



SOFTWARE ORIENTADO A OBJETOS

© 2018 POR EDITORA E DISTRIBUIDORA EDUCACIONAL S.A.

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Editora e Distribuidora Educacional S.A.

Presidente

Rodrigo Galindo

Vice-Presidente de Pós-Graduação e Educação Continuada

Paulo de Tarso Pires de Moraes

Conselho Acadêmico

Carlos Roberto Pagani Junior
Camila Braga de Oliveira Higa
Carolina Yaly
Danielle Leite de Lemos Oliveira
Juliana Caramigo Gennarini
Mariana Ricken Barbosa
Priscila Pereira Silva

Coordenador

Tayra Carolina Nascimento Aleixo

Revisor

Sérgio Eduardo Nunes

Editorial

Alessandra Cristina Fahl
Daniella Fernandes Haruze Manta
Flávia Mello Magrini
Hâmila Samai Franco dos Santos
Mariana de Campos Barroso
Paola Andressa Machado Leal

Dados Internacionais de Catalogação na Publicação (CIP)

S237s Santos, Márcio dos
Software orientado a objetos/ Márcio dos Santos –
Londrina: Editora e Distribuidora Educacional S.A. 2018.
114 p.

ISBN 978-85-522-1077-1

1. Software. 2. Computadores digitais (programação). I.
Santos, Márcio dos. Título.

CDD 000

Thamiris Mantovani CRB: 8/9491

Editora e Distribuidora Educacional S.A.
Avenida Paris, 675 – Parque Residencial João Piza
CEP: 86041-100 — Londrina — PR
e-mail: editora.educacional@kroton.com.br
Homepage: <http://www.kroton.com.br/>



SUMÁRIO

Apresentação da Disciplina	4
TEMA 01	5
Conceitos iniciais de POO: Abstração de dados e identificação de objetos	
TEMA 02	19
Objeto: Atributos e métodos	
TEMA 03	33
Classes e instância de objetos	
TEMA 04	48
Encapsulamento	
TEMA 05	70
Getters e Setters	
TEMA 06	89
Herança	
TEMA 07	105
Polimorfismo	
TEMA 08	123
Polimorfismo de Sobrecarga ou reescrita (sobreposição) de métodos	



Apresentação da disciplina

Você aprenderá, nessa disciplina, a respeito das bases que fundamentam o paradigma da Programação Orientada a Objetos, e poderá compreender o que torna essa metodologia tão necessária no desenvolvimento de sistemas, independentemente de seu porte.

Embora o contexto do curso, como um todo, possua cunho de Tecnologia da Informação, nada impede que seja realizado com êxito por qualquer pessoa com formação superior, em áreas que não sejam da computação.

Para que haja total eficácia no aprendizado, recomendamos que você dedique atenção especial à essa disciplina, que introduzirá ao mundo da Programação Orientada a Objetos para desenvolvimento de softwares eficazes, ágeis e de desenvolvimento célere.

Conforme os conteúdos forem apresentados, surgirão exemplos que auxiliarão a absorver a essência da explicação, bem como sua aplicabilidade prática no desenvolvimento de sistemas. Dessa forma, recomendamos que você execute os exercícios propostos, mas que também crie o hábito de extrapolar o material de estudo, indo além do que é proposto, expandindo ainda mais suas possibilidades de assimilação.

Atente aos trechos destacados por Assimile, Exemplificando e Para saber mais, pois esses espaços possuem conteúdos extras que auxiliarão ainda mais na absorção dos temas.

Por fim, seja bem-vindo e tenha uma excelente jornada de estudos, nesse extenso e fascinante mundo da computação eficiente!



TEMA 01

CONCEITOS INICIAIS DE POO: ABSTRAÇÃO DE DADOS E IDENTIFICAÇÃO DE OBJETOS

► Objetivos

- Compreender o que é o Paradigma da Programação Orientada a Objetos.
- Identificar objetos do mundo real para futura conversão em instâncias dentro da estrutura de um sistema.
- Conhecer as características e ações que um objeto pode ter/ realizar.
- Entender a importância da Orientação a Objetos.

► Introdução

No mundo da programação de sistemas existem diversas formas de atingir o mesmo objetivo, seja por meio de metodologias, padrões, uso de frameworks, ou ainda de maneira empírica, baseado em boas práticas já vivenciadas e comprovadas anteriormente.


Especificamente quanto ao desenvolvimento de um software, existem, por padrão, três paradigmas que são utilizados: Programação Estruturada, Programação Procedural e Programação Orientada a Objetos.

Algumas literaturas tratam os dois primeiros paradigmas como sendo um só, assim como aponta Ana Fernanda Gomes, professora universitária da área de informática, desde 1994, que dá ao paradigma estruturado a mesma equivalência do paradigma imperativo ou procedural (ASCENCIO, 2012, p. 12)¹. Outras já os abordam como sendo paradigmas distintos, ou ainda, que consideram o paradigma Procedural como sendo uma ramificação do paradigma da programação estruturada, como nos demonstra Luis Joyanes Aguilar:

No paradigma de programação procedimental (procedural ou por procedimentos), um problema se modela diretamente por meio de um conjunto de algoritmos. Por exemplo, a folha de pagamento de uma empresa ou a administração de vendas de uma loja de departamentos, é representada como uma série de procedimentos que manipulam dados. Os dados são armazenados separadamente e acessados por meio de uma posição global ou de parâmetros nos procedimentos. Três linguagens de programação clássicas, FORTRAN, Pascal e C, produziram um arquétipo da programação procedimental, também estreitamente relacionada e, às vezes, conhecida como programação estruturada. A programação com suporte em C++ proporciona o paradigma procedimental com uma ênfase em funções, modelos de funções e algoritmos genéricos (AGUILAR, 2011, p. 25)².

¹ ASCENCIO, Ana Fernanda Gomes. **Fundamentos da programação de computadores:** algoritmos, PASCAL, C/C++, (padrão ANSI) e JAVA. 3. ed. São Paulo: Pearson Education do Brasil, 2012.

² AGUILAR, Luis Joyanes. **Programação em C++:** algoritmos, estruturas de dados e objetos. 2. ed. Porto Alegre: AMGH, 2011.



Embora existam visões diferentes quanto ao mesmo tema, o mais importante ao desenvolvedor de software é que ele tenha a capacidade de compreender o que difere uma metodologia da outra, sabendo, assim, qual adotar em diferentes situações.

Um pouco mais adiante será explanado, em conceitos rápidos e diretos, os dois primeiros paradigmas (Procedural e Estruturado) para que você consiga realizar um comparativo de desempenho, tempo de desenvolvimento e possibilidade de reutilização de código entre a Programação Orientada a Objetos, em detrimento dos demais paradigmas.

Num primeiro momento, você precisa internalizar que as boas práticas de desenvolvimento, e o mercado por si só, tendem a adotar e padronizar a POO Programação Orientada a Objetos (POO) como único meio de desenvolvimento de software. Isso não descarta os outros dois paradigmas, pois existem muitas linguagens que nativamente não foram concebidas para serem orientadas a objeto, como a linguagem C, por exemplo.

A POO foi criada por Alan Kay (criador da linguagem Smalltalk), embora já houvesse sido conceituada anos antes por Ole-Johan Dahl e Kristen Nygaard no final da década de 1960, com o objetivo de abstrair os elementos do mundo real e levá-los ao mundo da computação, como forma de resolver problemas corriqueiros de duplicidade e reescrita de código, além do desempenho de softwares em si.

Atualmente, muitas linguagens já são multiparadigmas, ou seja, aceitam ser escritas de forma estruturada, procedural ou utilizando orientação a objetos. De maneira geral, a estrutura abordada acabará por resultar num produto final funcional, porém a manutenibilidade da aplicação e o tempo de desenvolvimento é que são os grandes fatores envolvidos nessa equação. Obviamente que até mesmo desenvolvimentos de forma estruturada ou procedural, e com uma boa base lógica, poderão ter melhor desempenho se comparados a sistemas desenvolvidos com POO e com uma base lógica deficiente. Assim, é importante que o desenvolvedor

esteja com a Lógica de Programação devidamente adequada para iniciar o desenvolvimento, independente do paradigma utilizado.

LINK



Confira os principais itens e instruções de programação no texto “Algoritmo e Lógica de Programação - Parte I: DCA 800 - Engenharia Química”. Disponível em: <https://www.dca.ufrn.br/~affonso/DCA800/pdf/algoritmos_parte1.pdf>. Acesso em: 22 jul. 2018.

► 1. Conceitos de programação estruturada e procedural

A programação estruturada segue o princípio básico da programação de sistemas, em que uma aplicação é interpretada pelo compilador de um computador, sendo lida de cima para baixo, da esquerda para a direita, assim como na leitura de um livro. É importante salientar que a POO também segue, num contexto geral, esse conceito de execução (esquerda para a direita, de cima para baixo), mas com a possibilidade de recorrer a linhas já passadas ou adiantar-se a linhas pospostas, além de poder recorrer a muitos outros arquivos que contenham continuidade do código. É como se você, na leitura de algum livro, pudesse pular alguns parágrafos ou alguns capítulos, sem necessariamente ler todo o conteúdo entre eles. Isso não significa que o conteúdo não lido não tenha importância, tampouco que não deveriam compor aquele livro; apenas significa que, naquele momento, a leitura de determinadas partes poderia ser deixada de lado, direcionando o fluxo de leitura a outras seções de maior relevância para aquela situação.

Os comandos são passados ao computador, que os executa de forma sequencial. Vejamos um exemplo de um algoritmo, utilizando pseudocódigo (ou portugol, como é chamado em algumas literaturas):



PARA SABER MAIS

Algoritmos são sequências lógicas de comandos utilizados para a resolução de problemas, podendo ser utilizados em qualquer situação e não apenas na computação. Existem algoritmos para ir ao trabalho, para fritar um ovo ou para resolver uma equação algébrica. Lembre-se de que, independente da aplicabilidade, todo algoritmo tem começo, meio e fim e seu contexto é composto por passos lógicos.

Algoritmo 1

1. String pessoa, sexo;
2. inteiro idade;
3. pessoa = "João da Silva"
4. sexo = "Masculino";
5. idade = 45;
6. Escreva(pessoa);
7. Escreva(sexo);
8. Escreva(idade);

O exemplo bastante simplório, disposto no **Algoritmo 1**, demonstra a execução estrutura de um sistema, que lê algumas variáveis de texto (linha 1), uma variável numérica inteira (linha 2) e então passa a atribuir valores a essas variáveis nas linhas 3, 4 e 5. Após as variáveis receberem valores, esses são exibidos na tela, conforme linhas 6, 7 e 8.

Atente a um detalhe: as variáveis pessoa, sexo e idade estão com valores atribuídos e em qualquer local do código, em que essas variáveis sejam utilizadas, os valores serão sempre os que foram atribuídos. Existe, claro,

a possibilidade de subscrever essas informações, atribuindo novos valores, porém o valor anterior seria perdido. Vejamos:

Algoritmo 2

1. String pessoa
2. pessoa = "João da Silva"
3. Escreva(pessoa);
4. // o resultado a ser exibido seria "João da Silva"
5. pessoa = "Maria Antônia"
6. Escreva(pessoa);
7. // o resultado a ser exibido seria "Maria Antônia"
8. // O primeiro dado, "João da Silva" seria
9. // simplesmente perdido

E se quiser, na programação estruturada, manter o registro do João da Silva e da Maria Antônia? Pela lógica, precisaria criar variáveis para cada novo registro, aumentando drasticamente a quantidade de código e tempo de programação, além de tornar o desenvolvimento extremamente manual e de difícil manutenção, como veremos mais adiante.

Algoritmo 3

1. String pessoa
2. pessoa = "João da Silva"
3. Escreva(pessoa);
4. pessoa2 = "Maria Antônia"
5. 5 Escreva(pessoa2);
6. ...

Fica fácil de ver a forma complexa que esse algoritmo assumiria, no caso de um crescimento exponencial de dados. A situação poderia tornar-se ainda mais calamitosa se, em algum momento, houvesse a necessidade de adicionar uma nova característica ao algoritmo. Fazendo uma junção entre o **Algoritmo 1** e o **Algoritmo 3**, idealize que cada registro de pessoas, nesse exemplo, é composto por um nome, um sexo e uma idade. Suponha ainda que, no decorrer do uso desse sistema, foram criados mil usuários e, em dado momento, surgiu a necessidade de adicionar uma nova característica a todos eles (data de nascimento, por exemplo). Consegue enxergar a dificuldade em editar todos os mil registros, adicionando a variável `dataNascimento`?

Esse é apenas um dos problemas que a POO vem resolver.

Tendo compreendido a base que envolve a programação estruturada, que, em termos práticos, refere-se à forma sequencial de escrita e execução de um algoritmo computacional, fica simples compreender a programação procedural, que, como já foi dito, é tida por alguns autores, como Ascencio (2012)³, como a mesma coisa que a programação estruturada.

A Programação Procedural também realiza o mesmo conceito da estruturada, mas adiciona algumas pequenas possibilidades de reaproveitamento de código por meio de funções.

PARA SABER MAIS



As funções, na programação, permitem o encapsulamento de várias linhas de código, que são executadas quando a função é executada. Para que isso ocorra, é necessário passar parâmetros a ela, que podem ser de tipos diferentes (textos, números inteiros, números fracionários etc.), mas será sempre necessário informar a mesma quantidade de parâmetros requerida pela função.

³ ASCENCIO, Ana Fernanda Gomes. **Fundamentos da programação de computadores:** algoritmos, PASCAL, C/C++, (padrão ANSI) e JAVA. 3. ed. São Paulo: Pearson Education do Brasil, 2012.

Algoritmo 4

1. função soma(inteiro a, inteiro b)
2. inteiro c = a + b;
3. retorne c;
4. Escreva(soma(10,20))
5. // será exibido o resultado 30 na tela

No exemplo do **Algoritmo 4**, a função denominada soma recebe dois parâmetros inteiros, chamados de a e b, realizando a soma de ambos e armazenando o resultado na variável inteira c. Por fim, a função retorna o valor de c. Na linha 4, é solicitado ao algoritmo que escreva um valor em tela. Esse valor é o resultado da função soma; para que a função seja executada, é necessário informar os parâmetros, ou seja, dois números inteiros.

Esse conceito até realiza o reaproveitamento de código, economizando algumas linhas de programação, mas não é tão versátil quanto a POO, que provê, dentre vários recursos, a segurança estrutural e funcional de um sistema (veremos esse assunto mais adiante, quando tratarmos sobre encapsulamento). Um outro detalhe que deixa a desejar, no paradigma procedural, é que a função tem seus valores subscritos cada vez que novos dados são informados, não sendo possível armazenar um conjunto de valores distintos. Nas palavras de Deitel, “um problema significativo da programação procedural é que as unidades de programa não espelham as entidades reais de maneira eficaz, de modo que essas unidades não são particularmente reutilizáveis”. DEITEL (2011, p. 8)⁴

► 2. Abstração de elementos do mundo real

Vamos entender a respeito do conceito de POO. Pense em uma caneta, que é um exemplo de elemento do mundo real. Sabemos que toda caneta tem algumas características peculiares e executa algumas ações. Uma forma

⁴ DEITEL, P. J. **C: como Programar**. 6. ed. São Paulo: Pearson Prentice Hall, 2011.

fácil de assimilar esse conceito é compreendendo que as características são adjetivos (às vezes podem ser substantivos), enquanto as ações são verbos. Considere que qualquer objeto, na programação, pode interagir com outros objetos, cada qual com seu próprio conjunto de características que são os **atributos**, enquanto “os **métodos**, por sua vez, são os verbos da nossa programação” (FÉLIX, 2016, p. 14)⁵. Vejamos algumas características e ações de uma caneta:

Quadro 1 | Características e ações de uma caneta

Características	Ações
<ul style="list-style-type: none">• Cor.• Marca.• Tipo (esferográfica, tinteiro, de quadro branco etc.).	<ul style="list-style-type: none">• Escrever

Fonte: elaborado pelo autor.

Observe que, no quadro, não foi especificado nenhum valor, ou seja, não sabemos se essa caneta é preta ou azul, nem se ela está em bom estado de funcionamento. A única certeza que temos é que o quadro apresenta características e ações de todas as canetas, assim, temos um escopo padrão (ou um molde) que toda e qualquer caneta pode utilizar. A esse molde damos o nome de classe, tema que exploraremos mais adiante.

ASSIMILE

Numa definição mais precisa, podemos dizer que uma classe é um molde de um objeto, pois todo objeto possui os métodos e atributos que compõem aquela determinada classe. Apesar de uma classe ditar o molde padrão de um objeto, cada objeto criado pode sobrepor atributos ou métodos que foram ditados na classe.

⁵ FÉLIX, Rafael. **Programação Orientada a Objetos**. São Paulo: Pearson Prentice Hall, 2016.

Similar às funções, os métodos (ações de uma classe) também recebem parâmetros. De forma geral, é possível declarar uma classe que utilizará métodos, mas esses métodos serem vazios.

2.1 Aplicação da POO

Com a utilização de POO, podemos elaborar um molde (denominado classe) e criarmos vários objetos, que utilizarão essa classe como base para se formarem. Para assimilarmos de maneira mais prática, vamos entender como a Programação Orientada a Objetos faria com que o nosso Algoritmo 3 se tornasse mais eficiente, do ponto de vista do desenvolvedor. Utilizaremos a mesma lógica, porém aplicando recursos de orientação a objetos.

Sabemos que em nosso algoritmo 3, todas as pessoas possuem nome, sexo e idade. Assim, sempre que uma nova pessoa vier a ser criada em nosso código, não são necessárias novas variáveis para armazenar esses dados, utilizamos as mesmas variáveis que estão dentro da classe Pessoa, sendo necessário apenas atribuir valores a elas.

Recordemos o exemplo aplicado anteriormente. Se esse sistema detiver mil usuários cadastrados e precisarmos adicionar uma nova característica (atributo) a todos eles, não seria necessário editar usuário por usuário, bastaria tão somente adicionar esse atributo à classe Pessoa, considerando que todos os usuários são objetos do tipo pessoa. Bem mais simples, não é mesmo?

EXEMPLIFICANDO

Vamos utilizar pseudocódigo para facilitar a compreensão da explicação:

1. Classe Pública Pessoa
2. String nome, sexo;
3. Inteiro idade;

```
4. Pessoa usuario = new Pessoa();
5. usuario.nome = "João da Silva";
6. usuario.sexo = "Masculino";
7. usuario.idade = 45;
8. Pessoa usuario2 = new Pessoa();
9. usuario2.nome = "Maria Antônia";
10.usuario2.sexo = "Feminino";
11.usuario2.idade = 37;
12.Escreva(usuario.nome); // exibirá "João da Silva"
13.Escreva(usuario2.nome); // exibirá "Maria Antônia"
```

Perceba que estamos referenciando a mesma variável nas linhas 15 e 16 (nome), mas cada uma exibirá uma informação diferente. E como pode uma mesma variável conter dados diferentes? O grande segredo está no que chamamos de instância de um objeto, pois, embora a variável seja a mesma, a sua instância é outra, totalmente distinta, o que permite que dados diferentes sejam atrelados a uma mesma variável, mas isso será detalhado nos vídeos desse tema e explanado também no tema 2.

► 3. Considerações Finais

- Os paradigmas da programação ditam as formas como o desenvolvedor programará um *software*. O paradigma mais eficiente, do ponto de vista do desenvolvimento, é a POO.
- Toda classe é um molde que armazena a estrutura comum a todos os objetos que serão criados a partir dele. Dessa forma, todos os objetos terão as mesmas características e ações que a classe contiver, podendo ou não sofrer uma sobreposição de valores.

- Toda classe pode possuir métodos (ações) e atributos (características). Geralmente as características são adjetivos ou substantivos, enquanto os métodos são verbos.
- Utilizar POO no desenvolvimento de sistemas computacionais permite a reutilização de código, economia de tempo de desenvolvimento e segurança dos dados.

Glossário

- **Instanciar:** pode ser considerado como o ato de criar um novo objeto a partir de uma classe, ou seja, de um molde padrão. Todo objeto é uma nova criação a partir da Classe X, assim, todo objeto é uma instância da Classe X.

VERIFICAÇÃO DE LEITURA TEMA 1

1. Com relação a programação Orientada a Objetos, assinale a alternativa correta.
 - a) É um software de desenvolvimento, que agiliza a programação de sistemas.
 - b) Contribui para desenvolvimento ágil, por meio da linguagem Smalltalk.
 - c) É um paradigma concorrente do paradigma de programação estruturada, embora siga seu mesmo princípio básico de execução.
 - d) Não permite a utilização de classes, apenas objetos.
 - e) Tem métodos (características) e atributos (ações)

na estrutura de classes.

