private Long numero;  
3      private String nome;  
4      private String nomePresidente;  
5  
6      // Getters e Setters  
7  }
```

Para deixar o número do banco fixo, precisamos trabalhar com um recurso muito importante que é a constante. Pelo próprio nome, podemos deduzir que é um valor que não será alterado ao longo do tempo ou do ciclo de vida daquele objeto. Uma constante é definida no Java por meio da palavra reservada *final*. Ela deve ser aplicada na declaração do objeto entre o modificador de acesso e seu tipo. Uma boa prática adotada pelo Java é que toda constante deve ser declarada com seu nome em caixa-alta (todas as letras maiúsculas) e, em caso de nome composto, separada por *underline* (_). Veja o resultado da aplicação na classe Banco.

```
1  public class Banco {  
2      private final Long NUMERO = 1L;  
3      private String nome;  
4      private String nomePresidente;  
5  
6      // Getters e Setters  
7  }
```

Observe a linha 2, onde colocamos a palavra *final* e aumentamos a caixa da palavra *número*. Mas um ponto novo é a atribuição de um valor diretamente em sua declaração. Isso ocorre pois precisamos estabelecer o valor dessa constante logo em sua declaração, já que não podemos alterá-lo em outro momento do sistema.

É importante ressaltar que a alteração do valor de uma constante não é mais possível a partir da criação do objeto na memória. Mas podemos deixar a inicialização mais flexível a partir do construtor do objeto, confira a seguir.

```
1  public class Banco {  
2      private final Long NUMERO;  
3      private String nome;  
4      private String nomePresidente;  
5  
6      public Banco(Long numero, String nome, String nomePresidente) {  
7          this.NUMERO = numero;  
8          this.nome = nome;  
9          this.nomePresidente = nomePresidente;  
10     }  
11  
12     // Getters e Setters  
13  }
```

Vamos fazer uma classe para testar a classe *Banco*.

```
1  public class TesteBanco {  
2      public static void main(String[] args) {  
3          Banco banco = new Banco(1L, "Brasil", "Maria José");  
4          banco.setNUMERO(2L);  
5      }  
6  }
```

Qual será o resultado ao executar o código? Vamos conferir.

```
Banco.java:17:13  
java: cannot assign a value to final variable NUMERO
```

O Java não permite compilar a classe *Banco* com a declaração de um método *set* ao atributo *final*. Isso significa que se o objeto for criado com determinado valor nesse atributo, para modificá-lo, deve-se excluir o objeto antigo e criar um novo. Atributos declarados como *final* são imutáveis.

A criação de um recurso constante não se limita à criação de atributos, é possível criar métodos e classes como constantes. Mas você deve estar se perguntando: qual seria a necessidade de um método *final* e uma classe *final*? Vamos considerar o cenário hipotético de uma herança de uma classe *Conta* (superclasse) para *Poupança* (subclasse). Por um motivo de negócios, quando o método *consultaSaldo* for invocado, ele deverá apenas retornar o valor do atributo *saldo*.

Para garantir isso em toda a hierarquia de classes do tipo *Conta*, definimos o método *consultaSaldo* como *final*. Um método *final* não pode ser sobreescrito por suas subclasses, sejam elas diretas ou indiretas (DEITEL; DEITEL, 2010). Vejamos como funciona isso na prática. Vamos declarar a classe *Conta* nesse início.

```
1  public class Conta {
2      private String cliente;
3      private Long numero;
4      private Double saldo;
5
6      // Construtor
7      // Getters e Setters
8
9      public final Double consultaSaldo() {
10         return saldo;
11     }
12 }
```

Na sequência, vamos declarar a classe *Poupança* como uma herança de *Conta* e tentaremos sobreescrivar o método *consultaSaldo*.

```
1  public class Poupanca extends Conta {
2      private Double taxaRendimento;
3
4      // Construtor
5      // Getters e Setters
6
7      @Override
8      public Double consultaSaldo() {
9          return taxaRendimento * getSaldo();
10     }
11 }
```

O resultado que temos, ao tentar compilar nosso projeto, é um erro. Veja a saída do console para esse cenário.

```
Poupanca.java:16:19
java: consultaSaldo() in financeiro.Poupanca cannot
override consultaSaldo() in financeiro.Conta overridden
method is final
```

Em tempo de compilação, o Java informa que não é possível realizar a sobreescrita do método *consultaSaldo* por ele ser final. Então sempre que um método tiver uma lógica que não pode ser modificada por nenhuma outra subclasse, precisamos declará-lo como final.

E em uma classe final? O que será que acontece com ela? Bem, uma classe final não pode se tornar uma superclasse. Não existe herança entre uma classe final e outra classe qualquer. A instância de um objeto criado a partir de uma classe final é única e não pode sofrer mutações

ao longo do processo do sistema. Um bom exemplo de classe final é a classe *String*.

O uso de classe final é importante por uma questão de segurança no seu código. É uma estratégia para que desenvolvedores não consigam driblar restrições em seu código por meio da herança entre classes (DEITEL; DEITEL, 2010). Não se esqueça de declarar explicitamente o que for final em seu projeto, pois, com as indicações corretas em um código, o compilador é capaz de realizar otimizações importantes no processo de gerar o bytecode.

Vamos transformar a classe *Conta* em final e entender o que acontece com a classe *Poupança* e o código ao compilá-los. A declaração da classe *Conta* deve ser modificada para a seguinte maneira:

```
public final class Conta { // Código omitido... }
```

Agora, ao tentar compilar o projeto, veja o erro apresentado.

```
Poupanca.java:3:31  
java: cannot inherit from final financeiro.Conta
```

O compilador já acusa que não é possível realizar herança da classe *Conta*, por ela ser uma classe final.



PARA SABER MAIS

Para compreender mais a fundo o uso e as aplicações sobre métodos e classes do tipo final, procure por “final java doc” na internet para acessar a documentação.

2 Atributos e métodos estáticos

Imagine que a partir deste momento queremos que o número da conta seja gerado automaticamente. Só poderemos consultá-lo, e não mais informá-lo ao objeto. Uma pequena alteração na classe `Conta` é necessária. Confira a seguir:

```
1  public class Conta {
2      private String cliente;
3      private Long numero = 0L;
4      private Double saldo;
5
6      public Conta(String cliente, Double saldo) {
7          this.cliente = cliente;
8          this.numero++;
9          this.saldo = saldo;
10     }
11     //...
12 }
```

Deixamos um valor inicial de 0 no número da conta para que seja possível efetuar o incremento a cada objeto novo construído. A princípio, isso é o suficiente para criar números sequenciais em nosso banco. Criemos a classe para testar nosso gerador de número sequencial.

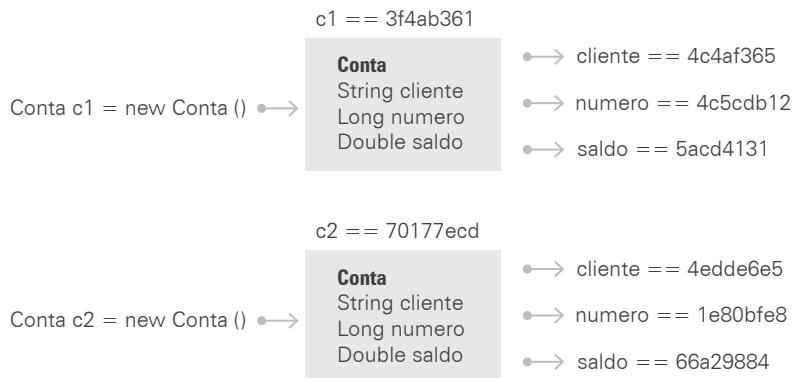
```
1  public class TesteBanco {
2
3      public static void main(String[] args) {
4          Conta c1 = new Conta("Andrey", 300.0);
5          Conta c2 = new Conta("Masiero", 200.0);
6
7          System.out.printf("Conta número %d pertence a %s\n",
8              c1.getNumero(), c1.getCliente());
9          System.out.printf("Conta número %d pertence a %s\n",
10             c2.getNumero(), c2.getCliente());
11      }
12  }
```

Confira o resultado da execução desse código.

```
Conta número 1 pertence a Andrey
Conta número 1 pertence a Masiero
```

Algo não saiu como o esperado. Ambas as contas têm o mesmo número. Deveria ter sido incrementado o número da conta entre os objetos *c1* e *c2*, saindo, assim, a conta número 1 e depois a número 2. Contudo, isso não aconteceu, mas por quê? Observe a figura 1.

Figura 1 – Valores armazenados nas variáveis após a criação da instância dos objetos no sistema

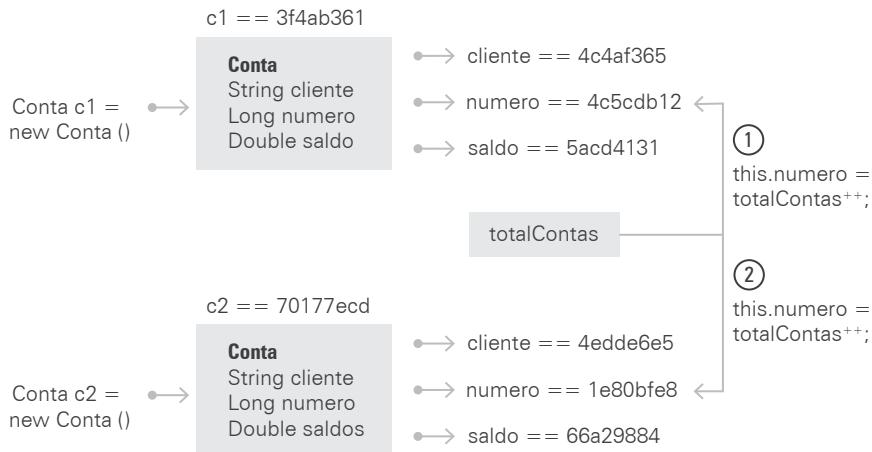


A figura 1 apresenta o que acontece no processo de criação de objetos em sistemas escritos com Java. As variáveis não armazenam valores, e sim endereços de memória, conforme observado nas variáveis comparando o endereço de memória, por exemplo, *c1 == 3f4ab361*. Os endereços são números hexadecimais que servem como um identificador de caixas na estante da memória. Cada objeto tem seu próprio identificador, como observado nas contas *c1* e *c2*. E cada atributo desses objetos também possui seus próprios endereços, conforme mostrado no capítulo 1.

Assim, não é possível compartilhar o mesmo valor entre as classes, e, por consequência, sempre ganhamos o valor 1 como número da conta. Por isso, os atributos de uma classe também são conhecidos como variáveis de instância, já que elas estão vinculadas ao endereço de memória que foram instanciadas ou criadas. A grande questão é: existe um jeito de criar um espaço compartilhado de valores entre as classes? A

resposta é: sim. O Java disponibiliza um recurso chamado atributo estático ou variável de classe. A figura 2 ilustra como funciona esse recurso.

Figura 2 – Criação de uma variável estática para compartilhar valores entre as classes



Na figura 2, foi criada uma variável `totalContas` que, de certa maneira, todos os objetos da classe `Conta` podem acessar. Toda vez que a variável `totalContas` é chamada no código, ela recupera o último valor atribuído. Dessa forma, vamos modificar a classe `Conta` para atender a essa funcionalidade.