2. Assinale a alternativa correta. Os métodos e atributos:

- a) São, respectivamente, ações e características de um objeto.
- b) São, respectivamente, características e ações de uma classe.
- c) São, respectivamente, características e ações de um objeto.
- d) São, respectivamente, ações e características de uma classe.
- e) São, respectivamente, características e ações do paradigma procedural.

3. Com relação a POO, assinale a alternativa incorreta:

- a) POO permite reutilização de código e economia no tempo de desenvolvimento.
- b) O paradigma da POO existe há mais de cinquenta anos.
- c) Métodos podem receber parâmetros apenas do mesmo tipo.
- d) Métodos, em POO, podem receber parâmetros de tipos diferentes.
- e) Atributos ou métodos podem ser sobrepostos pelos objetos instanciados.

Referências Bibliográficas

AGUILAR, Luis Joyanes. **Programação em C++**: algoritmos, estruturas de dados e objetos. 2. ed .Porto Alegre: AMGH, 2011.

ASCENCIO, Ana Fernanda Gomes. **Fundamentos da programação de computadores:** algoritmos, PASCAL, C/C++, (padrão ANSI) e JAVA. 3. ed. São Paulo: Pearson Education do Brasil, 2012.

DEITEL, P. J. C: **como Programar.** 6. ed. São Paulo: Pearson Prentice Hall, 2011.

FELIX, Rafael. **Programação Orientada a Objetos.** São Paulo: Pearson Prentice Hall, 2016.

Gabarito – Tema 1

Questão 1 - Alternativa C.

O paradigma da POO é concorrente da programação estruturada. Embora o contexto básico de execução siga o mesmo preceito a ambas, a linha de pensamento de um sistema desenvolvido de forma estruturada é muito diferente da linha de um sistema desenvolvido em POO.

Questão 2 - Alternativa D.

Métodos são as ações (verbos) e atributos são características (adjetivos e, em alguns casos, substantivos). Os métodos e atributos não pertencem ao objeto, mas sim à classe, que é o molde principal responsável por formar objetos a partir de si.

Questão 3 - Alternativa C.

Os métodos podem receber parâmetros de tipos diferentes, como, por exemplo, Strings, Inteiros, Booleanos etc.



TEMA 02

OBJETO: ATRIBUTOS E MÉTODOS

► Objetivos

- Conhecer o conceito mais profundo de objetos.
- Identificar características de um objeto.
- Identificar ações de um método.
- Aplicar os conhecimentos adquiridos aos exercícios propostos.

Introdução

Agora que já temos uma ideia da utilidade do paradigma da Programação Orientada a Objetos, bem como a definição de um objeto, partiremos para os primeiros conceitos que envolvem esse assunto.

É necessário compreender a respeito do nível de abstração que utilizaremos aqui, pois, só com essa visão ampla, a compreensão dos tópicos e do curso como um todo será plena.

Veremos uma explanação detalhada sobre o que é um objeto dentro da programação e qual seu real vínculo com uma classe. Ao final dessa aula, você terá uma base conceitual sólida para ingressar sem receios nos temas que virão a seguir.

Saiba que, em alguns momentos, determinados tópicos que fazem parte de outros temas serão apresentados de maneira superficial apenas para formar a plenitude de uma ideia. É importante que você tenha o conhecimento dessa metodologia, pois em vários momentos falaremos de um mesmo assunto em unidades distintas de estudo e não estaremos sendo redundantes, apenas abordando em detalhes um tema citado, como complemento.

Seja sempre bem-vindo e bons estudos!

1. Atributos

Na computação, cada elemento que possa pertencer a um conjunto de elementos similares, pode ser considerado como um objeto e esses objetos são agrupados em uma classe (FÉLIX, 2016)¹. Como exemplo, temos um produto, num sistema do supermercado ABC. São vários produtos, cada qual com sua respectiva categoria, nome, destinação,

¹ FÉLIX, Rafael. **Programação Orientada a Objetos**. São Paulo: Pearson Prentice Hall, 2016.

manuseio. Seja um pacote de biscoitos, uma lata de ervilhas ou um detergente em pó, todos têm uma similaridade: são produtos.

Dessa maneira, é possível identificar um objeto como sendo um elemento dentre vários elementos pertencentes a uma mesma categoria, sendo representados de forma simplificada e padronizada, gerando um modelo que, por sua vez, pertence a uma categoria (SEPE e NETO, 2017)².

O importante, nesse exemplo, é internalizar dois conceitos: o pacote de biscoitos é o objeto; e esse objeto pertence à categoria produtos. De forma similar, a lata de ervilhas também é um objeto que pertence à categoria produtos. Essa categoria, como já vimos, é denominada classe. Cada item que deriva dessa classe é um objeto.

Ao analisarmos um pacote de biscoitos e uma lata de ervilhas, fica evidente as diferenças entre eles, mas se eles pertencem ao mesmo grupo (produtos), então alguma similaridade deve ter, conforme análise mostrada no quadro abaixo:

Quadro 1 | Análise comparativa entre objetos

PRODUTO	
Lata de ervilhas, contém:	Pacote de biscoitos, contém:
Embalagem.	Embalagem.
Data de validade.	Data de validade.
Peço.	Preço.
Marca.	Marca.
Ervilhas.	Biscoitos.

Fonte: elaborado pelo autor.

Esse quadro mostra várias similaridades entre dois elementos distintos de uma mesma categoria, que fazem com que sejam objetos pertencentes à mesma classe. Entretanto, quando tratamos a respeito do conteúdo do produto, surge uma diferença, que não é grandiosa o suficiente para

² SEPE, Adriano e NETO, Roque Maitino. **Programação Orientada a Objetos**. Londrina. Editora e Distribuidora Educacional S/A, 2017.

retirar o elemento da categoria produto, pois, embora o conteúdo seja diferente, ambos ainda são consumíveis, ou seja, há diferença entre eles, mas a essência ainda é mantida.

Essa explanação foi uma maneira sutil de chegarmos no tema principal, que é o **atributo** de uma classe. Sempre que lidarmos com as características que uma categoria tem, mesmo que seus objetos tenham características diferentes, como o caso da ervilha e dos biscoitos, nos referimos aos atributos, que nada mais são que variáveis que podem ou estão, previamente, atreladas a valores. Essas características também podem ser tidas como informações que compõem uma classe (ASCENCIO, 2012)³.

É possível inicializar atributos de uma classe, sem que eles possuam valores pré-estabelecidos, fazendo apenas a reserva de espaço na memória e, geralmente, é esse o tipo de inicialização que é realizada.

PARA SABER MAIS



Na computação, variáveis são elementos que podem receber valores de vários tipos (números inteiros, caracteres, números decimais etc.). Quando iniciamos uma variável e apontamos a qual tipo de dados ela pertence, um espaço equivalente é reservado na memória do computador, de forma que quando aquela variável venha a ser utilizada para receber um valor, esse possa ser devidamente armazenado no espaço previamente reservado.

Existem casos pontuais em que o programador inicializa variáveis já com valores atribuídos, mas isso é relativo e varia de acordo com o propósito da aplicação que esteja sendo desenvolvida.

³ ASCENCIO, Ana Fernanda Gomes. **Fundamentos da programação de computadores:** algoritmos, PASCAL, C/C++, (padrão ANSI) e JAVA. 3. ed. São Paulo: Pearson Education do Brasil, 2012.

Cada variável pode ter restrições de acesso, visualização e manipulação, que veremos adiante. Vamos trabalhar com a ideia de que o programador tenha desenvolvido a seguinte classe:

1. Classe Pública Produto
2. String nomeProduto = "Lata de Ervilhas";
3. Real precoProduto = 6.99;
4. Fim da Classe

PARA SABER MAIS



Existem algumas convenções adotadas no mundo da programação de sistemas, que são padrões de desenvolvimento adotados, de forma a gerar uma estrutura igualitária de sistemas por todo o mundo.

Na convenção de Programação Orientada a Objetos, recomenda-se que o nome de classes seja sempre iniciado com letra maiúscula, seguindo a grafia CamelCase, ou seja, a letra inicial de cada nova palavra deverá também ser maiúscula, por exemplo: Classe ProdutosDeInformatica.

Não há problema algum em criar a classe da maneira como está descrito acima. Entretanto, se ela é um molde (uma estrutura da qual derivarão vários produtos), torna-se um pouco desconexo atribuir valores padrões às suas variáveis, considerando que a cada novo objeto (produto) criado, novos valores serão atribuídos. Dessa maneira, recomendamos apenas criar a classe, declarar suas variáveis e deixá-las sem valor algum.

Compreenda que essa pode ser considerada a regra padrão, mas existe a exceção para casos específicos. Novamente, ressaltamos que o uso do

bom senso guiará sobre o que é mais recomendado a fazer dependendo de cada situação.

LINK



Acesse esse conteúdo da Universidade Federal do Mato Grosso do Sul, que trata a respeito de Programação Orientada a Objetos. Recomendamos que seja vista, apenas, a seção de atributos e métodos. Disponível em: <http://www.facom.ufms.br/~lianaduenha/sites/default/files/fundamentos_0.pdf>. Acesso em: 09 ago. 2018.

► 2. Métodos

Se os atributos tratam a respeito das características de uma classe, os métodos referem-se às suas ações ou funcionalidades. Voltemos aos exemplos anteriores, utilizando a classe Produto. Nessa classe, a partir da qual derivamos dois objetos (lata de ervilhas e pacote de biscoitos), em que cada um—**ervilhas** / **biscoitos**—pode ter um conjunto de ações atreladas. Pense nessas ações como sendo verbos, que, na Língua Portuguesa, permitam ser conjugados nas pessoas do plural e singular.

ASSIMILE



A lata de ervilhas, que tem um preço definido, pode sofrer alteração em seu valor, aumentando ou diminuindo seu preço de venda, logo, duas das ações atreladas ao produto ervilha poderiam ser `aumentarPreco` e `diminuirPreco`.

Observe que ambas as ações são verbos (`aumentar` / `diminuir`), que podem ser conjugados no singular ou plural: eu aumento,

tu aumentas, ele/ ela aumenta; nós diminuímos, vós diminuís, eles/ elas diminuem.

Essa noção é importante para que você não confunda ao criar métodos e atributos, atribuindo, erroneamente, as características de um ao outro. A palavra `nomeProduto` também compõe a classe `Produto`, mas não é um verbo.

Os métodos, comportamentos de uma classe, geralmente, têm a função de modificar os valores dos atributos de uma classe ou realizar passagem de mensagens (SINTES, 2002)⁴, embora não seja o papel fundamental deles, que podem executar ações diversas no escopo da classe. Entretanto, para esse momento, focaremos em seu papel principal que é alterar os valores dos atributos.

A nossa classe `Produto` poderá ser acrescida dos métodos `aumentarPreco` e `diminuirPreco`, ficando com a seguinte estrutura:

1. Classe Pública `Produto`
2. `String nomeProduto;`
3. `Real precoProduto;`
4. `AumentarPreco(aumenta)`
5. `Preco = preço + aumenta`
6. `DiminuirPreco(diminui)`
7. `Preco = preco - diminui`
8. Fim da Classe

Analise a linha 4 e veja que utilizamos o método `aumentarPreco`, seguido de dois parênteses que armazenam um valor chamado `aumenta`. A

⁴ SINTES, Tony. **Aprenda Programação Orientada a Objetos em 21 dias**. São Paulo, Pearson Education do Brasil, 2002.

esse valor, damos o nome de parâmetro, que são, em geral, variáveis recebidas pelos métodos e que vêm com algum valor atrelado a elas. É evidente que em algum local ou outra seção do *software*, existiu um momento em que o acréscimo de preço foi atribuído à variável *umenta*, fazendo com que ela, na linha 4 do código acima, contivesse o valor do aumento desejado.

EXEMPLIFICANDO

No sistema do Supermercado ABC, o desenvolvedor criou uma tela de edição de produtos, com um campo para modificar o preço, aumentando ou diminuindo em percentual. Quando a pessoa responsável pela edição do preço digitar o percentual, esse valor será armazenado na variável *umenta*, que será convertida em parâmetro a ser recebido pelo método *aumentarPreco*, da classe *Produto*.

Quando o método for executado, ele receberá o valor modificado (armazenado na variável *umenta*) e executará as operações da linha 5:

Preço receberá: preço atual + valor aumentado.

A mesma lógica pode ser aplicada ao conceito do método *diminuirPreco*.

2.1 Métodos construtores

São métodos especiais, invocados apenas no momento de criação dos objetos. Esses métodos ficam dentro do escopo da classe, mas são invocados e criados assim que os objetos são instanciados., que tem o objetivo de garantir que o objeto seja inicializado, garantindo sua consistência,

embora também a personalização do objeto seja uma das funções dos métodos construtores (SEPE; NETO, 2017)⁵.

Por padrão, os métodos construtores devem ter o mesmo nome da classe e o retorno de suas ações deve ser omitido. Essa omissão acontece por meio de um `return void` (vazio) ao final da estrutura do método.

Vejamos como ficaria nossa classe `Produto` com a inserção do método construtor:

1. Classe Pública `Produto`
2. `String nomeProduto;`
3. `Real precoProduto;`
4. `// método construtor`
5. `Produto()`
6. `aumentarPreco = 0;`
7. `Return void;`
8. `AumentarPreco(aumenta)`
9. `Preco = preço + aumenta`
10. `DiminuirPreco(diminui)`
11. `Preco = preco-diminui`
12. Fim da Classe

Na linha 5, indicamos o início do método construtor, com o mesmo nome da classe (`Produto`); na linha 6, inicializamos o valor do método. Por fim, na linha 7, inserimos um valor de retorno como vazio.

Algumas linguagens, como o PHP, requerem a inserção de métodos destrutores, que têm a função de eliminar os métodos quando não estiverem

⁵ SEPE, Adriano e NETO, Roque Maitino. **Programação Orientada a Objetos**. Londrina. Editora e Distribuidora Educacional S/A, 2017.

mais em uso, desobstruindo espaço na memória, mas, geralmente, a função de destruir métodos já ocorre de forma natural e automática na maioria das linguagens.

Quando tratarmos a respeito de instâncias, veremos como alimentar os objetos com dados.

2.2 Diagrama de Classes

Uma forma prática de enxergar essa estrutura de métodos e atributos é por meio dos diagramas de classe, que são simples e compostos por uma tabela com três divisões: na primeira, inserimos o nome da classe; na segunda, os atributos; e na terceira, os métodos.

Esse diagrama não está completo, pois, para fins didáticos, algumas informações quanto ao encapsulamento de dados foram propositalmente omitidas para serem apresentadas mais adiante.

Quadro 2 | Diagrama de classes

PRODUTO
String nomeProduto Real precoProduto
aumentarPreco() diminuirPreco()

Fonte: elaborado pelo autor (2018).

O conceito é bastante simplório, mas auxilia na visualização do que se pretende realizar com relação a código, pois explicita os pontos mais importantes de uma classe: nome, métodos e atributos, delimitados cada um em seu próprio espaço. Um outro artifício, implementado nos diagramas de classe, são os critérios de encapsulamento, que veremos na próxima unidade.

QUESTÃO PARA REFLEXÃO

Vimos um exemplo de criação de uma classe, para armazenagem de produtos, composta por métodos e atributos que armazenavam e manipulavam as informações de acordo com o objeto. É recomendado que você não se prenda apenas aos exemplos apresentados nas aulas, mas expanda tais informações. Tente, inclusive, realizar testes próprios, criando classes inéditas com as configurações que você desejar. Uma recomendação inicial seria criar uma classe para verificação de saldos bancários, na qual existem muitas possibilidades a serem exploradas para colocar em prática todo conhecimento adquirido até aqui.

► 3. Considerações Finais

- Atributos são as características de um objeto.
- Métodos são as ações ou funções que um objeto pode ter ou realizar.
- Métodos construtores servem para inicializar os objetos, focando na consistência dos mesmos, sendo ainda responsáveis pela personalização inicial.
- Diagramas de classes permitem exibir as informações de forma visual e, geralmente, são realizados antes da implementação em código.

► Glossário

- **CamelCase:** escrever nomes de variáveis, métodos, atributos, classes, constantes, utilizando a alternância entre maiúsculas e minúsculas, sempre no início de cada palavra, facilitando a leitura.



VERIFICAÇÃO DE LEITURA

TEMA 2

1. Assinale a alternativa que indica a correta definição para Atributos:
 - a) Formas de instanciar objetos.
 - b) Características de uma classe.
 - c) Ações de um objeto.
 - d) Sinônimos para métodos construtores.
 - e) Representações gráficas de uma classe.
2. Métodos construtores têm como uma das funções:
 - a) Inicializar objetos, tornando-os consistentes.
 - b) Construir a estrutura de uma classe.
 - c) Prover o paradigma da Programação Orientada a Objetos.
 - d) Atribuir valores aos parâmetros de um método.
 - e) Inicializar variáveis.
3. Assinale a alternativa que indica a correta definição para “parâmetros”:
 - a) São valores recebidos por atributos.
 - b) São valores informados pelo programador.
 - c) São valores utilizados nos métodos destrutores.
 - d) São valores vindos de variáveis.
 - e) São valores que modificam os tipos de dados dos atributos.

Referências Bibliográficas

ASCENCIO, Ana Fernanda Gomes. **Fundamentos da programação de computadores:** algoritmos, PASCAL, C/C++, (padrão ANSI) e JAVA. 3. ed. São Paulo: Pearson Education do Brasil, 2012.

FELIX, Rafael. **Programação Orientada a Objetos.** São Paulo: Pearson Prentice Hall, 2016.

SEPE, Adriano e NETO, Roque Maitino. **Programação Orientada a Objetos.** Londrina. Editora e Distribuidora Educacional S/A, 2017.

SINTES, Tony. **Aprenda Programação Orientada a Objetos em 21 dias.** São Paulo, Pearson Education do Brasil, 2002.

Gabarito – Tema 2

Questão 1 - Alternativa B.

Os atributos são substantivos ou adjetivos, responsáveis por caracterizar as classes e, por consequência, seus objetos atrelados.

Questão 2 - Alternativa A.

Os métodos construtores, como o próprio nome induz, são responsáveis por construir ou inicializar os objetos, tornando-os mais consistentes dentro do código. É possível ainda utilizar os construtores para personalizar a inicialização desses objetos.

Questão 3 - Alternativa D.

Os parâmetros são recebidos pelos métodos por meio de variáveis que, em algum momento anterior, foram alimentadas com algum tipo de dado ou informação.



TEMA 03

CLASSES E INSTÂNCIA DE OBJETOS

► Objetivos

- Assimilar o conceito de classes.
- Compreender a formação de uma classe.
- Internalizar o funcionamento de instanciação de objetos.
- Entender os processos de instanciação.

Introdução

Nas aulas anteriores, vimos sobre o conceito de classes e sua função dentro do paradigma da Programação Orientada a Objetos (POO). Entendemos sua formação e vimos, no tema 1, alguns exemplos de código utilizando classes. Nessa aula, detalharemos os tópicos que norteiam o paradigma da POO, entendendo como um compilador faz a interpretação desses artifícios no momento de executá-los mediante invocação.

Paralelamente, veremos também o funcionamento de instâncias, compreendendo o que é, para que serve e como realizar tal ação na programação.

É premissa que você tenha revisado o material de leitura fundamental anterior, acompanhado as videoaulas referentes a esse material para que você tenha uma sólida base de conhecimento dos temas que antecedem os assuntos a serem abordados aqui.

A partir dessa aula, utilizaremos a linguagem de programação Java para alguns exemplos, concomitante aos pseudocódigos que temos utilizado até então. O domínio da linguagem não será pré-requisito, pois deixaremos o código o mais enxuto possível, afinal, o objetivo não é um aprofundamento numa linguagem de programação específica, mas sim o conhecimento amplo sobre o paradigma da POO aplicável a qualquer linguagem de programação.

1. Estrutura de uma classe

Como já vimos, as classes são moldes que armazenam características e ações de um conjunto de elementos em comum. Paralelo a essas classes, temos também o conceito denominado por superclasses abstratas, que, de acordo com Rafael Felix (2016), embora não tenham uma função ativa

na estrutura do código, “ajudam a evitar duplicação de código ao abrigar uma série de subclasses como suas herdeiras” (FELIX, 2016, p. 45)¹.

Esse conceito pode, num primeiro momento, parecer um tanto superficial, mas ao estudarmos o tema 6, veremos os detalhes específicos desse assunto, pois trataremos sobre herança. Para fins didáticos, elucidaremos rapidamente esse conceito aqui, de forma que a compreensão das classes abstratas seja mais ampla e eficiente.

Via de regra, na POO, o conceito de herança tem o mesmo significado que no mundo real, ou seja, “herança é um conjunto de bens e direitos, ativos e passivos cedidos a outrem”, conforme especificação do dicionário Michaelis (2018)². Trazendo para o mundo computacional, a herança caracteriza-se pela ação de um elemento **A ceder** características a outros elementos como B, C, D etc.