```

1  public class Conta {
2      private String cliente;
3      private Long numero;
4      private Double saldo;
5      private static Long totalContas = 1L;
6
7      public Conta(String cliente, Double saldo) {
8          this.cliente = cliente;
9          this.numero = totalContas++;
10         this.saldo = saldo;
11     }
12
13     //...
14
15 }
```

Na linha 5, declaramos o atributo com a palavra reservada `static`. Isso significa que o local na memória onde ele é alocado nunca será modificado. Sendo assim, todo objeto instanciado do tipo `Conta` conhecerá o endereço desse atributo. Perceba que não chamamos `totalContas` com a palavra `this`, pois ele não pertence à instância, e sim à classe como um todo. Poderíamos, em vez da palavra `this`, utilizar o nome da classe para deixar mais explícita a origem em nosso código de `totalContas`, indicando que ela é uma variável da classe. Se utilizar esse recurso de maneira explícita, a linha 9 deve ser apresentada da seguinte maneira:

```
this.numero = Conta.totalContas++;
```

Vamos conferir agora se o resultado foi obtido com sucesso.

```
Conta número 1 pertence a Andrey  
Conta número 2 pertence a Masiero
```

Da mesma maneira que temos atributos estáticos também temos os métodos estáticos. Mas qual seria a vantagem efetiva de um método estático? Para entender melhor a vantagem, vamos mudar o cenário bancário para outro mais comum e simples na vida de um desenvolvedor. Uma das tarefas que precisamos fazer todos os dias dentro de um sistema é formatar valores para exibi-los ao usuário. Para concentrar tudo em um único local, criamos uma classe com os métodos de formatação da saída ao usuário. Essas classes são conhecidas como *helpers* (auxiliares, em tradução livre).

Para ilustrar a situação, vamos construir um *helper* para data. Ele deve receber um objeto de data do sistema (`LocalDate`) e devolver uma *string* formatada no padrão brasileiro. Confira o código a seguir:

```
1 import java.time.LocalDate;
2 import java.time.format.DateTimeFormatter;
3
4 public class DataHelper {
5
6     public String formataData(LocalDate data) {
7         DateTimeFormatter formatter =
8             DateTimeFormatter.ofPattern("dd/MM/yyyy");
9         return data.format(formatter);
10    }
11 }
```

Na sequência, criaremos uma classe para testar nosso helper. O código da classe teste fica da seguinte maneira:

```
1 public class TesteDataHelper {
2     public static void main(String[] args) {
3         DataHelper helper = new DataHelper();
4         LocalDate hoje = LocalDate.now();
5         System.out.println(helper.formataData(hoje));
6     }
7 }
```

Veja que precisamos criar uma instância da nossa classe *DataHelper* (linha 3) apenas para usar um método simples de formatação de data (linha 5). Além disso, esse é um método que poderá ser utilizado em diversos pontos do nosso sistema. Se utilizarmos a estratégia implementada, teremos muitos objetos sendo criados sem necessidade.

Nesse tipo de cenário, os métodos estáticos facilitam para nós. Um método estático pode ser invocado sem criar uma instância da classe. Ele é um método da classe, e não do objeto da instância da classe. Vamos transformar o método *formataData* em estático.

```
1 import java.time.LocalDate;
2 import java.time.format.DateTimeFormatter;
3
4 public class DataHelper {
5
6     public static String formataData(LocalDate data) {
7         DateTimeFormatter formatter =
8             DateTimeFormatter.ofPattern("dd/MM/yyyy");
9         return data.format(formatter);
10    }
11 }
```

Incluímos a palavra reservada *static* na declaração do método *formataData* (linha 6). No próximo passo, modificaremos a nossa classe de teste. Veja a nova implementação.

```
1  public class TesteDataHelper {
2      public static void main(String[] args) {
3          LocalDate hoje = LocalDate.now();
4          System.out.println(DataHelper.formataData(hoje));
5      }
6  }
```

Note que a chamada de um método estático é por meio do nome da classe diretamente, sem a necessidade de uma nova instância (linha 4). Se analisarmos os códigos, vamos perceber outros métodos estáticos em uso, como o método *now* da classe *LocalDate* (linha 3). Métodos estáticos são grandes aliados quando sabemos suas aplicações de maneira eficiente. Agora que você já sabe sobre eles, esperamos que os use em seus sistemas com muita sabedoria.

Considerações finais

Neste capítulo, entendemos o que são constantes e como utilizá-las para tornar nosso código mais sucinto e com maior facilidade de manutenção. Aprendemos também a importância dos atributos e dos métodos estáticos em nossos códigos e como essa técnica nos permite compartilhar recursos na memória de nosso computador.

Referências

DEITEL, Paul; DEITEL, Harvey. **Java**: como programar. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

HORSTMANN, Cay S.; CORNELL, Gary. **Core Java, volume 1**: fundamentos. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

Capítulo 8

Genéricos

Imagine que você precisa utilizar uma lógica de determinado código, aplicado a diversos tipos de objetos. Até 1999, isso era feito sempre por meio da herança direta da classe *Object*, em códigos escritos com Java. Desde então, o release 1.5 do Java, ou Java 5.0 SE, apresentou

o conceito de Generics ao mundo do desenvolvimento. Com Generics, tornou-se possível escrever um código que possa ser utilizado por diversas classes, sendo assim, um código mais genérico (HORSTMANN; CORNELL, 2010).

Uma classe genérica é perfeita para trabalhar com coleções de dados, e provavelmente você já a utilizou em algum momento da sua carreira de desenvolvedor. Se não estiver acreditando, por acaso você já utilizou a classe *ArrayList* em algum cenário para armazenar uma coleção de dados na memória do sistema? Pois bem, a declaração do *ArrayList* vem acompanhada de um par de parênteses angulares (DEITEL; DEITEL, 2010). Entre esses parênteses, determinamos o tipo da coleção no momento do uso. Por exemplo, uma coleção de inteiros seria declarada *ArrayList<Integer>*, seguida do nome de acesso (HORSTMANN; CORNELL, 2010).

Existe uma outra motivação para a criação do conceito de Generics. Antes havia uma sobrecarga de diversos métodos que executavam o mesmo código, mas recebiam parâmetros com tipos diferentes. Com a ideia do método genérico, fica mais fácil desenvolver sistemas com sobrecarga de métodos apenas quando existe a necessidade de alterar a lógica do método de acordo com os parâmetros recebidos (DEITEL; DEITEL, 2010).

Vamos entender como era o desenvolvimento genérico antes da versão 5.0 do Java. Como mencionado, todo trabalho com tipos genéricos era realizado por meio do uso do polimorfismo sobre a classe *Object* (HORSTMANN; CORNELL, 2010). Veja um exemplo desse cenário por meio da implementação da classe *ArrayList*, anteriormente ao Java 5.0 SE.

```
1  public class ArrayList {
2      private Object[] elementos;
3
4      public Object get(int i) { ... }
5      public void add(Object o) { ... }
6      //...
7  }
```

Observe que o tipo declarado do vetor de elementos é *Object*. Isso permite que ele receba objetos de quaisquer tipos para armazenar. Durante o uso do *ArrayList* nas demais classes do sistema era possível atribuir quaisquer valores no conjunto de dados. O uso de coerção dos dados era fator obrigatório para a recuperação dos dados (DEITEL; DEITEL, 2010; HORSTMANN; CORNELL, 2010). Veja o exemplo a seguir do uso da classe *ArrayList*.

```
1     ArrayList nomes = new ArrayList();
2     //...
3     nomes.add("Andrey Masiero");
4     nomes.add(123);
5
6     String nome = (String) nomes.get(0);
```

Nas linhas 3 e 4, são realizadas as operações de inclusão no *ArrayList*, porém, cada linha insere um tipo diferente de dado. Na linha 3, temos a inclusão de uma *String*, seguida da inclusão de um número inteiro na linha 4. Essa operação é totalmente possível graças ao polimorfismo e principalmente ao uso de um array do tipo *Object* declarado na classe *ArrayList*. Entretanto, esse tipo de situação pode gerar problema ao recuperar o dado da coleção.

No processo de recuperação, é necessário fazer a coerção do dado para utilizá-lo, como no exemplo da linha 6. A coerção para *String* no primeiro elemento funciona, mas no segundo teremos um erro. Esse tipo de situação era bem comum nos primórdios dos sistemas escritos em Java. Isso amplia a possibilidade de ter um problema em tempo de execução do sistema. Erros em tempo de execução, nesse caso por causa de conversão entre variáveis, devem sempre ser evitados. O objetivo deve ser aumentar a segurança do desenvolvimento, trabalhando em situações em que possam ocorrer erros em tempo de compilação. O custo de erros em tempo de compilação é muito menor que o custo do tempo em execução, isso permite corrigir o problema antes mesmo que qualquer outra pessoa utilize o sistema.

Dessa maneira, a aplicação de métodos e classes genéricas será discutida neste capítulo. Vamos compreender como podemos trabalhar com essa técnica versátil, poderosa e elegante, para preparar o código para resolver diversos problemas. Vamos começar pelos métodos genéricos e descobrir a solução para o problema da exibição dos valores de um vetor.

1 Parâmetro de tipo

Com a introdução de Generics, temos uma solução para a abordagem do uso de objetos declarados como tipo *Object*. Desse momento em diante temos o que chamamos de parâmetros de tipo. Eles permitem escrever o código com uma especificação genérica e determinar seu tipo apenas na declaração e na instância dos objetos. Veja como fica a declaração do *ArrayList* agora com o uso da técnica de Generics (HORSTMANN; CORNELL, 2010).