ASSIMILE




Imagine uma herança genética, utilizando o seguinte exemplo: um pai, de olhos verdes e cabelos castanhos, passará por meio do DNA essas mesmas características aos seus filhos. Isso significa dizer que a maioria dos descendentes daquele mesmo pai têm a tendência a ter cabelos castanhos e olhos verdes.

Dessa forma, o conceito de herança também tem essa mesma aplicabilidade em Orientação a Objetos, onde uma classe (pai) detém determinadas características (atributos), que são repassadas às suas filhas (subclasses).

É importante lembrar que nem toda classe pai é uma classe abstrata, mas toda classe abstrata é uma classe pai (caso contrário não teria sentido de

¹ FELIX, Rafael. **Programação Orientada a Objetos**. São Paulo: Pearson Education do Brasil Ltda. 2016.

² DICIONÁRIO Michaelis da Língua Portuguesa. Disponível em: <<https://michaelis.uol.com.br/moderno-portugues/busca/portugues-brasileiro/heran%C3%A7a/>>. Acesso em: 27 ago. 2018.



ser criada). As classes abstratas têm como função o armazenamento de atributos, que são comuns a todas as classes filhas (subclasses). Assim, essas classes pais (abstratas) não são invocadas diretamente, servem apenas de molde para suas herdeiras.

Usemos como exemplo a classe Pessoa, dentro de um sistema de cadastro.

Esse mesmo sistema tem várias áreas de cadastro:

- Cadastro de fornecedores.
- Cadastro de clientes.
- Cadastro de funcionários.
- Cadastro de sócios.

Sabemos que todos esses cadastros são de uma mesma categoria, pois todos os elementos que serão cadastrados têm algo em comum: são pessoas. Entretanto, intuitivamente, sabemos que o cadastro de um funcionário se difere em vários aspectos do cadastro de um cliente, da mesma maneira que o cadastro de um fornecedor tem vários pontos diferentes do cadastro de um sócio.

O que fazer nessa situação em que todos os tipos de cadastro são similares e ao mesmo tempo são diversificados? Como atrelá-los a uma mesma classe, com tantas características distintas?

É nesse ponto em que recorreremos ao uso das classes abstratas, que serão responsáveis por dar, de herança, apenas as características comuns a todas as suas classes filhas. Todos os tipos de cadastro (fornecedores, clientes, funcionários e sócios) são classes, mas possuem acima de si, uma classe pai, que, nesse exemplo, é a classe Pessoa.

Embora as classes criadas para os cadastros utilizem as características herdadas da classe pai, ela (a classe Pessoa), não é chamada de forma direta, ou seja, sua única função é ceder as características às classes filhas. Esse é o principal papel de uma classe abstrata: prover os moldes comuns a N classes filhas. Com base nessa informação, podemos afirmar

corretamente que uma classe abstrata não tem razões plausíveis para ser utilizada de forma direta, ou seja, não é uma boa prática criar-se um objeto que derive a partir de uma classe abstrata.

Veja o exemplo: temos uma classe abstrata chamada *Pessoa*, que tem como única função gerar um escopo que será herdado por suas classes filhas (subclasses) *Funcionario* e *Cliente*. Ao realizarmos a instanciação, não chegamos a referenciar a classe *Pessoa* diretamente, referenciamos apenas a classe *Funcionario* ou a classe *Cliente*, pois essas já estão com acesso aos métodos e atributos da classe abstrata *Pessoa*.

PARA SABER MAIS



O conceito de classe abstrata é mais conceitual do que prático ou aplicável. Você poderá ver, claramente, que uma classe abstrata em nada se difere de uma classe convencional, senão pelo fato de ela nunca servir de molde direto para um objeto. A prática de instanciar objetos a partir de uma classe abstrata não é considerada uma boa prática de programação, apesar de não resultar em erros de compilação ou execução.

► 2. Subclasses

Como já explanado anteriormente, as subclasses são derivadas de uma classe principal que pode, ou não, ser uma classe abstrata. A essa derivação, damos o nome de herança (mais detalhes serão abordados no tema 6). As subclasses têm como característica ter os mesmos métodos e atributos de sua classe pai, além de seu próprio conjunto de ações e características. Vejamos um exemplo, utilizando a classe *Pessoa* e sua (s) herdeira (s):

Classe Pessoa

1. Classe pública Pessoa{
2. String nome;
3. int idade;
4. String cpf;
5. String enderecoCompleto;
6. }

Classe Funcionário

1. Classe pública Funcionario herdeira de Pessoa {
2. String setor;
3. Double salario;
4. }

Ao analisarmos a linha 1, da classe *Funcionario*, podemos observar algo novo na estrutura: a menção de herança de uma outra classe. Utilizando essa estrutura, a classe funcionário passa a ter a somatória de atributos da classe pessoa e de si própria, ou seja, podemos afirmar que a classe *Funcionario* é composta dos seguintes atributos:

- String nome (**atributo herdado**).
- int idade (**atributo herdado**).
- String cpf (**atributo herdado**).
- String enderecoCompleto (**atributo herdado**).
- String setor.
- Double salario.

Essa mesma estrutura na linguagem Java, teria a seguinte sintaxe:

Classe Pessoa

1. Public class Pessoa{
2. String nome;
3. Int idade;
4. String cpf;
5. String enderecoCompleto;
6. }

Classe Funcionário

1. Public class Funcionario **extends** Pessoa {
2. String setor;
3. Double salario;
4. }

O código Java é bastante similar ao pseudocódigo utilizado até então. Veja que na linha 1, da classe funcionário, adicionamos a palavra reservada **extends**, responsável por tornar aquela classe uma herdeira da classe Pessoa.

PARA SABER MAIS



Cada linguagem de programação tem suas particularidades. Utilizar pseudocódigo é a maneira mais genérica possível de se explicar conceitos que podem ser aplicados a qualquer linguagem. Para que você fixe o conteúdo com eficiência, é recomendado que, paralelamente ao uso de pseudocódigo, você opte por adotar uma linguagem de programação para executar seus testes. A recomendação é que seja adotado o Java, que é a linguagem que utilizaremos, mas você pode optar por alguma outra linguagem de sua escolha, que siga os padrões do paradigma da POO.

► 3. Instanciando objetos

Já vimos, anteriormente, um conceito superficial de instanciação de objetos. Nesse tópico, aprofundaremos a respeito para que você possa compreender o que é instanciar e qual sua função dentro da programação orientada a objetos.

3.1 O que é instanciar?

Em termos bastante práticos e diretos, instanciar é **criar** um objeto que até então ainda não existia dentro do código. Para isso, no momento da instanciação, utilizamos a palavra reservada *new*, que “cria um novo objeto da classe especificada à direita da palavra-chave” (DEITEL, 2010, p. 60)³.

Com uma classe à disposição, podemos criar quantos objetos desejarmos usando-a como molde. O ato de criarmos um novo objeto é justamente o ato de instanciá-lo. Dessa maneira, a cada novo objeto criado, tem-se a instância de uma classe.

Para que essa instância (ou criação) ocorra, é necessário:

- Ter uma classe criada.
- Um nome para o objeto em questão.

A criação do objeto acontece a partir do uso da palavra reservada *new*. Considerando que temos uma classe pai (abstrata ou não), chamada Pessoa e ela tenha subclasses (herdeiras ou filhas), que herdem de si algumas características (atributos) ou ações (métodos), já teríamos uma estrutura formada e pronta para ser utilizada por qualquer objeto. Entretanto, devemos lembrar que ainda não foi criado nenhum objeto dentro desse escopo, e é necessário informarmos ao código que um novo objeto usará os recursos vindos de uma classe (seja pai ou filha).

³ DEITEL, Harvey M. **Java Como Programar**. São Paulo: Pearson Prentice Hall, 2010.

Esse aviso serve para que exista consistência de informações e apenas objetos, previamente autorizados, utilizem os recursos disponibilizados pela (s) classe (s). É uma questão de boa prática de desenvolvimento adotada e, que se não for executada dessa maneira, resultará em erro de compilação.

EXEMPLIFICANDO

Considere o cenário de um supermercado, em que existem várias categorias, como cereais, carnes, verduras, legumes, pães. Vamos escolher a categoria pães para trabalhar. Arelados à categoria pães, estão vários produtos, como: pão integral, pão francês, pão de forma, pão sovado, pão de leite.

Cada um desses produtos, sabemos que têm suas próprias especificações, porém, todos eles têm características em comum. Todos contém:

- Farinha de trigo.
- Ovos em seu preparo.
- Fermento.

Assim, seria redundante que, a cada tipo de pão criado precisássemos escrever esses três ingredientes; estaríamos repetindo uma mesma característica que todos os pães possuem.

PARA SABER MAIS

A maneira mais eficiente de resolver essa redundância seria derivar todos os pães de uma mesma categoria principal, que armazenasse os ingredientes em comum, e, para cada pão especificamente, apenas adicionaríamos as suas próprias características. Suponhamos então, que temos a categoria principal chamada Pães e as subcategorias pão integral, pão francês, pão de forma, pão sovado, pão de leite.

Quando um objeto fosse instanciado a partir de uma classe, essa classe deveria ser alguma das subcategorias que contém, além das informações comuns da categoria principal, as características específicas daquela subcategoria, dando completeza ao objeto, que utilizará parte das características (atributos) da subcategoria e parte da categoria principal.

Vamos analisar a seguir como ficaria esta estrutura, utilizando ainda, como base, a classe Funcionario, que é herdeira (subclasse) de Pessoa:

```
Funcionario func = new Funcionario();
```

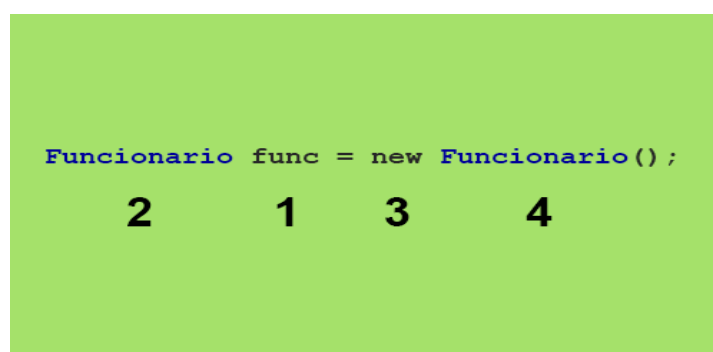
Figura 1 | Instanciação de classes.



Fonte: elaborado pelo autor (2018).

A maneira mais prática de realizar a leitura dessa linha de código seria começando a ler a partir do nome do objeto:

Figura 2 | Leitura da linha de instanciação



Fonte: elaborado pelo autor (2018).

Dessa forma, seguindo a sequência numérica, a leitura ficaria assim:

O Objeto func, que é do tipo Funcionario, será um novo objeto da classe Funcionario.

O texto parece redundante, mas é exatamente assim que o compilador espera encontrar o seu código para executá-lo com sucesso e criar uma nova instância de uma determinada classe. Atente aos parênteses ao final do código. Eles devem ser inseridos no seu código, mas permanecerem vazios (ao menos num primeiro momento). Mais adiante, quando tratarmos a respeito de métodos construtores, veremos que é possível inserir alguns parâmetros entre esses parênteses, no momento da instanciação dos objetos; mas isso é tema a ser tratado no capítulo 5.

LINK



No canal da Universidade Virtual de São Paulo (UNIVESP), é possível visualizar alguns exemplos de classes e objetos. Recomendamos a visualização das Aulas 0, 1 e 2. Disponível em: <<https://www.youtube.com/watch?v=IlgAMHgODUA>>. Acesso em: 22 ago. 2018.

► 3. Considerações Finais

- As classes podem ser abstratas ou não.
- É possível gerar classes herdeiras (chamadas também de classes filhas ou subclasses), a partir de classes principais chamadas de classes-pai.
- A instanciação de uma classe ocorre quando criamos um objeto no código. Esse objeto tem a capacidade de acessar os métodos e

QUESTÃO PARA REFLEXÃO

Vimos as possibilidades que a POO oferece ao desenvolvedor, por meio do uso de classes, classes abstratas, subclasses e objetos, com alguns exemplos. Propomos a você o desafio de elaborar uma estrutura, utilizando os conhecimentos vistos até então, para formar a arquitetura de cargos em uma empresa, tomando como base as seguintes informações:

- A empresa ABC tem vários cargos.
- Cada cargo pertence a um setor.
- Existem várias pessoas que atuam nos cargos e, por consequência, estão alocadas no setor correspondente.

Elabore as classes, e quantos objetos forem necessários, para treinar suas habilidades e pôr à prova seus conhecimentos.

Embora o contexto pareça assustador logo de início, basta você aplicar a mesma linha de raciocínio dos exemplos que vimos durante essa aula e, com certeza, chegará a uma estrutura adequada para atender à situação.

atributos da classe à qual pertence, bem como à classe pai, caso ele pertença a uma subclasse.

► Glossário

- **Compilador:** é o interpretador, que converte a linguagem de programação para linguagem de máquina e vice-versa.
- **Instanciação:** criação ou vinculação de um objeto a uma determinada classe.

- **Palavras reservadas:** são palavras nativas de determinadas linguagens de programação ou compiladores e não podem ser usadas, senão com o propósito específico para o qual foram criadas.

VERIFICAÇÃO DE LEITURA

TEMA 3



1. Assinale a alternativa que apresenta as características das subclasses:
 - a) Herdarem informações apenas das classes abstratas.
 - b) Herdarem métodos ou atributos de objetos, por meio da palavra reservada new.
 - c) Cederem dados às classes herdeiras, como, por exemplo, um conjunto de métodos.
 - d) Herdarem métodos ou atributos das classes pai, sejam elas abstratas ou não.
 - e) Cederem dados às classes abstratas, como, por exemplo, um conjunto de atributos.
2. Com relação às classes abstratas, assinale a alternativa correta:
 - a) Têm como função abastecer subclasses que tenham características ou ações em comum e, por questões de boas práticas, não são diretamente instanciadas por objetos.
 - b) Têm como função serem instanciadas diretamente por objetos, mas por questões de boas práticas de programação, não devem ceder atributos às classes filhas.
 - c) Têm como função secundária, serem instanciadas diretamente por objetos, mas por questões de boas práticas de programação, não devem ceder métodos às classes filhas.

- d) Têm como função abastecer classes pais que tenham métodos ou atributos similares e, por questões de boas práticas, não são diretamente instanciadas por objetos.
- e) Têm como função ceder métodos ou atributos aos objetos e às subclasses atreladas; por questões de boas práticas e convenção, os nomes dessas classes devem respeitar o padrão CamelCase em seus nomes.

3. Quanto ao objeto instanciado, é correto se afirmar que:

- a) Ele só tem acesso aos métodos e atributos da classe instanciada que seja herdeira de uma classe pai, porém, os métodos e atributos da classe pai não lhe são acessíveis.
- b) Ele só pode acessar os métodos e atributos de uma classe pai, se essa for abstrata.
- c) Ele tem acesso aos métodos e atributos de uma classe pai, mesmo que seja uma instância de uma classe filha herdeira.
- d) Ele tem pleno acesso aos atributos da classe pai e apenas aos métodos da classe filha (subclasse).
- e) Ele pode instanciar diretamente a classe pai e, assim, obter os métodos e atributos de uma classe herdeira.

Referências Bibliográficas

FELIX, Rafael. **Programação Orientada a Objetos**. São Paulo: Pearson Education do Brasil Ltda. 2016.

DEITEL, Harvey M. **Java Como Programar**. São Paulo: Pearson Prentice Hall, 2010.

MICHAELIS, Dicionário Brasileiro da Língua Portuguesa. Disponível em <<http://michaelis.uol.com.br/busca?id=ZNQp2>>. Acesso em: 10 nov. 2018.



Gabarito – Tema 3

Questão 1 - Alternativa D

As subclasses também são chamadas de classes filhas e podem herdar tanto métodos quanto atributos de suas classes pais, sejam elas abstratas ou não.

Questão 2 - Alternativa A

Conforme vimos, as classes abstratas não devem, embora possam, ser instanciadas diretamente pelos objetos por questões de boas práticas. Devem servir de base para alimentar subclasses (classes filhas) que tenham atributos (características) ou métodos (ações) em comum.

Questão 3 - Alternativa C

Quando se instancia uma classe filha, os objetos criados conseguem acessar todos os métodos e atributos da classe em questão e da classe pai, pois, o ato de herança transfere toda a estrutura da classe pai à filha.



TEMA 04

ENCAPSULAMENTO

► Objetivos

- Compreender o conceito de encapsulamento no paradigma da programação orientada a objetos.
- Diferenciar os tipos de encapsulamento existentes.
- Discernir entre os tipos mais adequados a serem utilizados em cada situação.
- Solidificar a importância do encapsulamento de classes, métodos e atributos.

► Introdução

No contexto da Programação Orientada a Objetos, um dos conceitos mais importantes é o encapsulamento aplicado às classes e aos seus métodos e atributos (FÉLIX, 2016)¹. Estudaremos, nessa unidade, os tipos de encapsulamento disponíveis, bem como sua implementação e importância para o sucesso de um projeto.

Desenvolver sistemas baseados no paradigma da Orientação a Objetos vai bem mais além do mero reaproveitamento de códigos por meio de classes ou heranças, como vimos no capítulo anterior e que será explicado de forma ampla no capítulo 6. Esse paradigma preza pela proteção dos dados que trafegam entre as classes e seus objetos, visando a confiabilidade da informação que esteja sendo manipulada por um determinado sistema, já que o acesso ao código teria uma camada de restrições de acesso (FÉLIX, 2016).

Ao tratarmos sobre proteção dos dados por meio de restrições de acesso, referimo-nos ao acesso que o próprio código poderia fazer em outras áreas de sua própria estrutura. É fundamental deixar esse ponto elucidado para que não haja divergências conceituais quanto a *quem* realiza tais acessos. A restrição não foca tornar uma classe, um método ou um atributo oculto ou visível aos programadores ou pessoas envolvidas no desenvolvimento do código; a restrição direciona suas regras às outras classes e outros métodos, dentro do mesmo sistema.

Apesar de parecer um tanto improvável, é comum em projetos grandes ou com muitos desenvolvedores envolvidos, existir alguns conflitos por uso de um mesmo nome para classes, variáveis ou objetos. Esses conflitos poderiam gerar, por exemplo, nomes de métodos idênticos, mas pertencentes a classes distintas. O encapsulamento garantiria que apenas o conjunto de métodos autorizados pudessem manipular os dados e

¹ FÉLIX, Rafael. **Programação Orientada a Objetos**. São Paulo: Pearson Education do Brasil Ltda., 2016.

atributos de uma determinada classe, evitando erros catastróficos originados por um acesso não esperado.

No decorrer dessa unidade, você verá exemplos de como o uso do encapsulamento é capaz de prover maior segurança aos sistemas, transmitindo maior confiabilidade à arquitetura do *software*.

► 1. Tipos de encapsulamento

Os tipos de encapsulamento em POO (Programação Orientada a Objetos) definem os níveis de acesso que uma estrutura pode ter, seja para escrever ou ler informações. Esses níveis de acesso são divididos em acesso público, acesso privado, acesso protegido e acesso padrão (*default*).

Se compararmos o paradigma procedural e o paradigma orientado a objetos, veremos que eles têm muita semelhança em seus aspectos funcionais. Um dos elementos que os distingue drasticamente é, justamente, o encapsulamento por meio dos modificadores de acesso, que podem ser atribuídos às classes, métodos ou atributos.

1.1 Criando um projeto em Java para aplicação de conceitos

Criaremos um projeto em Java, utilizando o IDE Netbeans, que pode ser baixado gratuitamente no seguinte link disponível em: <<https://netbeans.org/downloads/>>. Acesso em: 12 dez. 2018.

Escolha a versão mais adequada ao seu Sistema Operacional, bem como as linguagens com as quais você deseja trabalhar dentro do IDE. Recomendamos, nesse curso, que você foque apenas na linguagem Java, pois é a linguagem adotada nos exemplos, assim você terá mais facilidade em compreender e assimilar conceitos. Entretanto, os mesmos conceitos

ora transmitidos poderão ser aplicados a qualquer outra linguagem orientada a objetos, com pequenas adaptações de sintaxe.

Para que o Java funcione em sua máquina, é necessário um pacote de aplicações complementares, como uma máquina virtual, *Java Virtual Machine (JVM)* e o *Java Runtime Environment (JRE)*. Esse pacote de aplicações adicionais pode ser obtido diretamente no site da Oracle, detentora e mantenedora atual do Java.