```
ArrayList<String> nomes = new ArrayList<String>();
```

São inclusos parênteses angulares com o tipo *String* entre eles, logo após a declaração do tipo *ArrayList*. Isso faz com que o único tipo de variável aceita para armazenar no *ArrayList* seja o *String*. Poderia ser qualquer tipo que desejar, o tipo *String* é apenas um exemplo. Com o uso do parâmetro de tipo, se tentarmos inserir qualquer tipo diferente, em tempo de compilação, já obteremos um erro (DEITEL; DEITEL, 2010; HORSTMANN; CORNELL, 2010). A partir da declaração do parâmetro de tipo, nenhum dado diferente é inserido na coleção, o que evita o problema da coerção e mantém o código mais limpo e com fácil manutenção.

Outro ponto importante que podemos resolver com o uso de Generics é a sobrecarga de métodos. Vamos analisar o seguinte exemplo, apresentado por Deitel e Deitel (2010):

```
1  public class Metodos {
2
3      public static void main(String[] args) {
4          Integer[] inteiros = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
5          Double[] reais = {1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5};
6          Character[] letras = {'A', 'N', 'D', 'R', 'E', 'Y'};
7
8          System.out.println("Exibição do vetor de inteiros:");
9          exibe(inteiros);
10         System.out.println("Exibição do vetor de reais:");
11         exibe(reais);
12         System.out.println("Exibição do vetor de letras:");
13         exibe(letras);
14     }
15
16     private static void exibe(Integer[] inteiros) {
17         for(Integer elemento : inteiros) {
18             System.out.printf("%s ", elemento);
19         }
20         System.out.println();
21     }
22
23     private static void exibe(Double[] reais) {
24         for(Double elemento : reais) {
25             System.out.printf("%s ", elemento);
26         }
27         System.out.println();
28     }
29
30     private static void exibe(Character[] letras) {
31         for(Character elemento : letras) {
32             System.out.printf("%s ", elemento);
33         }
34         System.out.println();
35     }
36 }
37 }
```

No código demonstrado, temos três vetores com tipos diferentes: inteiro (linha 4), ponto flutuante (linha 5) e caracteres (linha 6). O objetivo da operação é exibir os três vetores na linha de comando. Para essa tarefa, foram criados três métodos chamados *exibe*, e cada um difere do outro apenas no parâmetro recebido. A sobrecarga de método apresentada no exemplo do código pode muito bem ser substituída por um único método, aplicando o mesmo princípio de Generics utilizado pela classe *ArrayList*.

2 Métodos genéricos

Um método genérico pode receber e retornar o tipo de parâmetro desejado. Para isso, declaramos ele com uma letra *T* maiúscula no lugar

do tipo que deve ser substituído. Em tempo de compilação, o Java prepara esse método para receber um tipo diferente, de acordo com a invocação feita.

Vamos analisar a nova versão do código de sobreulação de métodos para exibir os elementos de determinado vetor. A classe *MetodoGenerico* foi criada apenas para facilitar a apresentação de uma das maneiras de invocar métodos genéricos. Confira o código a seguir:

```
1  public class MetodoGenerico {  
2  
3      public static <T> void exibe(T[] elementos) {  
4          for(T elemento : elementos) {  
5              System.out.printf("%s ", elemento);  
6          }  
7          System.out.println();  
8      }  
9  }
```

Perceba que o método genérico foi declarado dentro de uma classe comum. Não é necessário que a classe inteira seja genérica para termos métodos genéricos, como o caso das classes abstratas. Antes do tipo de retorno do método, definimos um par de parênteses angulares envolvendo a letra *T* (linha 3). Essa mesma letra *T* é usada para definir o parâmetro recebido e o tipo do elemento no *for* apresentado nas linhas 4 a 6. A letra *T* será substituída por um tipo definido durante a chamada do método no momento do uso.



PARA SABER MAIS

Existe uma convenção nas letras que utilizamos para cravar os tipos de dados genéricos em Java. A biblioteca Java utiliza a letra E para o tipo de elemento em uma coleção e K e V para o conjunto de chave e valor, respectivamente. As letras T, U e S para quaisquer outros tipos. Para acessar em detalhes, faça a busca por convenção de letras para Generics em Java e você será direcionado para a documentação oficial.

Vamos ver a primeira opção para invocar os métodos genéricos a partir da classe `TesteMetodoGenerico`. Confira a seguir:

```
1  public class TesteMetodoGenerico {  
2  
3      public static void main(String[] args) {  
4          Integer[] inteiros = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
5          Double[] reais = {1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5};  
6          Character[] letras = {'A', 'N', 'D', 'R', 'E', 'Y'};  
7  
8          System.out.println("Exibição do vetor de inteiros:");  
9          MetodoGenerico.<Integer>exibe(inteiros);  
10         System.out.println("Exibição do vetor de reais:");  
11         MetodoGenerico.<Double>exibe(reais);  
12         System.out.println("Exibição do vetor de letras:");  
13         MetodoGenerico.<Character>exibe(letras);  
14     }  
15 }  
16 }
```

Nas linhas 9, 11 e 13, temos a invocação do método genérico. Repare que antes do nome do método, entre parênteses angulares, determinamos o tipo. Ele substituirá a letra *T* na declaração do método para aquele uso. Essa operação é transparente para o desenvolvedor; quem a realiza é o compilador de bytecode do Java. Porém, o Java já consegue inferir diretamente o tipo de dados que será utilizado no método pelo tipo do parâmetro inserido na chamada. Assim, podemos simplificar a chamada do método durante o uso. Veja como fica:

```
1  System.out.println("Exibição do vetor de inteiros:");  
2  MetodoGenerico.exibe(inteiros);  
3  System.out.println("Exibição do vetor de reais:");  
4  MetodoGenerico.exibe(reais);  
5  System.out.println("Exibição do vetor de letras:");  
6  MetodoGenerico.exibe(letras);
```

Simplesmente podemos invocar o método pelo nome, deixando o código mais limpo e legível. Vamos pensar em outro cenário para o uso de método genérico que pode ser muito útil. Neste novo problema, gostaríamos de encontrar o maior valor em uma quantidade X de valores. Na invocação do método *maior*, poderá ser passado de 1 a N objetos, isso ficará a cargo do desenvolvedor. A questão principal agora é: como garantir que os objetos informados são comparáveis? Podemos utilizar uma interface, mas então esse método não será genérico, certo? Pois bem, podemos determinar o que chamamos de limite superior de um

parâmetro de tipo. Esse limite superior é uma declaração que fazemos no tipo genérico dizendo que ele precisa ser filho de determinada classe ou implementar determinada interface.

Com o limite superior, podemos garantir que os objetos utilizados na invocação do método genérico implementem determinada operação. Não perdemos, assim, a beleza dos tipos genéricos e mantemos a harmonia dos contratos entre os objetos estipulados na modelagem de nosso sistema. Vamos implementar agora o método *maior* de maneira genérica.

```
1  public static <T extends Comparable<T>> T maior(T... valores) {  
2      T maior = null;  
3      for(T valor : valores) {  
4          if(maior == null || valor.compareTo(maior) > 0) {  
5              maior = valor;  
6          }  
7      }  
8      return maior;  
9  }
```

No método *maior*, temos uma implementação com alguns detalhes a mais do que no *exibe*. O primeiro ponto que nos chama atenção é a declaração do parâmetro de tipo; nele, adicionaram-se a expressão *extends* e a interface *Comparable*, também genérica. O que isso significa? Neste momento, determinamos o limite dos tipos aceitos nesse método. Apenas os tipos que implementam a interface *Comparable* podem ser passados como parâmetro de tipo. Isso é interessante, pois podemos utilizar quaisquer classes (não perdendo o objetivo de ser genérico) que possuam determinado método. Assim, podemos garantir que o nosso código fará a operação proposta de maneira adequada.

Utilizou-se também o *T* para o tipo de retorno do método. Os métodos genéricos também são utilizados para retornar qualquer tipo desejado, que, nesse caso, é o mesmo recebido como parâmetro. No parâmetro recebido pelo método, temos um novo operador, os três pontos. Esse operador é conhecido como *varargs*, que significa argumentos variáveis. Ele permite receber uma quantidade variável de objetos em

sua chamada. Pode ser um valor entre 0 e infinito. Dessa maneira, além do tipo recebido, também temos uma quantidade variável de objetos enviados para o processamento do método.

A variável maior é declarada com o tipo *T*, e ela deverá ser retornada no final do processo do método. A comparação feita dentro do método é por meio do método *compareTo* da interface *Comparable*.



PARA SABER MAIS

O método *compareTo* funciona da seguinte maneira:

Ao executar o código *object1.compareTo(object2)*, ele retorna o valor de acordo com a regra:

- 0 se os objetos forem iguais;
- número inteiro negativo se *object1* for menor que *object2*;
- número inteiro positivo se *object1* for maior que *object2*.

Assim, conseguimos determinar a regra de comparação dentro da classe que estamos enviando, facilitando a atribuição das responsabilidades de cada objeto. Vamos verificar como fica a execução desse método.

```
1  public class TesteMetodoGenerico {  
2  
3      public static void main(String[] args) {  
4          System.out.printf("Maior inteiro: %s\n",  
MetodoGenerico.maior(10, 5, 20));  
5          System.out.printf("Maior real: %s\n",  
MetodoGenerico.maior(10.5, 5.1, 20.5, 15.0, 20.6));  
6          System.out.printf("Maior letra: %s\n",  
MetodoGenerico.maior('A', 'B', 'D', 'C'));  
7      }  
8  }  
9 }
```

O resultado obtido com essa operação é o seguinte:

```
Maior inteiro: 20  
Maior real: 20.6  
Maior letra: D
```

Observe que a quantidade e os tipos das variáveis passadas são diferentes entre cada chamada do método. E, mesmo assim, ele conseguiu obter o resultado esperado. Assim, conseguimos fazer com que o nosso sistema seja expansível, sem muito esforço em implementações de métodos que devem promover a mesma lógica.

3 Classes genéricas

Assim como métodos, podemos ter classes inteiras genéricas. As classes genéricas podem receber um ou mais parâmetros de tipo. As aplicações mais comuns de classes genéricas são em coleções de dados. Vamos ver o exemplo de uma coleção do tipo *Pilha*, implementada sem a utilização da classe genérica. Posteriormente, a implementação da mesma classe utilizando a técnica de Generics. Confira a implementação da classe *Pilha* tradicional:

```
1  public class Pilha {  
2      private ArrayList<Integer> numeros = new ArrayList<>();  
3  
4      public void empilha(Integer numero) {  
5          numeros.add(numero);  
6      }  
7  
8      public Integer desempilha() {  
9          if (numeros.isEmpty()) return null;  
10         return numeros.remove(numeros.size() - 1);  
11     }  
12 }  
13 }  
14 }
```

Na implementação, utilizamos um *ArrayList* (linha 3) como base de armazenamento para focar apenas as diferenças entre uma *pilha* tradicional e a genérica. Para criarmos o *ArrayList*, precisamos determinar o

tipo de dado que será armazenado. No exemplo, escolhemos números inteiros, o que limita nossa implementação. Se precisarmos de uma *pilha* de funcionários, deveremos refazer essa implementação para aceitar o tipo de funcionário.

O limitante do *ArrayList* é expandindo para todo o restante da classe. O método *empilha* recebe apenas números inteiros como parâmetro. O método *desempilha* retorna apenas números inteiros como resultado do processamento. Todos esses métodos devem ser modificados para cada tipo de objeto que seja necessário trabalhar. Vamos ver como é o resultado da implementação genérica da *pilha*.

```
1  public class Pilha<E> {
2
3      private ArrayList<E> elementos = new ArrayList<>();
4
5      public void empilha(E valor) {
6          elementos.add(valor);
7      }
8
9      public E desempilha() {
10         if (elementos.isEmpty()) return null;
11         return elementos.remove(elementos.size() - 1);
12     }
13 }
14 }
```

Adicionamos na declaração da classe o espaço do parâmetro de tipo ao lado do nome da classe (linha 1). Dessa vez, utilizamos a letra *E* por se tratar de elementos de uma coleção de dados. O mesmo parâmetro de tipo recebido na declaração da *pilha* é repassado na declaração do *ArrayList* utilizado como apoio para essa tarefa (linha 3). Os métodos *empilha* e *desempilha* também recebem o parâmetro de tipo em suas declarações, facilitando, assim, o uso da classe. Conheça a implementação de uma classe para teste da classe genérica.

```
1  public class TestePilha {
2      public static void main(String[] args) {
3          Pilha<Integer> numeros = new Pilha<>();
4          numeros.empilha(1);
5          System.out.println(numeros.desempilha());
6
7          Pilha<String> nomes = new Pilha<>();
8          nomes.empilha("Andrey");
9          System.out.println(nomes.desempilha());
10     }
11 }
```

A implementação da *Pilha* genérica permite que utilizemos o mesmo código para diversos objetos, como, no exemplo, *Integer* e *String*. Da mesma forma que temos a implementação de classes genéricas, podemos criar interfaces genéricas. Voltemos à implementação da interface *Predicado* para compor uma regra que valida valores com base em alguma regra. Vamos analisar o *Predicado* anterior, sem o uso de Generics.

```
1     public interface Predicado {
2         Boolean valida(Integer valor);
3     }
```

Nesse cenário, apenas valores inteiros poderiam ser testados por meio da interface *Predicado*. Uma maneira que desenvolvedores podem utilizar para resolver essa situação é por meio da declaração dos métodos com os demais tipos de objetos que podem ser aceitos, mas seriam milhares, e até a escalabilidade do sistema seria prejudicada. Vamos fazer a implementação da interface *Predicado* de maneira genérica.

```
1     public interface Predicado<T> {
2         Boolean valida(T valor);
3     }
```

Adicionamos o parâmetro de tipo no *Predicado* e alteramos o valor recebido pelo método *valida* para o mesmo tipo de parâmetro. O próximo passo é implementar o método para realizar filtros de uma lista, na classe de utilitários do sistema.