Para que não haja a necessidade de instalar cada componente de forma isolada, você pode optar por fazer o *download* do *Java Development Kit (JDK)*, kit de desenvolvimento Java, que engloba todas as aplicações necessárias para que você execute códigos Java em seu computador. É possível obter o JDK, escolhendo a versão apropriada para seu Sistema Operacional, por meio do link disponível em: <<http://www.oracle.com/technetwork/pt/java/javase/downloads/jdk8-downloads-2133151.html>>. Acesso em: 12 dez. 2018.

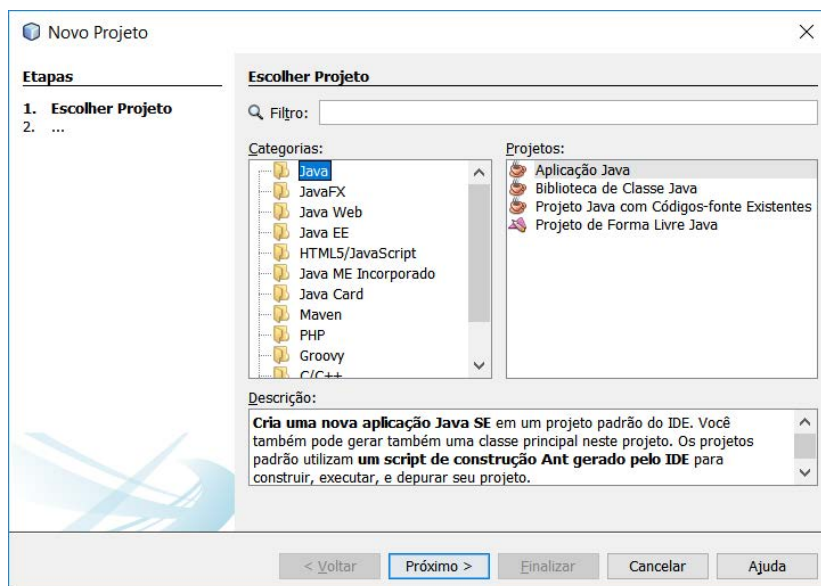
Recomendamos que, primeiramente, você instale o pacote JDK antes de instalar o Netbeans, que irá requerer as aplicações Java previamente instaladas em seu computador para que possa prosseguir com sua instalação.

Após ter instalado o JDK e o *Netbeans* em seu computador, siga os passos a seguir para criar o primeiro projeto Java, a fim de testar, na prática, os conceitos de encapsulamento que veremos mais adiante, bem como os demais testes que serão realizados de agora em diante.

1.1.1 Primeiros passos

Com o programa *Netbeans* aberto, clique sobre o menu *Arquivo > Novo Projeto*. A seguinte tela será exibida para você. Marque as opções *Java*, na aba esquerda; e *Aplicação Java*, na aba direita. Clique em *Próximo*.

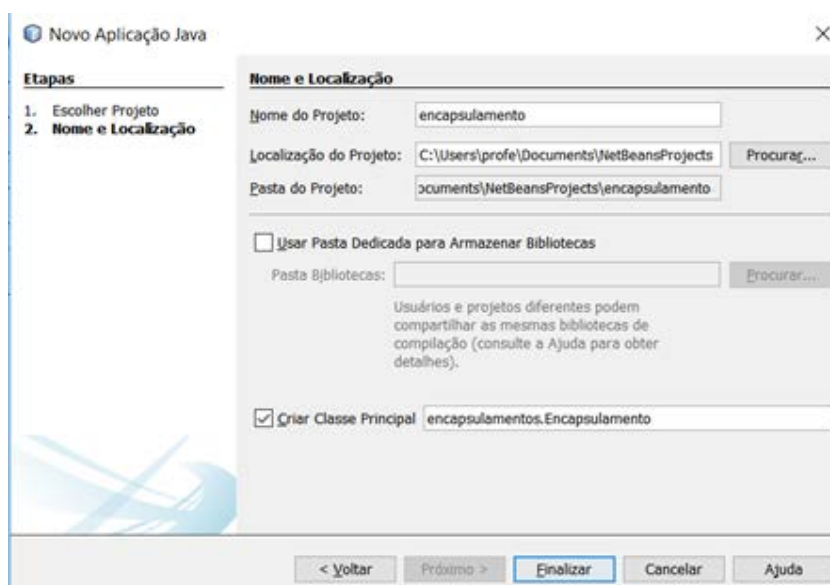
Figura 1 | Tela de criação de projeto, primeira parte



Fonte: adaptado pelo autor, do programa *Netbeans* (2018).

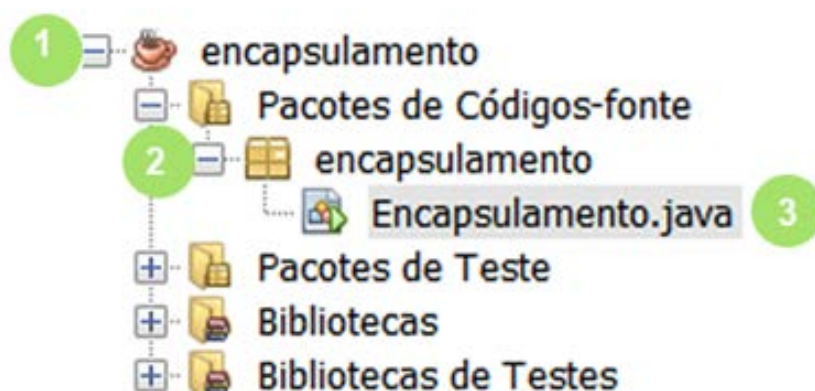
Você deverá apontar o nome do projeto, que será o nome do seu sistema. Para fins didáticos, o nome desse nosso projeto inicial será *encapsulamento*. De maneira automática, a primeira classe de seu projeto já será criada, com o mesmo nome do projeto. Por questões de convenção, todos os nomes de classes devem iniciar em maiúsculo. Clique em *Finalizar*.

Figura 2 | Tela de criação de projeto, segunda parte



Fonte: adaptado pelo autor, do programa *Netbeans* (2018).

O projeto será criado e terá a seguinte estrutura, na lateral esquerda:



Fonte: adaptado pelo autor, do programa *Netbeans* (2018).

Legendas:

- 1: o nome do projeto.
- 2: um pacote com o mesmo nome do projeto.
- 3: classe principal, com a inicial em maiúsculo.

PARA SABER MAIS

Os pacotes (packages) são como pastas, que servem para organizar seu projeto, juntando classes que tenham relação umas com as outras. Por exemplo: pense num sistema de gestão empresarial que tem algumas seções de acordo com os departamentos da empresa:

- Contas
 - A pagar.
 - A receber.
- Estoque

- Entrada de estoque.
- Baixa de estoque.
- Inventário.
- Vendas

É evidente que os assuntos relacionados estão agrupados dentro de uma mesma categoria. Essa mesma lógica deve ser mantida para a estrutura de classes e pacotes: aloque apenas as classes que têm relação entre si num mesmo pacote. Isso será útil tanto para a manutenção do sistema quanto para um determinado nível de encapsulamento que veremos mais adiante.

No console principal terá uma estrutura similar a essa. Alguns trechos de comentários foram propositalmente excluídos para que o código pudesse ficar mais legível e de fácil compreensão.

Figura 4 | Classe principal do projeto

```
1  package encapsulamento;
2
3  public class Encapsulamento{
4
5      public static void main(String[] args) {
6
7      }
8
9  }
```

Fonte: adaptado pelo autor, do programa *Netbeans* (2018).

Podemos observar que no início da classe, sempre apontamos o nome do pacote (*pasta*) a que pertence e, em seguida, declaramos a classe como sendo pública, pois deverá ser acessada e visualizada por todas as demais classes do sistema.

Em Java, uma classe avulsa contendo apenas métodos e atributos não pode ser executada diretamente, a menos que seja uma classe principal (que tenha um escopo para suportar ser executada). Em nosso exemplo da Figura 4, o que caracterizará essa classe como principal é a declaração **main**, contida na linha 5 do código. Essa declaração fará com que seja, nativamente, a tela inicial do sistema e consiga executar a si e às instâncias das demais classes.

Por questões metódicas, em POO, procuramos organizar as funções do sistema em classes separadas da principal, de forma que haja a possibilidade de reaproveitar o código, que é justamente um dos principais objetivos da Orientação a Objetos. Assim, vamos criar uma nova classe chamada Pessoa, que armazenará o escopo no qual trabalharemos. O processo é simples: clique sobre o pacote atual, com o botão direito do mouse, e selecione a opção *Novo > Classe Java*. Atribua o nome de *Pessoa* a essa classe e clique em *Finalizar*. Com esses passos, uma nova classe estará aninhada à classe Encapsulamento.

Alimentaremos essa classe com alguns trechos de código, vistos nas aulas anteriores. Teremos um conjunto de atributos e um método chamado *envelhecer*, que receberá um parâmetro do tipo inteiro, com nome atribuído de *aumentaldade*.

Figura 5 | Classe Pessoa

```
1
2 package encapsulamento;
3
4 public class Pessoa{
5     // atributos
6     public String nome;
7     public int idade = 10;
8     public String cpf;
9     public String enderecoCompleto;
10    public int anos;
11    // métodos
12    public void envelhecer(int aumentaIdade){
13        System.out.println(idade = idade + aumentaIdade);
14    }
15 }
16
```

Fonte: adaptado pelo autor, do programa *Netbeans* (2018).

Na linha 7, declaramos que a idade inicial apontada é 10. Isso significa que todo objeto que for criado a partir dessa classe, terá como idade inicial o valor 10.

Já na linha 12, podemos observar que o método *envelhecer* possui um parâmetro inteiro, denominado *aumentaldade*, que, como visto na unidade 2, receberá um valor vindo a partir de um objeto instanciado.

Por fim, na linha 13, utilizamos o comando *System.out.println* para exibir uma informação na tela, que seria a somatória da idade de uma determinada pessoa acrescida do valor armazenado dentro do parâmetro que o método recebeu.

Em seguida, prepararemos a classe principal (Encapsulamento) para que possa interagir com a classe Pessoa e possam trafegar dados entre si.

Figura 6 | Classe Principal (Encapsulamento)

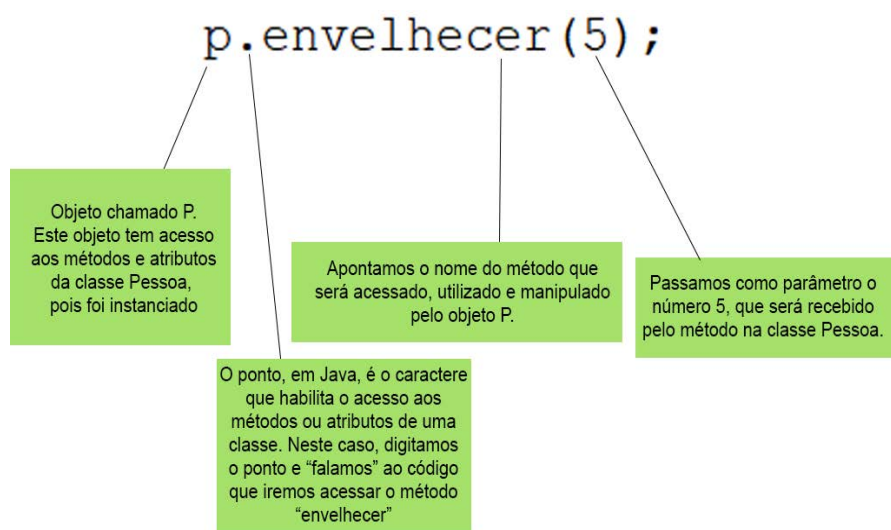
```
1  package encapsulamento;
2
3  public class Encapsulamento{
4
5      public static void main(String[] args) {
6          // Instância da classe
7          Pessoa p = new Pessoa();
8          System.out.println("anos" + p.idade);
9          p.envelhecer(5);
10     }
11
12 }
13
```

Fonte: adaptado pelo autor, do programa *Netbeans* (2018).

As linhas iniciais desempenham o mesmo papel já visto na classe Pessoa, então partiremos direto à estrutura da linha 7, onde fizemos a instância da classe Pessoa. Agora temos um objeto chamado *p*, que é uma instância da classe Pessoa, ou seja, esse objeto é capaz de acessar os métodos e atributos da classe pessoa.

Para fins comparativos, na linha 8, pedimos ao sistema que imprimisse a idade daquele objeto antes que, por meio do método *envelhecer*, aumentássemos essa idade. Logo em seguida, na linha 9, utilizamos o mesmo objeto, acessando o método *envelhecer* e passando como parâmetro o valor 5. Vamos entender o que está acontecendo:

Figura 7 | Detalhamento da linha 9, da Classe Principal



Fonte: adaptado pelo autor, do programa *Netbeans* (2018).

Ao executarmos essa classe, o resultado exibido no console de saída será o seguinte:

Figura 8 | Console de saída com resultado de execução

```
Saída - encapsulamento (run)
run:
anos10
15
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Fonte: adaptado pelo autor, do programa *Netbeans* (2018).

O primeiro resultado impresso foi *anos10*, que é o valor inicial do atributo *idade* da classe Pessoa. Imprimimos esse valor apenas para que pudéssemos comparar se, de fato, a idade seria aumentada após a inserção do método.

Logo depois, o valor *15* é exibido na tela, comprovando que o método foi acessado e recebeu o valor do parâmetro (5), somando-o à idade inicial, resultando na idade final, *15*.

Tente reproduzir essa mesma estrutura, em seu computador, e tenha autonomia e liberdade para modificar os valores e testar novos cenários.

LINK




Monografia de conclusão de curso, que utiliza vários conceitos com linguagem simples e prática para auxiliar na compreensão dos conceitos de encapsulamento. SOUSA, Alisson Rodrigues de. Aplicações de padrões de projeto com a linguagem PHP. Disponível em: <<http://www.bibliotecadigital.ufmg.br/dspace/bitstream/handle/1843/BUOS-94NMN6/alissonrodrigues.pdf?sequence=1>>. Acesso em: 27 ago. 2018.

1.2 Acesso público (*public*)

Agora que já testamos de maneira prática um dos conceitos de POO, nos aprofundaremos quanto à estrutura de encapsulamento, que é o personagem principal desse capítulo. Deixamos um capítulo inteiro para esse tópico, pois o encapsulamento é o que norteia a base teórica da Programação orientada a Objetos (SOUSA, 2009 apud LEITE & JUNIOR, 2009)².

² SOUSA, Alisson Rodrigues de. **Aplicação de Padrões de Projeto com a Linguagem PHP**. Disponível em: <<http://www.bibliotecadigital.ufmg.br/dspace/bitstream/handle/1843/BUOS-94NMN6/alissonrodrigues.pdf?sequence=1>>. Acesso em: 14 ago. 2018.



Nos capítulos anteriores, criamos algumas classes usando pseudocódigo e a primeira informação que inserimos ao definir a classe foi “Classe pública”, assim como no exemplo que acabamos de ver. Posteriormente, convertemos essa estrutura para código Java, originando o termo “*public class*”.

Essa definição de classe pública é a mais genérica, ou seja, é a forma convencional e nativa de um código trabalhar: sem restrições de acesso.

Com as classes em modo público, qualquer outra classe poderá ter acesso para ler seus métodos e atributos, bem como herdar seus dados. Mesmo que uma classe não seja herdeira explícita de outra, o acesso à sua estrutura estará liberado caso seu escopo seja público.

Todos os valores, desde a classe até métodos e atributos, estão com de acesso público, ou seja, é possível que qualquer classe ou método, interno ou externo, realize modificações nessa estrutura; uma outra classe também poderá ainda herdar dados a partir dessa.

É importante ressaltar que quando nos referimos a acessos de uma classe a outra, não estamos falando propriamente de uma classe herdeira. Uma classe comum (A) pode acessar os atributos e métodos de outra classe (B), desde que a classe A seja herdeira da classe B.

Por padrão, se nenhuma palavra-chave de encapsulamento for inserida antes do nome da classe, método ou atributo, esses serão considerados públicos. Ainda que a não inserção da palavra reservada *public* já transpareça que a intenção é tornar um determinado elemento público, podemos considerar como boa prática mencioná-la, a fim de padronizarmos a estrutura do código em desenvolvimento. Em resumo, numa regra geral, evita-se declarar classes, métodos ou atributos sem nenhum tipo de restrição.

PARA SABER MAIS

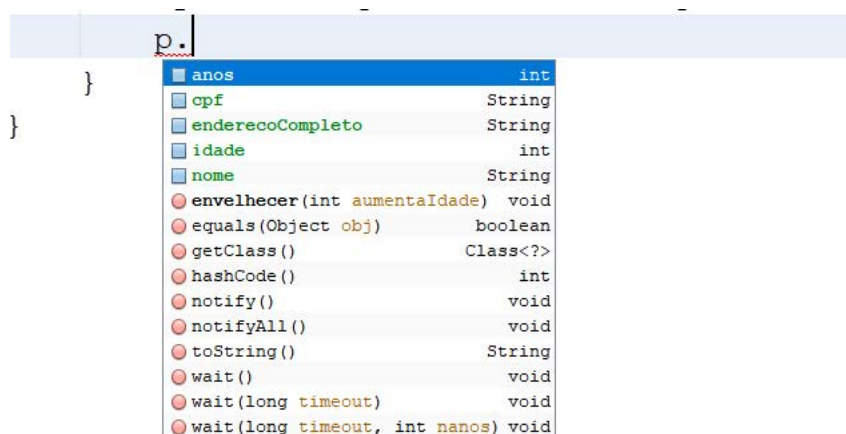


Lembre-se de que é possível aplicar encapsulamentos às Classes, métodos ou atributos. Isso implica algumas regras:

- Se uma classe for pública, todos seus métodos e atributos também o serão, exceto se forem explicitamente ocultados por meio de encapsulamento próprio.
- Se uma classe for privada, todos os seus métodos e atributos serão ocultados, mesmo que sejam declarados públicos.

Por meio do *Netbeans* podemos ver claramente a disponibilidade de um método ou um atributo, no momento em que tentarmos invocá-lo, a partir de um objeto que seja uma instância de classe. No momento em que inserimos o *ponto* após o nome do objeto, podemos pressionar as teclas CTRL+Espaço e uma lista dos métodos e atributos de uma classe serão exibidos. Apenas métodos liberados aparecerão na listagem:

Figura 9 | Acessando os métodos e atributos de uma classe por meio de seu objeto instanciado



Fonte: adaptado pelo autor, do programa *Netbeans* (2018).

É possível verificar, na Figura 9, que todos os atributos (anos, cpf, endereçoCompleto, idade, nome) e o método (envelhecer) estão disponíveis para serem usados. Se não estivessem, sequer seriam exibidos nessa listagem.

1.3 Acesso privado (*private*)


Ao usar a declaração de encapsulamento do tipo privado, apenas a classe que contém o método ou atributo poderá ter acesso para manipulá-los. Se essa classe for uma classe-pai ou uma classe abstrata, que tenha vinculada a si alguma outra classe (classe-filha) como herdeira, a herdeira não terá acesso a esses métodos e atributos privados da classe pai.

EXEMPLIFICANDO

Considere essa estrutura como sendo algo no mundo real: um pai é detentor de alguns bens, um carro, uma moto, um apartamento, uma casa de praia e uma fazenda. Por questões de direitos de herança, seus filhos têm total direito de usufruir desses bens, todavia, em determinado momento, o pai impõe uma regra que determina o seguinte parâmetro: a fazenda, o apartamento e a casa de praia são de uso privado dele e nenhum filho poderá utilizá-los. Essa regra faz entender que, mesmo que os bens citados pudessem ser de direito de uso por parte dos filhos, existe uma determinação explícita para não utilizarem. Restará, aos filhos, fazerem uso apenas dos bens que não foram proibidos pelo pai, ou seja, o carro e a moto.

- Pai: classe-pai ou classe abstrata.
- Filhos: classes-filhas, classes herdeiras ou subclasses.
- Bens: métodos e atributos.

Havendo a determinação explícita de proibição de uso de um determinado método ou atributo,



apenas a classe-pai poderá utilizá-lo, ou seja, a própria classe que detém os métodos e atributos é que deverá ser instanciada por um objeto para ter acesso de manipulação dos dados.

A exemplificação acima apontou, como base, uma classe pai ou uma classe abstrata, mas como visto no capítulo 3, via de regra, não geramos instâncias de classes abstratas, considerando que possuem apenas um escopo genérico para serem utilizados por subclasses. O exemplo incluiu o conceito de classe abstrata apenas para complementar o pensamento pedagógico do assunto, que não geraria erros de execução, apenas uma incoerência lógica.

É comum não ficar muito claro os motivos que levam o programador/desenvolvedor a utilizar classes privadas, mas consideremos a situação seguinte para podermos discutir o assunto, afim de compreendermos a respeito da importância do uso do encapsulamento privado.

Baseado no exemplo anterior, pudemos compreender que o acesso aos métodos e atributos de uma classe são liberados se a classe for pública, permitindo tais acessos por seus próprios métodos ou por suas classes herdeiras. Já uma classe privada, limita esse acesso apenas a si própria, não autorizando outras classes (herdeiras ou não) a utilizar sua estrutura de métodos e atributos.

Entretanto, também verificamos que uma classe pública pode ter seus métodos e atributos privados. Essa restrição fará com que apenas determinado método ou atributo não possa ser acessado externamente, mesmo que seja por uma instância da própria classe. Vejamos um exemplo a seguir.

Utilizando a mesma estrutura já trabalhada (classes *Pessoa* e *Encapsulamento*), vimos tornar alguns atributos inacessíveis por meio do encapsulamento privado:

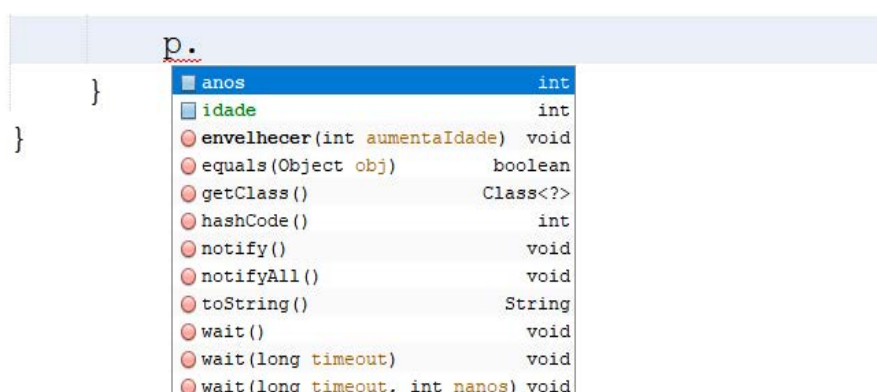
Figura 10 | Encapsulamento privado

```
1
2 package encapsulamento;
3
4 public class Pessoa{
5     // atributos
6     private String nome;
7     public int idade = 10;
8     private String cpf;
9     private String enderecoCompleto;
10    public int anos;
11    // métodos
12    public void envelhecer(int aumentaIdade){
13        System.out.println(idade = idade + aumentaIdade);
14    }
15
16 }
17
```

Fonte: adaptado pelo autor, do programa *Netbeans* (2018).

Limitamos o acesso aos atributos *nome*, *cpf* e *enderecoCompleto*, tornando-os privados. Ao tentarmos acessá-los, por meio da classe *Encapsulamento*, podemos ver o resultado dessa restrição:

Figura 11 | Encapsulamento privado vetando o acesso aos atributos



Fonte: adaptado pelo autor, do programa *Netbeans* (2018).

Mesmo que a classe *Pessoa* seja pública, o acesso a alguns de seus atributos está vetado. Compare a Figura 11 com a Figura 9 e veja que há um déficit entre os atributos disponíveis para uso. Mesmo que a classe

Encapsulamento fosse uma herdeira da classe Pessoa, esse acesso estaria vetado; o mesmo vale para o uso dos métodos.

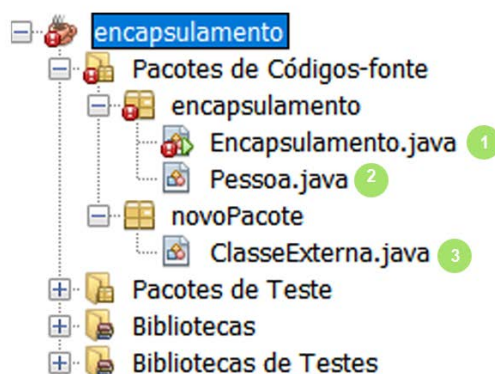
Tente, em seu computador, executar as mesmas operações para comprovar a eficácia do encapsulamento privado.

1.4 Acesso padrão (*default*)

Diferente dos demais modificadores de acesso, o modificador default (padrão) não é escrito explicitamente. Seu emprego acontece pela ausência de qualquer modificador.

O acesso ao atributo e método sem modificador (*default*) é liberado apenas aos métodos das classes que estiverem no mesmo pacote (*package*) que o atributo sem modificador, como demonstra a Figura 12.

Figura 12 | Encapsulamento privado, vetando o acesso aos atributos



Fonte: adaptado pelo autor, do programa *Netbeans* (2018).

Na Figura 12, temos um novo *package* chamado novoPacote, com uma classe chamada ClasseExterna. Se as classes *Pessoa* e *Encapsulamento* (1 e 2) do primeiro pacote estiverem com o modificador *default*, ou seja, sem nenhum modificador, conseguem acessar os métodos e atributos,

uma da outra, livremente, enquanto a classe que está no *NovoPacote* (*ClasseExterna*), não.

ASSIMILE



Respeitar os pacotes (linguagem Java) pode ser considerado um dos critérios de boa prática para desenvolvimento. Além de prover a organização do código, auxilia de forma direta no encapsulamento do conteúdo. Assim, temos:

- Classes públicas: acessíveis a qualquer classe, mesmo que estejam em pacotes diferentes.
- Classes defaults: acessíveis apenas às classes que estiverem no mesmo pacote, ou seja, que tratem do mesmo assunto como já mencionamos anteriormente.

1.5 Acesso Protegido (*protected*)

O modificador *protected* acaba por causar algumas divergências conceituais em programadores com pouca experiência em Orientação a Objetos, pois se comporta praticamente da mesma forma que o modificador default, com a diferença que as classes filhas conseguem ter acesso ao atributo ou método independente de estar no mesmo pacote ou não, desde que a classe a ser acessada seja subclasse (classe filha) da classe que está tentando o acesso.

1.6 Resumo de regras dos modificadores

A tabela a seguir poderá contém as regras dos modificadores de acesso na Orientação a Objetos.

Tabela 1 | Resumo das regras dos modificadores

MODIFICADOR	CLASSE	PACOTE	SUBCLASSE	TODOS
default(padrão)	SIM	SIM	NÃO	NÃO
public	SIM	SIM	SIM	SIM
private	SIM	NÃO	NÃO	NÃO
protected	SIM	SIM	SIM	NÃO

Fonte: adaptado pelo autor, do programa *Netbeans* (2018).

QUESTÃO PARA REFLEXÃO

No decorrer desse capítulo, apresentamos algumas propostas de modificações dos exemplos utilizados. Um bom aprendizado se baseia em testes e tentativas para chegar a novos resultados. Se você precisasse expandir o projeto, adicionando os métodos *rejuvenecer*, *caminhar*, *modificarNome*, qual implementação faria?

► 2. Considerações Finais

- Encapsulamento é a proteção dada às classes, métodos e atributos, permitindo ou negando o acesso de outros métodos a eles.
- O modificador público (*public*) permite que qualquer outra classe o acesse, estando ou não no mesmo pacote.
- Modificadores *default* ficam implícitos. Um método ou um atributo está com encapsulamento *default* se antes de sua declaração não houver nenhuma palavra reservada.
- O encapsulamento privado (*private*) limita o acesso aos métodos e atributos da classe apenas a ela mesma.

- O encapsulamento protegido (*protected*) limita o acesso dos métodos e atributos apenas a subclasses da classe pai, estando elas no mesmo pacote ou não.

VERIFICAÇÃO DE LEITURA

TEMA 4



1. Encapsular, em POO, significa:

- a) Proteger as classes para evitar acesso de outros programadores que estejam trabalhando no mesmo projeto.
- b) Proteger os atributos, por meio da palavra reservada `public`.
- c) Proteger os métodos, por meio da palavra reservada `public`.
- d) Proteger os atributos e métodos para que não sejam modificados por métodos de outras classes, sem as devidas permissões.
- e) Proteger a classe para que não seja modificada por uma classe abstrata, por meio de sua instancição incorreta.

2. Quanto aos parâmetros, podemos afirmar que:

- a) São enviados, pelo objeto, ao método da classe instanciada.
- b) São recebidos pela classe instanciada, pois são enviados por uma classe herdeira.
- c) São enviados de uma classe herdeira a uma classe-pai, apenas se estiverem no mesmo pacote.
- d) São enviados de um pacote ao outro, apenas se, no momento da instanciação do objeto, o

encapsulamento for público.

- e) São enviados do objeto à classe que esteja, obrigatoriamente, no mesmo pacote.

3. Em Java:

- a) Os pacotes funcionam como pastas que armazenam métodos similares.
- b) Os atributos devem ficar em pacotes similares para fins organizacionais.
- c) Os pacotes têm o mesmo papel de pastas e servem para agrupar classes que tratem de assuntos relacionados.
- d) O acesso aos métodos e atributos de qualquer classe privada se dá por meio de um ponto.
- e) É necessário informar a qual pacote cada classe pertence. Essa declaração pode ser feita por meio da palavra-chave `import`.

Glossário

- **Modificador:** usamos esse termo como um sinônimo, na computação, para encapsulamento, pois o modificador de acesso muda o tipo de encapsulamento atribuído.
- **Void:** significa valor. Na computação, utilizamos essa palavra reservada para informar que uma determinada operação não precisará retornar um valor ao final do processamento (mesmo que o retorno ocorra implícito).

► Referências Bibliográficas

FELIX, Rafael. **Programação Orientada a Objetos**. São Paulo: Pearson Education do Brasil Ltda., 2016.

SOUSA, Alisson Rodrigues de. **Aplicação de Padrões de Projeto com a Linguagem PHP**. Disponível em: <<http://www.bibliotecadigital.ufmg.br/dspace/bitstream/handle/1843/BUOS-94NMN6/alissonrodrigues.pdf?sequence=1>>. Acesso em: 14 ago. 2018.

► Gabarito – Tema 4

Questão 1 – Alternativa: D

O objetivo do encapsulamento é dar proteção aos métodos e atributos para que sejam modificados apenas quando explicitamente autorizados no código.

Questão 2 – Alternativa: A

O objeto instanciado, quando for invocar o método ou atributo, deve informar o parâmetro que será passado (caso exista).

Questão 3 – Alternativa E

Os *packages* servem para organizar as classes que possuem similaridade ou relacionamento de conteúdo, assim, num mesmo pacote, agrupam-se as classes que tratem de um mesmo objetivo no sistema.



TEMA 05

GETTERS E SETTERS

► Objetivos

- Compreender a aplicabilidade dos métodos acessores/ modificadores (Getters e Setters), na estrutura de uma classe.
- Entender as sintaxes dos métodos getters e setters.
- Conhecer possíveis derivações dos métodos getters e setters.

► Introdução

Toda linguagem de programação, que se baseie ou ao menos utilize o paradigma da Programação Orientada a Objetos, utiliza a estrutura que denominados como métodos especiais.

Para conceber com plenitude o conteúdo desta unidade, recomendamos que revise as aulas onde tratamos a respeito dos métodos e suas estruturas. A aula de encapsulamento também pode auxiliar a entender os motivos que tornam os métodos getters e setters necessários e tão importantes.

É esperado que, ao final dessa unidade, você domine as bases teóricas de como aplicar tais métodos em uma estrutura de sistema, bem como saiba manipular as informações tratadas por meio deles.

Trataremos ainda de um método que esteve presente conosco desde o início, mas de forma implícita, denominado método construtor. Embora seja mais conceitual que aplicável, é importante que cubramos todo o conteúdo para que nada fique para trás na estrutura de nosso curso. É pré-requisito para um melhor aproveitamento, que você tenha instalado em seu computador, a IDE Netbeans e o JDK por meio dos *links* de *download* que recomendamos na aula anterior. Também será um pré-requisito que, baseado na unidade 4, recorde dos passos necessários para criarmos um novo projeto Java, novos pacotes e novas classes.

► 1. Conceituando

Vimos, anteriormente, que o encapsulamento possibilita a segurança de acesso a determinados atributos ou métodos de uma classe, que é um dos focos do POO (Programação Orientada a Objetos). De forma similar, é possível ainda conceder ou vetar permissões aos atributos de uma

determinada classe, impedindo ou permitindo acesso ao seu escopo. Essa medida de segurança provê maior confiabilidade do sistema.

Existem dois tipos de métodos especiais que fazem atingir esses critérios de segurança: os métodos acessores (*getters*) e os métodos modificadores (*setters*). Em geral, utilizaremos ambos métodos como *get* e *set*.

Sabemos que todos os métodos têm uma característica bastante peculiar em sua estrutura, que os diferencia bastante dos atributos, que é o fato de poderem receber parâmetros vindos de um objeto instanciado. Esses parâmetros são colocados entre parênteses, logo depois do nome do método:

```
public void caminhar()
```

ASSIMILE



Os métodos *getters* não possuem parâmetros, enquanto os *setters*, sim. Lembre-se de que parênteses são necessários, afinal, estamos tratando a respeito de métodos, que precisam ter em sua estrutura os parênteses após o nome; o que é dispensado nos *getters* são seus parâmetros, fazendo com que o método fique vazio, como no exemplo acima.

Fazendo uma alusão ao Inglês, *get* seria algo como pegar, resgatar, ter acesso. Logo, pressupõe-se que ao utilizarmos os métodos *get* explicitamos a intenção de pegar alguma informação. Já o método *set*, tem a função de setar, apontar, inserir/ alterar algo, o que nos leva a compreender que quando quisermos realizar mudanças, utilizaremos o *set*.

A invocação dos métodos é feita utilizando essas próprias palavras-chave *get* e *set*, inseridas antes do nome do atributo com o mesmo nome. Essa é a forma mais simplista de se realizar tal operação. Vamos continuar

utilizando o exemplo da classe Pessoa, trabalhado até então, e gerar os métodos *getters/setters*:

Figura 1 | Criando getters

```
1 package encapsulamento;
2
3 public class Pessoa{
4     // atributos
5     protected String nome;
6     public int idade = 10;
7     private String cpf;
8     private String enderecoCompleto;
9     public int anos;
10    // métodos
11    private void envelhecer(int aumentaIdade){
12        System.out.println(idade = idade + aumentaIdade);
13    }
14
15    public int getAnos() {
16        return anos;
17    }
18 }
```

Fonte: adaptado pelo autor (2018).

Utilizando o código que desenvolvemos na unidade 4, criamos agora o nosso primeiro *getter* para o atributo *anos*.

PARA SABER MAIS



Caso você tenha muitos atributos a serem protegidos pelos getters e setters, pode utilizar um recurso do Netbeans para inseri-los de forma automática. Basta posicionar o cursor no local onde deseja que os métodos sejam inseridos e pressionar ALT+INSERT. Uma tela será exibida para escolher o que deseja inserir. Escolha os getters ou os setters. Feito isso, uma janela será apresentada, requerendo que você selecione quais os atributos que deseja implementar.

► 2. Qual a importância desses métodos?

Simule o seguinte cenário: você está num restaurante e mesmo sem ter consumido nada, pede a conta ao garçom, que informa que você não deve nada. Para que o garçom pudesse dar essa informação, primeiramente, verificou, junto ao caixa, se sua comanda continha algum registro de consumo. Em seguida, volta à sua mesa e retorna a resposta, comunicando que o valor do **débito é igual a zero**.

Veja que alguns princípios precisaram ser respeitados nessa operação:

1. Você não teve acesso direto ao sistema do caixa para verificar sua conta.
2. O garçom foi o elo entre você e o sistema do caixa, realizando a ação de pegar um dado e trazê-lo a você.

Trazendo para o mundo do POO, temos os seguintes personagens e seus respectivos papéis:

- Você: o método que acessa os atributos.
- Garçom: o método *getter* que pega a informação.
- Débito: o valor do atributo (no caso, zero).

Compreenda que os *getters* são focados, especificamente, em intermediar o acesso que um método teria a um atributo. Logo mais, veremos o funcionamento dos *setters*.

Num primeiro momento, a impressão que surge é que esse processo apenas aumentou a complexidade do sistema, mas essa é uma medida de segurança para impedir que os métodos tenham livre acesso aos dados do atributo, seja para resgatar valores ou para modificá-los. O uso dos *getters* e *setters* não é uma obrigatoriedade para que seu código funcione, todavia, se temos acessível um recurso de segurança, não há motivos para não o utilizar.



LINK

Você pode executar seus códigos Java on-line, sem propriamente precisar ter instalado algo em seu computador. Essa é uma alternativa para quando você quiser treinar seus conhecimentos, mas não estiver com uma IDE à mão. Disponível em: <<http://browxy.com/>>. Acesso em: 28 ago. 2018.

► 3. Como utilizá-los?

Abaixo temos a classe Pessoa e nas linhas 18, 19 e 20, criamos o *getIdade*, que resgatará o valor da idade contido no atributo. O método precisa sempre ter um valor de retorno, pois se resgatou alguma informação, precisa retornar o que foi resgatado. O tipo de dado do retorno também precisa ser o mesmo que está definido no atributo e, se o atributo for um inteiro, o método precisa ser um inteiro; se o atributo for uma String, o método precisa ser uma String; procure se lembrar do exemplo do restaurante. Faria algum sentido você perguntar ao garçom em qual valor estava sua conta e ele, depois de verificar, voltar à sua mesa e não dizer nada? Da mesma forma, faria algum sentido ele retornar e falar quais os tipos de suco que eles têm disponíveis? Obviamente, nenhuma das suposições faz sentido. Se você questionou o garçom sobre o valor de sua conta, ele precisará, obrigatoriamente, dar um **retorno** que atenda ao **tipo** de pergunta que você fez.

Figura 2 | Criando getters para mais atributos

```
1 package encapsulamento;
2 public class Pessoa{
3     // atributos
4     protected String nome;
5     public int idade = 10;
6     private String cpf;
7     private String enderecoCompleto;
8     public int anos;
9     // métodos
10    private void envelhecer(int aumentaIdade){
11        System.out.println(idade = idade + aumentaIdade);
12    }
13
14    public int getAnos() {
15        return anos;
16    }
17
18    public int getIdade() {
19        return idade;
20    }
21 }
```

Fonte: elaborado pelo autor (2018).

Já na classe Encapsulamento, que é a mesma utilizada na unidade anterior, as mudanças são muito sutis. Veja a imagem a seguir:

Figura 3 | Classe Encapsulamento invocando um atributo pelo getter

```
1 package encapsulamento;
2
3 import novoPacote.NovaClasseExterna;
4
5 public class Encapsulamento extends NovaClasseExterna{
6
7     public static void main(String[] args) {
8         // Instância da classe
9         Pessoa p = new Pessoa();
10        System.out.println(p.getIdade());
11    }
12 }
```

Fonte: elaborado pelo autor (2018).

A mudança ocorreu na linha 10. Fizemos apenas a chamada de um atributo, porém, dessa vez, passando pela proteção do *getter*. Compare abaixo ambas as formas e veja como aconteceram poucas alterações:

Tabela 1 | Comparativo de uso do Getter

Acesso direto	Acesso via <i>getter</i>
<code>p.idade;</code>	<code>p.getIdade();</code>

Fonte: elaborado pelo autor (2018).

Embora estejamos invocando um atributo que **não tem parênteses em seu escopo**, ao proteger o acesso a ele via *getter*, que é um método, esse atributo passa a ser invocado por seu representante, que também é um método e, por consequência, recebe parênteses.

A lógica dos *setters* segue a mesma linha, ou seja, é criado um intermediário que ficará entre o método que solicita a alteração e o atributo que recebe a alteração. Novamente, retome o exemplo da lanchonete para facilitar a compreensão:

EXEMPLIFICANDO

Outrora, o que tráfegou entre você (método) e o garçom (*getter*) foi apenas a informação a respeito dos seus débitos no restaurante. Entretanto, você não entregou nada ao garçom, apenas solicitou (pegou, *get*) informações e recebeu um retorno. Suponha que a situação sofreu uma pequena mudança: você já realizou algum tipo de consumo e, novamente, pede a conta ao garçom (*get*); que verifica junto ao caixa o valor de seus débitos e retorna, informando que você deve R\$ 50,00. Com base no retorno, você retira uma nota de R\$ 50,00 e a entrega ao garçom. No momento em que esse

dinheiro for registrado no caixa, o seu débito, que tinha um valor atrelado, será modificado para zero. O garçom volta à sua mesa, com a nota fiscal e dá um retorno, informando que a conta está paga.

O que diferencia a situação do getter, utilizado anteriormente, para esse novo cenário com o setter, é que houve a entrega de algo do método (você) ao garçom (setter): a nota de R\$ 50,00 que alteraria o valor do débito.

Assim, observe que é uma regra padrão: para que haja alteração de um atributo por um setter, o método precisa, obrigatoriamente, enviar um parâmetro. Via de regra, o método get sinaliza o recebimento de algo, já o método set aponta o envio de uma informação (FELIX, 2016) ¹.

Os métodos *setters* não têm como regra a obrigatoriedade de parâmetros, mas, geralmente, é esse o cenário que vemos na maioria dos sistemas. A estrutura do *setter* dentro da classe Pessoa seguiria a mesma lógica do *getter*, inclusive no momento da inserção, que pode ser feita por meio dos atalhos citados anteriormente (para IDE Netbeans).

Figura 4 | Uso do *setter*