```
1     public class Utils<T> {
2
3         public List<T> filtra(List<T> elementos, Predicado<T> predicado) {
4             return elementos
5                 .stream()
6                 .filter(elemento -> predicado.valida(elemento))
7                 .collect(Collectors.toList());
8
9         }
10    }
```

A classe *Utils* também recebe um parâmetro de tipo que é repassado para os elementos de retorno do método *filtra* e seu respectivos parâmetros, um do tipo *List* e outro do tipo *Predicado*. Como ambos também são

genéricos, eles recebem o mesmo parâmetro determinado na classe *Utils*. Vejamos como fica a aplicação da classe *Utils* em um cenário que filtra os números pares e outro que filtra os nomes que iniciam com A.

```
1  public class TesteFiltro {
2      public static void main(String[] args) {
3          List<Integer> numeros = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9,
10);
4          Utils<Integer> filtroNumerico = new Utils<>();
5          List<Integer> pares = filtroNumerico.filtrar(numeros, numero ->
numero % 2 == 0);
6
7          List<String> nomes = List.of("Andrey", "Ana", "Joao",
"Kathia", "Anabele", "Barbara", "Gerliane");
8          Utils<String> filtroNomes = new Utils<>();
9          List<String> nomesComA = filtroNomes.filtrar(nomes, nome ->
nome.toUpperCase().startsWith("A"));
10
11         System.out.printf("Números pares:%22s\n", pares);
12         System.out.printf("Nomes com A:%24s\n", nomesComA);
13
14     }
15 }
```

Repare como ficou mais fácil trabalhar com as classes genéricas, uma vez que a necessidade da implementação pelo tipo de variável em uso é independente da lógica aplicada na solução do problema.

4 Exercícios de fixação

Os exercícios a seguir lhe guiarão para colocar em prática tudo que foi aprendido ao longo do capítulo.

1. Crie uma interface genérica para efetuar a translação de objetos 2D.
2. Crie uma interface genérica para efetuar a translação de objetos 3D.
3. Crie uma interface genérica para efetuar a escala de objetos 2D.
4. Crie uma interface genérica para efetuar a escala de objetos 3D.
5. Crie uma interface genérica para efetuar a rotação de objetos 2D.
6. Crie uma interface genérica para efetuar a rotação de objetos 3D.

7. Teste no conjunto de classes de *Forma* com diversos tipos de medida.

Considerações finais

Generics é um dos conceitos mais importantes dentro da programação em Java para cuidar da escalabilidade e da manutenção do código. Com ele, é possível escrever um código e executar com diversos tipos diferentes.

Inicie aos poucos o uso de genéricos para adquirir maturidade e se sentir confortável até ter a experiência para desenvolver seu próprio código genérico. Mas a partir do momento que você o construir é um caminho sem volta. Existem muitos fóruns e comunidades para discutir e aprender cada vez mais. Só depende de você agora.

Referências

DEITEL, Paul.; DEITEL, Harvey. **Java**: como programar. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

HORSTMANN, Cay S.; CORNELL, Gary. **Core Java, volume 1**: fundamentos. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

Sobre os autores

Allen Oberleitner é coordenador acadêmico do curso superior de tecnologia em análise e desenvolvimento de sistemas e banco de dados e data science no Centro Universitário Fiap. Professor de disciplinas relacionadas à linguagem de programação, engenharia de software e marketing digital no Centro Universitário Fiap e de pós-graduação no Centro Universitário Toledo Prudente. Possui mestrado em engenharia da informação pela Universidade Federal do ABC (2012); pós-graduação em gestão financeira pela Universidade Municipal de São Caetano do Sul (2005); e graduações em ciência da computação pela Universidade Municipal de São Caetano do Sul (2003) e ciência & tecnologia pela Universidade Federal do ABC (2010). Tem várias publicações na área de lógica de programação e experiência de 15 anos em processos de aprendizagem, com artigos apresentados em congressos internacionais.

Andrey Araujo Masiero é professor e desenvolvedor de software com interesse em arte, tecnologia e cultura. Doutor em engenharia elétrica na área de inteligência artificial pelo Centro Universitário FEI e bolsista Prosup/Capes. Recebeu o prêmio destaque acadêmico 2015, na instituição em que cursa o doutorado, pelo projeto RoboFEI @Home. Mestre em engenharia elétrica na área de inteligência artificial pelo Centro Universitário FEI e bolsista de desenvolvimento tecnológico industrial do CNPq – Nível B, tempo integral, do projeto FINEP Pesquisa e Estatística baseada em Acervo Digital de Prontuário Médico do Paciente em Telemedicina Centrada no Usuário. Bacharel em ciência da computação pelo Centro Universitário FEI, foi premiado com o primeiro lugar na Exposição dos Trabalhos Acadêmicos da Ciência da Computação (Expocom) com o trabalho de sistema de gerenciamento de patterns, anti-patterns e personas (SIGEPAPP). Profissional com 16 anos de experiência no mercado, participou de projetos com o Governo do Estado de São Paulo e projetos de sociedade privada, como a integração entre os bancos Santander e Real.