```
1 package encapsulamento;
2 public class Pessoa{
3     // atributos
4     protected String nome;
5     public int idade = 10;
6     private String cpf;
7     private String enderecoCompleto;
8     public int anos;
9     // métodos
10    private void envelhecer(int aumentaIdade){
11        System.out.println(idade = idade + aumentaIdade);
12    }
13
14    public int getIdade() {
15        return idade;
16    }
17
18    public String setNome(String nome) {
19        this.nome = nome;
20        return nome;
21    }
```

Fonte: elaborado pelo autor (2018).

¹ FELIX, Rafael. **Programação Orientada a Objetos**. São Paulo: Pearson Education do Brasil Ltda., 2016.

Fazendo novamente a ponte com o exemplo do restaurante, lembremos que o garçom deveria, obrigatoriamente, dar retorno do mesmo tipo de sua pergunta? Assim, ao utilizarmos os métodos *setters*, ele deve ser do mesmo tipo que o atributo a ser modificado e deverá ainda ter um retorno. Esse retorno pode ser diverso: um cálculo, uma frase, uma imagem etc. Entretanto, em nossa classe Pessoa/ Encapsulamento, queremos que o retorno seja um nome, logo, pediremos ao método que dê um **return nome**.

Uma nova palavra-chave surge nesse momento, na linha 19: **this**.

A maneira mais prática de entender essa palavra é observar o nome da variável nome e o nome do parâmetro recebido. Veja a imagem a seguir:

Figura 5 | Palavra-reservada *this*

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

O diagrama mostra o código acima com três anotações explicativas:

- Uma linha tracejada aponta do parâmetro `nome` na assinatura do método para o texto: "Parâmetro chamado 'nome'".
- Uma linha tracejada aponta do atributo `nome` no lado direito da atribuição `this.nome = nome;` para o texto: "Variável (atributo) chamado 'nome'".
- Uma linha tracejada aponta para o trecho `this.nome = nome;` com o texto: "Informo ao meu sistema que o atributo 'nome' deverá receber o valor vindo no parâmetro 'nome'".

Fonte: elaborado pelo autor (2018).

A palavra nome se repete algumas vezes no código. Na primeira linha, é um parâmetro; na segunda, no trecho "**this.nome**", se refere ao atributo *nome* daquela classe (por isso o uso da palavra *this*, que em inglês significa essa, esse-leia como nessa classe). Em termos práticos, `this.nome = nome` deveria ser lido como **nessa classe, o atributo nome recebe o parâmetro nome**. Sempre que for necessário referenciar algum atributo dentro da própria classe, recomenda-se recorrer ao uso do *this*.

É importante não confundir o conceito do *this*. Seu propósito não é diferenciar nomes, afinal, é possível utilizar qualquer nome para os parâmetros e não exatamente o mesmo nome que os atributos. Foque em utilizar

o this sempre que estiver utilizando um recurso interno da classe dentro dela mesma.

Enquanto isso, na classe Encapsulamento, podemos modificar o valor do atributo nome, que outrora era vazio, para qualquer nome que desejarmos:

Figura 6 | Classe Encapsulamento, modificando um atributo pelo setter

```
1 package encapsulamento;
2
3 import novoPacote.NovaClasseExterna;
4
5 public class Encapsulamento extends NovaClasseExterna{
6
7     public static void main(String[] args) {
8         // Instância da classe
9         Pessoa p = new Pessoa();
10        System.out.println(p.setNome("Jose"));
11    }
12 }
```

Fonte: elaborado pelo autor (2018).

A forma anterior (sem *setter*) e essa nova forma, com *setter*, é similar, com pequenas adições que garantirão que apenas os métodos autorizados consigam acessar e modificar os atributos de uma classe. Compare a diferença na Tabela 2:

Tabela 2 | Comparativo de uso do Setter

Modificação direta	Modificação via <i>setter</i>
<code>p.nome = "José";</code>	<code>p.setNome("José");</code>

Fonte: elaborado pelo autor (2018).



PARA SABER MAIS

É importante que, sempre que um novo atributo venha a ser criado, criar imediatamente os seus respectivos getters e setters. Isso auxiliará a criar um padrão de desenvolvimento, habituando-se a sempre utilizar esses mecanismos de proteção de acesso.

Apesar de parecer um trabalho oneroso, dependendo da quantidade de atributos que sua classe tenha, o custo-benefício dará como recompensa um sistema que unirá forças com o encapsulamento, gerando como resultado uma plataforma sólida e segura.

► 4. Método construtor

O método construtor, como o próprio nome já demonstra, faz a construção inicial de uma classe, de forma que os objetos instanciados a partir dela tenham algumas configurações iniciais, antes mesmo que haja qualquer implementação.

Utilizando a classe Pessoa, vista até então, e seu método *envelhecer*; suponhamos que num sistema que estejamos desenvolvendo, queiramos que cada objeto instanciado já inicie com um valor padrão para envelhecer, ou ainda, que cada objeto instanciado já comece com uma idade inicial. Esses recursos são fornecidos pelos métodos construtores, que constroem um objeto com valores padrões, que podem ser modificados posteriormente, mas se não forem, terão os dados definidos.

Essa técnica é uma forma de garantir que, mesmo que um objeto não receba atributos, já tenha uma configuração padrão comum a todos os objetos derivados daquela classe.

Em Java, para que um método seja considerado como construtor, deve ter o mesmo nome que a classe. Cada linguagem tem sua abordagem quanto aos construtores, mas como nosso foco é o Java, trataremos apenas de sua estrutura.

Figura 7 | Método construtor

```
1 package encapsulamento;  
2 public class Pessoa{  
3  
4     public void Pessoa() {  
5     }  
6 }
```

Fonte: elaborado pelo autor (2018).

Na figura 7 temos a estrutura base do método construtor da classe Pessoa. Esse método deve receber os valores padrões, que serão atribuídos à classe, logo que algum objeto a instanciar, tomando as devidas cautelas para que o nome do método seja o mesmo nome da classe, além de ser público:

Figura 8 | Método construtor com valores iniciais

```
1 package encapsulamento;  
2 public class Pessoa{  
3  
4     public Pessoa() {  
5         this.nome = "João";  
6         this.anos = 65;  
7     }
```

Fonte: elaborado pelo autor (2018).

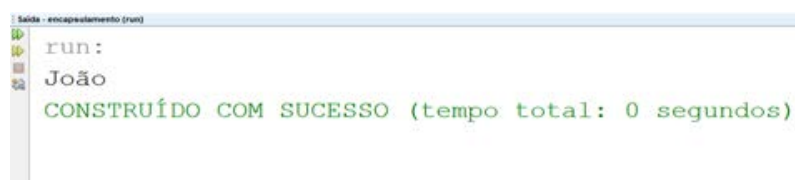
Ao executarmos a classe principal, com um objeto instanciando a classe Pessoa, o nome e os anos serão pré-carregados e, caso seja solicitado que esses dados apareçam na tela, aparecerão com os valores declarados no construtor, conforme pode ser visto nas Figura 9 e 10:

Figura 9 | Instância utilizando método construtor

```
1 package encapsulamento;
2
3 import novoPacote.NovaClasseExterna;
4
5 public class Encapsulamento extends NovaClasseExterna{
6
7     public static void main(String[] args) {
8         // Instância da classe
9         Pessoa p = new Pessoa();
10        System.out.println(p.nome);
11    }
12 }
```

Fonte: elaborado pelo autor (2018).

Figura 10 | Resultado da Instância, utilizando método construtor



```
Saida - encapsulamento (run)
run:
João
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Fonte: elaborado pelo autor (2018).

Os métodos construtores ainda podem receber parâmetros iniciais, ou seja, é possível alimentar os métodos construtores com valores pré-definidos, que são colocados nos parênteses de instanciação do objeto:

Figura 11 | Método construtor recebendo parâmetros

```
1 package encapsulamento;
2 public class Pessoa{
3
4     public Pessoa(String nome, int anos){
5         this.nome = nome;
6         this.anos = anos;
7     }
8 }
```

Fonte: elaborado pelo autor (2018).

Na linha 4, temos dois parâmetros que serão recebidos, uma *String* chamada *nome* e um inteiro chamado *anos*. Isso significa que no momento de criação de um objeto instanciado dessa classe, obrigatoriamente, devemos informar esses dois parâmetros ao construtor:

Figura 12 | Passando parâmetros iniciais ao método construtor

```
1 package encapsulamento;
2
3 import novoPacote.NovaClasseExterna;
4
5 public class Encapsulamento extends NovaClasseExterna{
6
7     public static void main(String[] args) {
8         // Instância da classe
9         Pessoa p = new Pessoa("Benedito", 36);
10        System.out.println(p.nome);
11        System.out.println(p.anos);
12    }
13 }
14
15 Saída - encapsulamento (run)
16
17 run:
18 Benedito
19 36
20 CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Fonte: elaborado pelo autor (2018).

Os métodos construtores ainda podem ser protegidos pelos *getters* e *setters* para que haja uma camada adicional de segurança.

Cada linguagem também é peculiar no tratamento de seus métodos construtores. PHP, por exemplo, aponta métodos destrutores para *esvaziar* a memória daqueles atributos que não estão mais sendo utilizados. Já com o Java, o esvaziamento dos atributos ocorre de forma natural, no momento em que a classe deixa de ser utilizada pelo programa.

QUESTÃO PARA REFLEXÃO

Baseado no conteúdo de *getters* e *setters*, vistos nessa unidade, podemos rotular o tema como um dos principais agentes de segurança do paradigma da Programação Orientada a Objetos, junto com o encapsulamento. Como ambos os temas acabam por se cruzar de forma direta, como seria o comportamento dos *getters* e *setters* diante dos métodos e tributos, com o encapsulamento privado e protegido?

► 5. Considerações Finais

- *Getters* são métodos especiais, responsáveis por pegar uma informação, no escopo de um atributo, servindo de intermediário entre o atributo em si e o método que requer seu valor.
- *Setters* são métodos especiais, responsáveis por modificar uma informação, no escopo de um atributo, servindo de intermediário entre o atributo em si e o método que modifica seu valor.
- *Métodos Construtores* são métodos especiais responsáveis por valorar os atributos de uma classe, no momento em que elas são criadas, pré-fixando dados iniciais nos atributos.
- É possível alimentar os métodos construtores com valores específicos, no momento de criação de um objeto. Para isso, é necessário passar valores do objeto ao método por meio de parâmetros.

► Glossário

- **Return:** na programação, a palavra reservada *return* é utilizada para apontar o retorno de um método, função, *procedure*. Esse retorno funciona como o resultado de um processamento. Dessa forma, depois de um método processar N ações, por exemplo, tem como obrigação prestar um retorno que será usado em algum outro ponto do código.

VERIFICAÇÃO DE LEITURA TEMA 5



1. Considerando os métodos especiais, é possível afirmar que:
 - a) Agem diretamente sobre métodos e atributos de

- uma classe, podendo ou não ter parâmetros.
- b) Agem diretamente sobre os atributos de uma classe, sendo responsáveis por intermediar o contato entre o método e o atributo da classe.
 - c) São divididos em três categorias: *getters*, *setters* e *construtores*, sendo esse último o responsável por alimentar o objeto com valores iniciais.
 - d) Fazem parte da segurança, dentro do paradigma da Programação Orientada a Objetos, agindo sempre em conjunto com o encapsulamento, que impõe regras de acesso aos métodos e atributos.
 - e) Apenas os *setters* e os *construtores* trabalham com parâmetros de forma obrigatória.
2. Em um algoritmo que utilize os recursos de segurança dos *getters* e *setters*, a forma correta de se transmitir um parâmetro do objeto instanciado a um atributo, seria:
- a) `classe.setInformacao(parametro)`.
 - b) `objeto.getInformacao(parametro)`.
 - c) `objeto.getInformacao() = parametro`.
 - d) `objeto.setInformacao(parametro)`.
 - e) `classe.setInformacao() = parametro`.
3. Os métodos construtores têm algumas características. Assinale a alternativa em que ao menos uma característica **não** pertence aos métodos construtores:
- a) Esses métodos: pré-fixam informações na classe; podem receber parâmetros; aceitam *set* e *get* em seu escopo.
 - b) Esses métodos: não têm um tipo específico como *int*, *String*, *Boolean*; pertencem à categoria dos métodos

especiais da POO; podem ser alimentados no momento da instanciação de um objeto.

- c) Esses métodos: são armazenados na memória, enquanto a classe a que pertencem estiver em uso; usam a palavra-chave *this* para acessar atributos, pois o acesso ocorre dentro da mesma classe; requerem os tipos de dados para cada parâmetro recebido.
- d) Esses métodos: podem receber parâmetros ilimitados, desde que correspondam à mesma quantidade de parâmetros enviados pelo objeto; precisam ter um *return* para funcionar corretamente; precisam ter o mesmo nome da classe a que pertencem.
- e) Esses métodos: apontam seus parâmetros recebidos separados por vírgula; aceitam dentro de si outros métodos; requerem que os parâmetros enviados sejam do mesmo tipo dos parâmetros esperados; garantem a intermediação de acesso aos atributos, provendo maior segurança.

Referências Bibliográficas

FELIX, Rafael. **Programação Orientada a Objetos**. São Paulo: Pearson Education do Brasil Ltda., 2016.

Gabarito – Tema 5

Questão 1 - Alternativa B

Como estudado, o principal papel dos métodos especiais *getters* e *setters* é o de intermediar o contato entre um método que tenta ler ou modificar os dados de um atributo.

Questão 2 - Alternativa D

Para transmitirmos informação, usamos os modificadores, ou seja, os *setters*, ou seja, *set*. Essa informação é transmitida a partir do objeto, que depois de instanciado, tem livre acesso à classe. O parâmetro, por fim, fica sempre dentro de parênteses. Dessa forma, a alternativa que atende a esses requisitos é a D: `objeto.setInformacao(parametro)`.

Questão 3 - Alternativa D

Das três afirmativas apontadas na alternativa D, apenas a segunda está incorreta, pois os métodos construtores não requerem um retorno, visto que sua função é apenas armazenar dados iniciais e não retornar algo ao sistema.



TEMA 06

HERANÇA

► Objetivos

- Conhecer as formas de invocar heranças de classes.
- Identificar classes como superclasses ou subclasses.
- Entender as características dos recursos de herança.
- Compreender as vantagens de utilizar herança em Programação Orientada a Objetos.
- Implementar recursos de herança em código Java.

► Introdução

O tema herança já foi abordado em capítulos anteriores quando tratamos a respeito de classes, métodos, encapsulamento, *getters* e *setters*. Deixar esse espaço para falar especificamente desse tema, comprova sua eficiência e utilidade.

A herança acaba por ser um dos vários recursos da Programação Orientada a Objetos (POO), que além de ser uma praticidade ao programador, faz com que a estrutura do sistema fique mais concisa, robusta e sem ambiguidades, pois elimina-se qualquer repetição desnecessária de código, acoplando as possíveis recorrências em apenas uma classe que funcione de repositório.

Na visão de Deitel (2010), a Programação Orientada a Objetos “tira proveito dos relacionamentos de herança, dos quais as classes de objetos novas são derivadas absorvendo-se características de classes existentes e adicionando-se características únicas dessas mesmas classes” (DEITEL, 2010, p. 15)¹.

Trataremos, nesse capítulo, sobre conceitos novos da herança e veremos, também, concomitante à linguagem Java, a forma que a herança é invocada em outras linguagens de programação como C++ e PHP. Além disso, veremos ainda, em Java, como o encapsulamento e os métodos especiais interferem de forma direta na herança e as formas de sanar possíveis problemas de acesso aos atributos e métodos de uma superclasse.

► 1. Superclasses e subclasses

As superclasses não são um conceito novo dentro de nossa estrutura de aulas. Já tratamos delas várias vezes, mas com outro nome, que, por

¹ DEITEL, Harvey M. **Java Como Programar**. São Paulo: Pearson Prentice Hall, 2010.

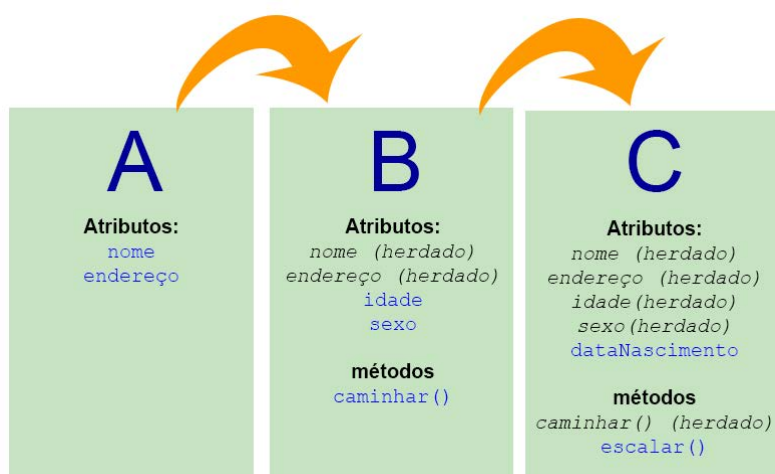
questões meramente didáticas, era o mais adequado a se fazer no momento. Conhecemos as superclasses como *classe principal* ou ainda como *classe-pai*.

Qualquer classe é elegível à posição de superclasse. Essa atribuição é mais conceitual que prática, pois, com relação a estrutura, uma superclasse em nada se difere de uma classe filha, que pode, inclusive, ser uma superclasse, caso dela derive alguma outra classe que resgate seus métodos e atributos.

► 2. Herança de herança

Um recurso interessante é a possibilidade de realizar a herança de herança. Por exemplo, uma classe C herda características de uma classe B, que, por sua vez, havia herdado características de uma classe A:

Figura 1 | Herança



Fonte: elaborado pelo autor (2018).

Dessa forma, é possível que uma classe seja, ao mesmo tempo, uma superclasse e uma subclasse, como é o caso da classe B, apresentada na Figura 1, que tem o papel de subclasse se comparada com a classe A, mas tem o papel de superclasse se comparada com a classe C.

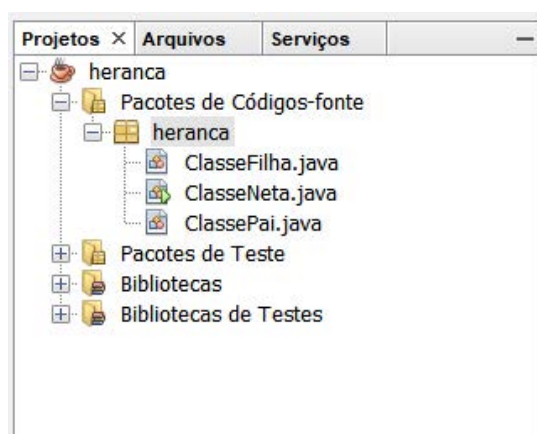
Esse tipo de recurso é extremamente útil, pois permite manter a estrutura de códigos do sistema de forma enxuta e reaproveitável, o que torna a herança um dos pilares do POO (FÉLIX, 2016)².

Agora que já vimos, conceitualmente, como uma subclasse e uma superclasse se comportam em relação uma à outra, vejamos como essa estrutura é formada dentro de nossa IDE, *Netbeans*. Primeiramente, crie um novo projeto chamado *heranca*. Nomeie a Classe Principal para “ClasseNeta”; o pacote principal a ser criado pode ser mantido como *heranca* mesmo. Conclua a criação do projeto.

Você deverá criar ainda mais duas classes Java, dentro do pacote *heranca*: *ClassePai* e *ClasseFilha*.

Ao final de toda a criação dessa estrutura, seu esquema de navegação do projeto deverá estar da seguinte forma:

Figura 2 | Esquema de navegação do Projeto



Fonte: elaborado pelo autor (2018).

PARA SABER MAIS

Você deve ter percebido que nos nomes de arquivos, projetos, métodos, atributos, classe, geralmente, suprimimos todo e qualquer caractere especial (como acentuação, espaços).

² FELIX, Rafael. **Programação Orientada a Objetos**. São Paulo: Pearson Education do Brasil Ltda., 2016.

Embora o Java até aceite alguns caracteres, como o cedilha, é uma questão de boas práticas a escrita de elementos da programação sem nenhum caractere diferente de letras ou números. Assim, sempre que você vir uma palavra grafada sem algum acento que nativamente teria, saiba que o motivo é meramente por questões estruturais de programação.

Veremos agora sobre a aplicação dessa estrutura, com relação a código. Iniciaremos pela ClassePai, que, se comparada com a Figura 1, seria a nossa *Classe A*. Para existir uma mesma linha de pensamentos, utilizaremos os mesmos métodos e atributos que foram usados na Figura 1.

Figura 3 | ClassePai

```
1 package heranca;
2
3 public class ClassePai {
4     public String nome;
5     public String endereco;
6 }
7
```

Fonte: elaborado pelo autor (2018).

Não há nenhum recurso novo nesta estrutura, pois apenas usamos artefatos que já havíamos estudado anteriormente em outros capítulos. Vamos passar agora ao código da ClasseFilha:

Figura 4 | ClasseFilha

```
1 package heranca;
2
3 public class ClasseFilha extends ClassePai {
4     public int idade;
5     public char sexo;
6
7     public void caminhar() {
8
9     }
10 }
```

Fonte: elaborado pelo autor (2018).

Análise a linha 3, da figura 4. Deixamos em destaque o seguinte trecho de código: `extends ClassePai`. Esse é o trecho que informa ao código que a `ClasseFilha` é uma extensão da `ClassePai`, ou seja, é uma herdeira da `ClassePai`, podendo, assim, ter acesso aos seus métodos e atributos livremente (salvo questões de encapsulamento, como já tratado no capítulo 4).

A palavra reservada *extends* é aplicável na linguagem Java, mas cada linguagem tem sua própria maneira de apontar heranças. Em C++, por exemplo, essa estrutura ficaria da seguinte forma:

`ClasseNeta :ClasseFilha`

Já na linguagem PHP, a herança é realizada da mesma maneira que no Java, por meio da palavra reservada *extends*. O ideal é concentrar seus esforços em compreender a base conceitual do tema herança, bem como dos demais temas já vistos e os outros que ainda serão abordados para, então, escolher uma linguagem de sua preferência para realizar os testes.

LINK



Confira mais detalhes sobre herança, nas linguagens PHP e C++ no manual de PJP, na descrição feita pelo site da Microsoft. Disponível em: http://php.net/manual/pt_BR/language.oop5.inheritance.php e <https://msdn.microsoft.com/pt-br/library/a48h1tew.aspx>. Acesso em: 31 ago. 2018.

Em Java, só será possível utilizar os recursos de uma herança dentro de uma classe que contenha um método principal, já visto em capítulos anteriores:

```
Public static void main(String[] args) {  
  
}
```

Esse método aponta ao código qual é a classe principal de um determinado pacote, ou seja, a tela principal onde todo o fluxo do sistema acontece. Dentro de um mesmo pacote, geralmente, temos apenas uma classe principal (com método principal) e as demais são apenas auxiliares, que passarão métodos e atributos para serem utilizados.

Atente ao destaque dado à palavra *geralmente* no parágrafo anterior. Isso significa que pode ter outras classes principais dentro de um mesmo pacote, mas, às vezes, por questões de organização estrutural, pode ser mais viável criar um método principal por pacote.

Retornando ao exemplo, continuaremos o processo de herança, realizando o mesmo procedimento dentro da ClasseNeta:

Figura 5 | ClasseNeta

```
1 package heranca;  
2  
3 public class ClasseNeta extends ClasseFilha{  
4     public static void main(String[] args) {  
5  
6     }  
7 }
```

Fonte: elaborado pelo autor (2018).

É importante lembrar que, caso a classe pai esteja em outro pacote, é necessário importar o pacote para dentro da subclasse por meio da seguinte linha a ser adicionada no início do código:

```
Import nomeDoPacote.NomeDaClasse;
```

É possível realizar essa mesma operação, importando o pacote diretamente na linha de declaração da classe (usaremos como base a linha 3, da Figura 5):

```
Public class ClassePai extends nomeDoPacote.NomeDaClasse{
```

Retornando à estrutura da Figura 5, podemos perceber que não aconteceram muitas alterações, senão a inclusão do método principal na linha 4, que caracteriza essa classe como sendo a principal desse pacote, ou seja, elegível a ser executada.

EXEMPLIFICANDO

Suponha que você esteja desenvolvendo um sistema para uma padaria, que terá um módulo de cadastro de produtos com vários campos para inserção do nome do produto, categoria a que pertence, preço, fornecedor, código de barras etc. Se pararmos para analisar essa estrutura, perceberemos que numa mesma tela estão envolvidos assuntos distintos. Que tal tentarmos separá-los e organizá-los em categorias específicas?

Assuntos relacionados diretamente ao Produto:

- Nome.
- Preço.
- Código de Barras.

Assuntos relacionados indiretamente ao Produto:

- Categoria.
- Fornecedor.

Para não confundir sobre os itens que estão ou não relacionados diretamente a algum elemento, basta pensar da seguinte forma: esse item existiria sozinho, independente do elemento principal? Se a resposta for sim, então esse item deve ser desacoplado, senão deverá pertencer como item do elemento principal.

Tomando esse exemplo como base, podemos chegar à conclusão que um Fornecedor existe, independente do produto que ele comercialize. Uma categoria também existe dentro dos mesmos moldes. Já o preço de um produto precisa, obrigatoriamente, de um produto para que possa existir.

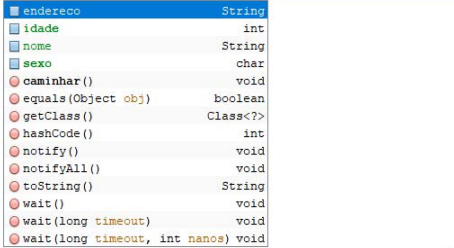
Ao realizar essa separação, é possível organizar cada elemento como uma classe de seu sistema. Nesse caso, teríamos as classes:

- Produtos (herdeira de categorias e de Fornecedores).
- Categorias (superclasse).
- Fornecedores (superclasse).

A essência da herança ocorrerá agora, no momento em que importaremos os recursos das classes Filha e Pai para dentro da classe Neta. Para que isso ocorra, é necessário realizarmos a instanciação da superclasse. Não precisamos instanciar a ClassePai; basta instanciar a ClasseFilha que, em decorrência da herança, já detém o acesso a todos os métodos e atributos da ClassePai, podendo passá-los adiante até a ClasseNeta. Vejamos como ficaria essa estrutura:

Figura 6 | Instanciando ClassePai e ClasseFilha dentro de ClasseNeta

```
1 package heranca;
2
3 public class ClasseNeta extends ClasseFilha{
4     public static void main(String[] args) {
5         ClasseFilha filha = new ClasseFilha();
6         filha.
7     }
8 }
9
10
```



Fonte: elaborado pelo autor (2018).

Na Figura 6, estamos dentro da ClasseNeta.Java. Na linha 5, realizamos a instância da ClasseFilha (que já estava herdando métodos e atributos da ClassePai). No momento da instanciação, criamos o objeto chamado *filha*. Na linha 6, chamamos o objeto *filha* e, em seguida, colocamos o *ponto final*, que, conforme já vimos, é a maneira utilizada em Java para acessar os métodos e atributos de uma classe.

Observe que ao colocarmos esse ponto, um quadro se abre (mediante pressionamento das teclas *CTRL+ Espaço* de forma simultânea). A lista de recursos disponíveis para uso é a composição acumulada de todos os métodos e atributos constantes na ClassePai e ClasseFilha, ou seja, conseguimos atingir o objetivo disposto na Figura 1, que era gerar uma herança de herança por meio de código Java.

Desse ponto em diante, o uso dos recursos do objeto, modificação dos atributos ou acesso aos métodos funcionam da mesma maneira, como já tratamos em capítulos anteriores.

ASSIMILE



Um método principal é um método genérico, responsável por iniciar a execução do aplicativo Java (DEITEL, 2010)³. De acordo com Deitel (2010), dentre os vários métodos de um programa Java, um deles deve, obrigatoriamente, se chamar *main*, para que a Máquina Virtual do Java (JVM) execute a aplicação. Via de regra, todos os métodos devem retornar algo após seu processamento; caso não haja nenhum retorno (retorno vazio), ocorre a declaração *void*, também presente na linha de declaração do método principal.

Saiba que, para que qualquer software Java seja executado, é necessário ter, ao menos, um método público chamado *main* em sua estrutura, descrito pela sintaxe: `public static`

³ DEITEL, Harvey M. Java Como Programar. São Paulo: Pearson Prentice Hall, 2010.

`void main(String[] args){}`. Adicionalmente, todo e qualquer conteúdo que venha a ser visível no programa, deve estar dentro do par de chaves destacados em **negrito**.

► 3. Herança com encapsulamento

No capítulo 4, vimos que o encapsulamento é uma técnica de proteção de acesso aos dados entre as classes. O encapsulamento sobrepõe os critérios de herança em POO, pois mesmo uma subclasse não consegue acessar os membros *private* de sua superclasse, tendo seu acesso restrito apenas aos membros que não sejam privados (DEITEL, 2010)⁴.

Dessa maneira, é possível compreender que, embora a base teórica da herança seja ceder acesso, essa regra não é absoluta, por estar submissa ao recurso de encapsulamento que é superior a ela.

Vamos fazer uma implementação em nossa **ClassePai**, apontando alguns atributos como privados e outros como protegidos para entendermos o que poderá ocorrer com a herança:

Figura 7 | Atribuindo encapsulamento privado e protegido aos atributos da ClassePai

```
1 package heranca;
2
3 public class ClassePai {           extends ClassePai {
4     private String nome;
5     protected String endereco;
6 }
```

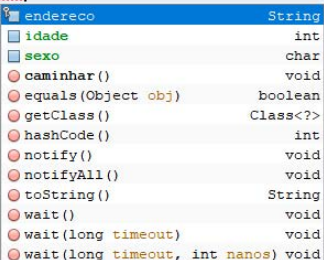
Fonte: elaborado pelo autor (2018).

Vejamos o que acontece ao tentar acessar novamente os atributos da ClassePai, por meio da ClasseNeta:

⁴ DEITEL, Harvey M. **Java Como Programar**. São Paulo: Pearson Prentice Hall, 2010.

Figura 8 | ClasseNeta sem acesso aos atributos privados da ClassePai

```
1 package heranca;
2
3 public class ClasseNeta extends ClassePai {
4     public static void main(String[] args) {
5         ClasseFilha filha = new ClasseFilha();
6         filha.
7     }
8 }
9
10
```



Fonte: elaborado pelo autor (2018).

Como especificado no capítulo 4, encapsulamento, os atributos privados não ficam disponíveis nem ao menos às classes herdeiras; deverão ser utilizados apenas pelas classes que os declararam. Já para os atributos protegidos, nesse caso em que estamos trabalhando com o mesmo pacote, o acesso fica disponível, embora uma *chave*, simbolizando a proteção, seja colocada ao lado do nome do atributo (na IDE *Netbeans*).

PARA SABER MAIS



Em POO existe o conceito de herança múltipla, que se caracteriza pelo fato de uma subclasse possuir mais de uma classe pai. Entretanto, essa é uma particularidade de algumas linguagens de programação. Java, por exemplo, aceita herança simples ou herança única, mas não aceita herança múltipla (DEITEL, 2010)⁵. Para recorrer às vantagens de herança múltipla, em Java, seria necessário criar o processo que fizemos anteriormente: propor uma estrutura baseada em herança de herança, em que uma ClassePai transfere seus métodos e atributos a uma ClasseFilha, que, por sua vez, os retransfere a uma subclasse (ClasseNeta).

⁵ DEITEL, Harvey M. Java Como Programar. São Paulo: Pearson Prentice Hall, 2010.

► 4. Resumindo o assunto herança

Com a herança dentro da Programação Orientada a Objetos, é possível ampliar as possibilidades ao se trabalhar com um conjunto específico de classes, mas é preciso evitar seu uso desordenado, considerando que uma das premissas da POO é justamente a organização e reutilização de código.

Realize alguns testes de herança e veja como as classes se comportam diante de diferentes cenários; aproveite para implementar recursos de encapsulamento, *getters* e *setters*.

QUESTÃO PARA REFLEXÃO

Pudemos observar, nesse capítulo, que a herança possibilita uma classe ter acesso aos métodos e atributos de outras classes das quais se derive. Vimos ainda que os critérios de encapsulamento estão acima dos critérios de herança. Com base nessas proposições, existiria algum cenário em que seria possível ter acesso aos métodos privados de uma superclasse, por meio de suas classes filhas? Justifique sua resposta.

► 5. Considerações Finais

- Superclasses também são chamadas de classe pai e delas pode ser derivada uma ou mais subclasses, chamadas também de classes-filhas.
- Em Java não é possível trabalhar com herança múltipla, mas é possível trabalhar com herança de herança.
- Métodos e atributos privados não podem ser acessados via herança,

ou seja, a nível de restrição de acesso, o encapsulamento é superior aos conceitos de herança.

► Glossário

- **Ambiguidade:** nesse caso, utilizamos a palavra ambiguidade com o sentido de *imprecisão*.

VERIFICAÇÃO DE LEITURA TEMA 6



1. A nível de código, forma correta de uma classe herdar dados de outra, na linguagem Java é:
 - a) `Public class Herdeira extends ClasseComMetodoPrincipal{}`.
 - b) `Public class ClassePrincipal extends ClasseHerdeira{}`.
 - c) `Public class Herdeira extends ClassePai(){}`.
 - d) `Public class ClassePrincipal extends Herdeira(){}`.
 - e) `Public class Herdeira extends ClassePai{}`.
2. Quanto ao encapsulamento de dados, podemos afirmar corretamente que:
 - a) Com o encapsulamento privado, uma subclasse tem acesso aos atributos e métodos da classe pai, desde que ambas estejam no mesmo pacote.
 - b) Separar as classes, em pacotes distintos, dificulta o encapsulamento e impede a implementação de

herança em Java.

- c) A herança pode ser realizada entre classes que estejam no mesmo pacote ou pacotes diferentes, desde que seja feita a importação do pacote para a subclasse.
- d) A palavra-chave para o uso de herança é sempre a mesma em qualquer linguagem de programação, pois isso é uma questão de boa prática de desenvolvimento, aprovada em convenção de programação.
- e) É possível realizar a herança de uma classe pai para uma classe filha ou de uma classe filha para uma classe pai. Esse conceito é denominado herança bilateral.

3. Sobre o método *main* em Java, escolha a assertiva correta:

- a) Ele é o método principal, que é executado sempre que uma classe é instanciada.
- b) Ele é um método que pode ter seus valores acessados a partir de qualquer subclasse, desde que estejam dentro de um pacote também chamado *main*.
- c) O método *main* caracteriza a classe como executável, ou seja, se implementado, possibilita que a máquina virtual do Java execute a aplicação.
- d) É um método que deve constar em todas as subclasse para identificá-las como derivadas de uma superclasse.
- e) É um método que executa uma aplicação e, geralmente, é a tela principal do sistema e é uma extensão da classe *MainContext()*.

Referências Bibliográficas

DEITEL, Harvey M. **Java Como Programar**. São Paulo: Pearson Prentice Hall, 2010.

FELIX, Rafael. **Programação Orientada a Objetos**. São Paulo: Pearson Education do Brasil Ltda., 2016.

Gabarito – Tema 6

Questão 1 - Alternativa E

A forma correta seria uma classe herdeira ser uma extensão de uma classe pai. Vale lembrar que essa classe pai também pode ser uma subclasse de outra classe pai, gerando o conceito de herança de herança.

Questão 2 - Alternativa C

É possível que uma classe acesse classes que estão em outros pacotes para referenciá-las como superclasses e herdar suas características e ações, mas para isso é necessário realizar a importação do pacote para que o código tenha acesso ao caminho completo da classe da qual se pretende derivar.

Questão 3 - Alternativa C

Como vimos, o método *main* é o que torna uma aplicação executável. Sem ele, uma aplicação Java não poderá ser executada pois não será reconhecida pela Máquina Virtual do Java (JVM).



TEMA 07

POLIMORFISMO

► Objetivos

- Desenvolver estruturas orientadas a objetos com utilização de polimorfismo.
- Aplicar as teorias polimórficas a sistemas, de forma coerente e eficaz.
- Compreender a aplicação do polimorfismo no conceito abstrato da Orientação a Objetos.

► Introdução

Este tópico é o penúltimo tema a ser abordado nessa caminhada pelo mundo da Programação Orientada a Objetos (POO). Sua colocação **não** faz jus à sua importância, pois, como veremos no decorrer da aula, o polimorfismo é uma estrutura muito importante, que permite implementar um método de N formas diferentes em uma ligação de herança (TUCKER, 2010)¹.

O polimorfismo é a junção de duas palavras: *poli*, que significa muitos ou muitas e *morfo*, que significa formas, assim, ao tratarmos de polimorfismo, estamos literalmente nos referindo a algo que tenha ou assuma várias formas (TUCKER, 2010).

A aplicação desse termo está, intrinsecamente, ligada aos conceitos de métodos e herança, que vimos em detalhes no capítulo 6. Dessa forma, recomendamos que sua base teórica esteja bem solidificada para que você consiga acompanhar com fluidez os conteúdos que serão apresentados.

Esse conteúdo será visto de maneira minuciosa, mas num resumo substancial, o polimorfismo permite que um mesmo método se comporte de maneiras diferentes em um mesmo método. Assim, você poderá aproveitar da estrutura de um método (operações, variáveis, funções etc.), mas para desempenhar papéis diferentes no escopo de seu algoritmo.

► 1. Entendendo o polimorfismo

Para conceituar o assunto, vamos iniciar já com um exemplo prático, a fim de dar um direcionamento aos demais tópicos que virão.

Pense na seguinte situação hipotética:

¹ TUCKER, Allen B. **Linguagens de Programação**: princípios e paradigmas. Porto Alegre: AMGH, 2010.

Diariamente, você vai ao trabalho de carro. Sempre segue o mesmo caminho, passando pelas mesmas paisagens, avenidas movimentadas, comércios, pontes e viadutos; pega os mesmos engarrafamentos, cruza com os mesmos agentes de trânsito e não raro, se depara sempre com os mesmos motoristas que traçam o mesmo trajeto. Determinado dia, você opta por mudar um pouco sua rotina e deseja ir de moto. O destino continua sendo o mesmo e até parte do caminho também, mas de moto você consegue realizar trajetos um pouco diferentes, encontrando assim, em seu caminho, novas paisagens, novos comércios, novos viadutos e talvez, nenhum trânsito. Empolgado com a mudança de hábitos, você deseja num terceiro dia, ir ao trabalho de bicicleta. Novamente o trajeto passa por uma mudança, pois você não pegará engarrafamentos, considerando que a ciclovia é contínua de sua casa até o seu trabalho; da mesma forma, a paisagem ao seu redor muda consideravelmente já que parte do trajeto ocorre entre parques e praças.

É possível perceber, nesse exemplo, que existem alguns pontos principais em comum: todos têm como objetivo a ida ao trabalho e todos envolvem a ação de pilotar algo (carro, moto e bicicleta). Como vimos em detalhes no capítulo 1 e 2, sempre que nos referimos a algum verbo, estamos tratando a respeito de um método, logo, temos o método *pilotar*, executado de várias formas diferentes. Polimorfismo!

Ao lidar com polimorfismo, é necessário que entendamos que algo que contém muitas formas precisa manter a sua essência e modificar apenas detalhes que o caracterizam, ou seja, “polimorfismo é a propriedade que permite que um operador ou uma função atue de modo diferente em função do objeto sobre o qual se aplicam” (AGUILAR, 2011, p. 27)². No exemplo citado, a ação de *sair de sua casa, pilotar e chegar ao trabalho*, representam a essência; já as características ficam por conta das peculiaridades do trajeto, de acordo com o tipo de veículo escolhido.

² AGUILAR, Luis Joyanes. **Programação em C++**: algoritmos, estruturas de dados e objetos. Porto Alegre: AMGH, 2008.

Dessa forma, é mandatório que a essência de seu método seja mantida para que o polimorfismo exista. Em POO, mantém-se essa essência por meio da imutabilidade de nome do método, ou seja, métodos diferentes podem ter o mesmo nome, porém formas diferentes. Analise a imagem abaixo para compreender a ideia melhor:

Figura 1 | Polimorfismo

```
1 package polimorfismo;
2
3 public class Teste {
4     public void ligar() {
5     }
6
7     public void ligar() {
8     }
9 }
```

Fonte: elaborado pelo autor (2018).

O conceito de polimorfismo seria o apresentado na Figura 1, ou seja, significa mais de um método (*ligar*) com o mesmo nome. Entretanto, algo está errado nessa estrutura, pois, como vimos em nosso exemplo, a *essência* do método se mantém, mas suas características devem ser alteradas. Essa mudança nas características é a única forma de um método não se sobrepor ao outro: terão o mesmo nome, agirão até de maneiras similares, mas serão independentes um do outro. Fazendo alusão ao exemplo trabalhado, você sairia de sua casa, iria ao mesmo emprego e pilotaria algum veículo, mas tanto o trajeto quanto o veículo seriam diferentes.

Na Figura 1 é possível vermos, na linha 7, um aviso de erro, que, se for expandido (no *Netbeans* apresentará a seguinte mensagem: *method ligar() is already defined in class Teste* (o método *ligar()* já está definido na classe *Teste*), ou seja, não é possível definir dois ou mais métodos com o mesmo

nome, exceto se possuírem alguma característica que os tornem únicos. A essa característica, damos o nome de assinatura.

LINK



Nesse link, do Instituto Federal de Santa Catarina, você poderá ver uma explicação adicional ao conceito de polimorfismo. Disponível em: <<https://wiki.sj.ifsc.edu.br/wiki/images/e/e2/Heran%C3%A7a20151.pdf>>. Acesso em: 22 out. 2018.

► 2. Assinaturas de métodos

A assinatura de um método irá diferenciá-lo de outros métodos que tenham o mesmo nome dentro da mesma classe.

Essa assinatura está disposta dentro dos parênteses do método, propriamente nos tipos e quantidades de seus parâmetros. Isso significa que são dois métodos com o mesmo nome, mas com parâmetros diferentes (nem que seja ao menos no tipo), já possuem assinaturas diferentes, tornando-se únicos perante a estrutura do sistema.

Vejamos como ficaria a adaptação dos nossos métodos *ligar()*, aplicando parâmetros distintos. Vamos simular que no primeiro método estamos lidando com um número de telefone, ou seja, nossa intenção é receber um número inteiro. Já no nosso segundo método, estamos lidando com um valor *booleano*, ou seja, receberemos como parâmetro um valor verdadeiro ou falso:

Figura 2 | Polimorfismo. Assinaturas de métodos

```
1 package polimorfismo;
2
3 public class Teste {
4     public void ligar(int numero) {
5     }
6
7     public void ligar(boolean status) {
8     }
9 }
```

Fonte: elaborado pelo autor (2018).

Como podemos observar, mesmo que os métodos tenham o mesmo nome e até a mesma quantidade de parâmetros, os tipos de cada parâmetro criam uma assinatura totalmente diferente para cada um, diferenciando-os e os tornando únicos.

ASSIMILE

Na Figura 2, utilizamos como exemplo dois tipos de dados: inteiro e booleano. Haveria a possibilidade de ambos os parâmetros receberem o mesmo nome (numero/ status) e, ainda assim, seriam diferentes, pois os tipos de dados escolhidos não seriam iguais. Lembre-se de que só é possível setar assinaturas diferentes para métodos polimórficos, por meio de:

- Quantidade de parâmetros recebidos.
- Tipos de parâmetros recebidos.
- Ordem de exibição desses tipos.

Basta uma pequena mudança entre os parâmetros e isso já caracterizará uma nova assinatura. Veja na Figura 3, a seguir, a similaridade dos parâmetros, salvo pelo tipo do último parâmetro recebido:

Figura 3 | Polimorfismo. Assinaturas de métodos

```
1 package polimorfismo;
2
3 public class Teste {
4     public void ligar(int a, int b, String c, char d){
5     }
6
7     public void ligar(int a, int b, String c, byte d){
8     }
9 }
```

Fonte: elaborado pelo autor (2018).

No primeiro método *ligar()* temos 4 parâmetros, dos tipos *int*, *int*, *String* e *char*, respectivamente. Já no segundo método *ligar()*, temos também 4 parâmetros, do tipo *int*, *String* e *byte*, respectivamente. Apenas o último parâmetro já é suficiente para modificar a assinatura do método.

PARA SABER MAIS



A assinatura de um método é caracterizada pela quantidade de parâmetros e seus respectivos tipos. O nome do parâmetro em si não define uma assinatura, ou seja, dois métodos com o mesmo nome, que recebam parâmetros de tipos iguais e nomes diferentes, resultaram no mesmo erro disposto na Figura 1, informando que aquele parâmetro já existe.

Dessa maneira, sua atenção deve estar voltada para não repetir a quantidade de parâmetros e os tipos de dados; o nome do parâmetro

Figura 4 | Polimorfismo. Assinaturas de método com erro por similaridade

```
1 package polimorfismo;
2
3 public class Teste {
4     public void ligar(int a, int b, String c, char d){
5     }
6
7     public void ligar(int a, int b, String c, char e){
8     }
9 }
```

Fonte: elaborado pelo autor (2018).

Os conceitos de retorno que cada método terá (inclusive o retorno vazio), bem como o escopo a ser executado pelo método, não são considerados para composição da assinatura. Assim, toda a estrutura do método pode ser idêntica para os métodos envolvidos no polimorfismo, desde que a estrutura dos parâmetros seja distinta:

Figura 5 | Polimorfismo. Estruturas idênticas, com parâmetros distintos

```
1 package polimorfismo;
2
3 public class Teste {
4     public void ligar(int a){
5         System.out.println(a);
6     }
7
8     public void ligar(String a){
9         System.out.println(a);
10    }
11 }
```

Fonte: elaborado pelo autor (2018).

► 3. Tipos de polimorfismo

Os tipos habituais de polimorfismo são o de sobrecarga de métodos e sobreposição de métodos. Para fins didáticos, essa aula será dividida em dois capítulos para não deixar o conteúdo demasiadamente denso. Falaremos, nesse capítulo, sobre o Polimorfismo de Inclusão, que abrirá as portas ao tema e, no capítulo 8, trataremos dos outros três tipos.

3.1 Polimorfismo de Inclusão

Esse tipo de polimorfismo ocorre quando existe uma classe pai, com um determinado método e uma classe filha, com o mesmo método e com a mesma assinatura, ou seja, a mesma quantidade de parâmetros e todos os parâmetros do mesmo tipo (na mesma ordem). Por questões organizacionais, e também por diretivas específicas do Java, teremos três arquivos envolvidos:

- Uma classe principal (Polimorfismo.java), responsável por executar a aplicação como vimos no capítulo anterior.
- Uma classe Pai (Pai.java), que conterá apenas um método em seu escopo.
- Uma classe Filha (Filha.java), que também conterá apenas um método em seu escopo, com o mesmo nome e assinatura do método disposto na classe Pai.

A estrutura dos arquivos deverá estar dessa maneira:

Figura 6 | Polimorfismo de inclusão. Estrutura de arquivos



Fonte: elaborado pelo autor (2018).

Dentro da classe Pai, iremos inserir o método Acao(), responsável por imprimir na tela a seguinte frase: *MÉTODO PAI*.

Figura 7 | Polimorfismo de inclusão. Classe Pai

```
1 package polimorfismo;
2
3 public class Pai {
4     protected void Acao() {
5         System.out.println("MÉTODO PAI");
6     }
7 }
```

Fonte: elaborado pelo autor (2018).

Agora cederemos a herança da classe Pai à classe Filha, por meio da palavra reservada `extends`; adicionalmente vamos inserir o método Acao() em seu escopo.

Figura 8 | Polimorfismo de inclusão. Método Filha

```
1 package polimorfismo;
2
3 public class Filha extends Pai{
4     /**
5      * Aula sobre Polimorfismo: Inclusão
6      */
7     @Override
8     public void Acao() {
9         System.out.println("MÉTODO FILHA");
10    }
11 }
```

Fonte: elaborado pelo autor (2018).

A princípio, pode parecer estranho esses elementos similares a comentários nas linhas 4, 5 e 6, bem como a chamada na linha 7. Entretanto, em Java, esses elementos são normais e serão requeridos pela própria IDE,

no momento em que se utilizar algum recurso que os requeira, como no caso, uma sobreposição (*override*).

PARA SABER MAIS



Em Java, O `@Override` funciona como espécie de alerta que auxiliará a não cometer erros lógicos de estrutura. Sempre que um bloco é precedido pela notação `@Override`, o próprio Java se encarrega de compreender que as instruções a seguir funcionarão como uma sobreposição de outra ação, ou seja, sempre que o método sobrescrito for invocado, o Java invocará, na verdade, o novo método que o substituiu, evitando incoerência na estrutura do código e em seu funcionamento como o esperado.

Na linha 4, temos uma instrução do tipo Javadoc, que é uma documentação automática gerada e que utiliza como base alguns elementos como as instruções Javadoc iniciadas como uma linha de comentário (`/*... */`). Para que possam compor a documentação (Javadoc), esses comentários devem receber um asterisco adicional, logo após o asterisco de abertura. Geralmente, trechos que causem interferências em outras áreas do código, como uma sobreposição de métodos, são dignos de serem inseridos no Javadoc para que qualquer desenvolvedor, que venha a realizar manutenções no código, consiga compreender a estrutura da aplicação.

A linha 5 (e outras que pudessem ser geradas abaixo dela), seriam usadas para apontar os comentários propícios. Adicionalmente, na linha 7, temos a declaração de sobreposição de um método já existente. Essa informação também constará na documentação automática do Javadoc.

Para gerar o Javadoc, no *Netbeans*, basta clicar com o botão direito sobre o nome do projeto e escolher a opção *Gerar Javadoc*. O *browser* abrirá com

a documentação automática do Java para sua aplicação. Acessaremos a classe em questão (Filha) e lá teremos as informações que inserimos, ou seja, comentários e *override*:

Figura 9 | Polimorfismo: Geração de Javadoc



Fonte: elaborado pelo autor (2018).

LINK

Par obter mais informações sobre o Javadoc e sua aplicabilidade, consulte a documentação oficial no site da Oracle. Disponível em: <<https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>>. Acesso em: 22 out. 2018.

Agora que já temos uma classe *Pai*, com um método *Acao()*, e uma classe filha com um método de mesmo nome, acessaremos a classe principal (polimorfismo.java) e criaremos uma extensão da classe filha. Dessa forma, a classe principal será filha da classe filha (ou neta da classe pai).

Atente que essa estrutura é opcional, foi feita por questões meramente organizacionais do código para que você pudesse ter a mesma estrutura de arquivos trabalhadas até aqui.

Dentro da classe principal, criaremos uma instância das classes Pai e Filha, de forma que consigamos acessar seus métodos e atributos (no caso, apenas o único método, *Acao*, criado). Em seguida, invocaremos o método *Acao()* de cada uma das classes, de forma que possam imprimir seus valores em tela. Nossa estrutura ficará dessa forma:

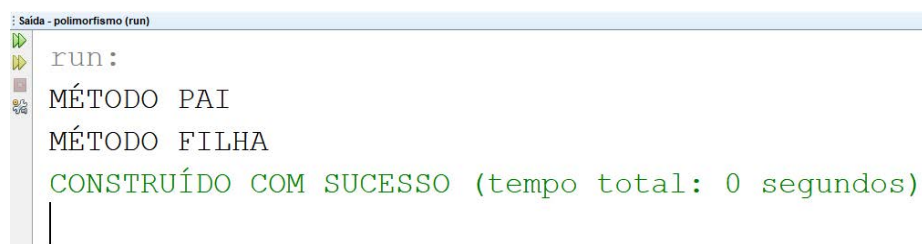
Figura 10: | Polimorfismo: Classe principal instanciando classes superiores

```
1 package polimorfismo;
2 public class Polimorfismo extends Filha{
3
4     public static void main(String[] args) {
5         Filha filha = new Filha();
6         Pai pai = new Pai();
7
8         pai.Acao();
9         filha.Acao();
10    }
11
12 }
```

Fonte: o autor (2018).

Já quanto aos resultados, teremos essa saída no console:

Figura 11 | Polimorfismo: Processamento de classe principal



```
run:
MÉTODO PAI
MÉTODO FILHA
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Fonte: elaborado pelo autor (2018).

O momento do polimorfismo de inclusão se dará agora, quando atribuiremos a instância de pai à instância de filha, ou seja, informaremos à aplicação que o objeto *pai* deverá receber os dados do objeto *filha*. Dessa forma, o método ora armazenado em *pai*, simplesmente será substituído

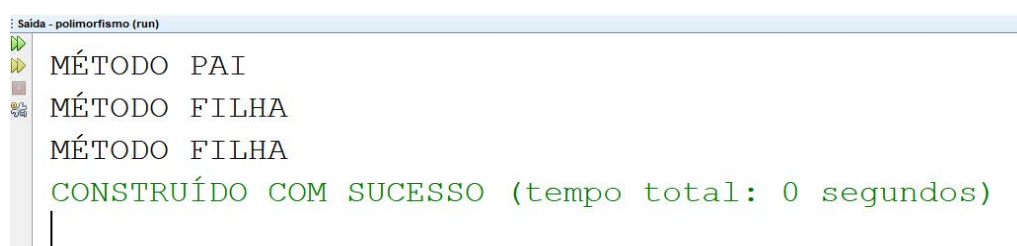
pelo método armazenado em *filha*. Confira a estrutura na Figura 12 e a saída na Figura 13.

Figura 12 | Polimorfismo: Inclusão

```
1 package polimorfismo;
2 public class Polimorfismo extends Filha{
3
4     public static void main(String[] args) {
5         Filha filha = new Filha();
6         Pai pai = new Pai();
7
8         pai.Acao();
9         filha.Acao();
10        pai = filha;
11        pai.Acao();
12    }
13 }
```

Fonte: elaborado pelo autor (2018).

Figura 13 | Polimorfismo: Saída do console com polimorfismo de Inclusão



```
Saída - polimorfismo (run)
MÉTODO PAI
MÉTODO FILHA
MÉTODO FILHA
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Fonte: elaborado pelo autor (2018).

Note que na linha 11, da Figura 12, estamos imprimindo o método pai, que, anteriormente, tinha como resultado de saída a frase: *MÉTODO PAI*. Após o polimorfismo de inclusão, o objeto assumiu o valor *MÉTODO FILHA* para o método, sobrepondo o valor anterior.

Ainda que a ideia possa parecer um tanto complexa, é considerada a forma mais simples de construção polimórfica (MANZANO, 2014)³, onde a

³ MANZANO, José Augusto N. G. **Programação de computadores com Java**. São Paulo: Editora Érica, 2014.

herança ocorre, mas em sentido inverso daquele que estamos habituados a trabalhar, ou seja, a classe pai acaba por herdar algum recurso vindo da classe filha.

QUESTÃO PARA REFLEXÃO

O uso do polimorfismo é um recurso extremamente útil no paradigma da Programação Orientada a Objetos, principalmente, pelo advento da economia de código, possibilitando reaproveitar trechos de outros métodos sem precisar escrevê-los a partir do zero.

Conhecemos em mais detalhes um dos tipos de polimorfismo, chamado de Polimorfismo de inclusão, que é a instância de um recurso da classe filha sobre a classe pai.

Com essa base, você conseguiria exemplificar um cenário onde o polimorfismo de Inclusão seria aplicado em uma aplicação do mundo real?

► 4. Considerações Finais

- Polimorfismo significa que um mesmo método pode assumir várias formas dentro de uma mesma estrutura de aplicação.
- Existem quatro tipos de métodos polimórficos: Inclusão, Paramétrico, Sobrecarga e Sobreposição.
- Polimorfismo de Inclusão permite que um recurso de uma classe filha instancie um recurso sobre a classe pai.
- Métodos polimórficos são diferenciados por meio de sua assinatura, que compreende os tipos, quantidade e ordem de seus parâmetros.

Glossário

- **Booleano:** é um tipo de dado (*boolean*) que tem representação binária. Um dado do tipo *boolean* não pode ter nenhum outro tipo de saída diferente de verdadeiro (*true*) ou falso (*false*).
- **Byte:** é um tipo de dado primitivo, numérico, do Java.

VERIFICAÇÃO DE LEITURA

TEMA 7

1. A assinatura de um método polimórfico é registrada por meio de:
 - a) Seus parâmetros.
 - b) Seus tipos de dados.
 - c) Sua sequência de atributos.
 - d) Seus objetos.
 - e) Sua classe pai.
2. É possível que dois ou mais métodos tenham a mesma assinatura, caso:
 - a) A quantidade de atributos, seus referidos tipos e seus métodos sejam exatamente iguais em todos os elementos envolvidos.
 - b) A quantidade de parâmetros, sua sequência e seus tipos de dados forem exatamente iguais em todos os métodos envolvidos.
 - c) Todos os tipos de dados forem exatamente idênticos entre os elementos envolvidos, mesmo que os nomes de seus parâmetros sejam distintos.
 - d) A ordem de seus parâmetros seja similar, mesmo que

os tipos de dados sejam diferentes, pois essa não é uma exigibilidade para caracterizar uma assinatura igual à outra.

e) A quantidade de parâmetros seja vazia em todos os elementos envolvidos.

3. No polimorfismo por inclusão, quando ocorre a sobreposição de um valor em um método da classe pai por um método da classe filha?

a) Quando a classe filha é instanciada e os valores de seus métodos são apontados.

b) Quando a classe pai é instanciada e apresenta seus métodos para serem modificados.

c) Quando um objeto da classe pai recebe por atribuição um objeto da classe filha, cujos métodos de ambas as classes envolvidas tenham a mesma assinatura.

d) Quando os métodos das classes envolvidas possuem a mesma assinatura e são instanciados no mesmo arquivo.

e) Quando os métodos das classes envolvidas possuem a mesma assinatura e são instanciados em arquivos diferentes.

Referências Bibliográficas

AGUILAR, Luis Joyanes. **Programação em C++**: algoritmos, estruturas de dados e objetos. Porto Alegre: AMGH, 2008.

MANZANO, José Augusto N. G. **Programação de computadores com Java**. São Paulo: Editora Érica, 2014.

TUCKER, Allen B. **Linguagens de Programação**: princípios e paradigmas. Porto Alegre: AMGH, 2010.

Gabarito – Tema 7

Questão 1 - Alternativa A

O que identifica a assinatura de um método polimórfico é a estrutura de seus parâmetros, que pode ser vazia ou não.

Questão 2 - Alternativa B

Para que uma assinatura de um método seja igual a de outro método é necessário que a quantidade de parâmetros, sua ordem e seus respectivos tipos, sejam idênticos entre todos os elementos envolvidos.

Questão 3 - Alternativa C

A sobreposição ocorre no momento em que um objeto da classe pai recebe a atribuição de um objeto da classe filha. Exemplo: Pai = Filha. Essa atribuição faz com que os métodos de mesma assinatura da classe pai sejam substituídos pelos métodos da classe filha.



TEMA 08

POLIMORFISMO DE SOBRECARGA OU REESCRITA (SOBREPOSIÇÃO) DE MÉTODOS

► Objetivos

- Ser capaz de implementar recursos de sobrecarga e reescrita de métodos.
- Compreender a importância da sobrecarga de métodos.
- Diferenciar sobrecarga de reescrita.
- Compreender o uso do polimorfismo paramétrico e sua ligação com o polimorfismo de sobrecarga.

Introdução

O tópico de polimorfismo é, sem dúvida, um dos mais complexos de se compreender dentro do tema de Programação Orientada a Objetos e, por esse motivo, ganhou duas seções dentro da Leitura Fundamental, a fim de que pudesse ser explicado de forma pautada e com ampla possibilidade de exemplos facilitadores de compreensão.

Nessa unidade, trataremos especificamente dos últimos três tipos de polimorfismo, ainda que um deles, o paramétrico, já tenha sido indiretamente mencionado na unidade 07, ao tratarmos do assunto assinatura de métodos. Você pode observar, inclusive, que, embora falemos de três tipos de polimorfismos, o título dessa unidade comporta apenas dois deles e entenderemos o motivo logo mais. Observe que existe muita similaridade entre alguns conteúdos já vistos e os polimorfismos abordados nessa unidade e, de fato, estão intrinsecamente ligados. Tratamos nessa unidade de atribuir a nomenclatura científica oficial a cada um desses elementos, vistos de maneira isolada anteriormente.

Esse tema envolve toda uma base conceitual lógica sobre o paradigma da POO, assim, foi deixado como último elemento de maneira proposital, pois é esperado que você, aluno, ao chegar nesse ponto do aprendizado, tenha adquirido toda a bagagem de conhecimento necessária para assimilar o conjunto de informações transferidas, sobretudo, sua aplicabilidade a nível de linguagem de programação.

Continuaremos com o uso da linguagem Java, mas incentivamos você a ter autonomia para explorar os mesmos recursos em outras linguagens como C#, C++, PHP, Python. Isso expandirá seu campo de visão quanto às possibilidades que a POO pode oferecer, além de prover a experiência de trabalhar com uma mesma lógica implementada em sintaxes distintas.

Tratando especificamente do polimorfismo, recomendamos não apenas ler esse conteúdo ou fazer os exercícios propostos, mas, também de

forma autônoma, tentar implementar situações-problema para executar esse recurso a nível de código.

1. Polimorfismo de sobrecarga

O polimorfismo do tipo sobrecarga, como o próprio nome diz, impõe ao método a responsabilidade de ter várias implementações distintas atreladas a si. Dessa forma, o método, que já tinha um conjunto de ações a desempenhar, assume N conjuntos adicionais como carga adicional de tarefas, o que justifica seu nome.

O que diferenciara um método de outro, com o mesmo nome, é a assinatura que contém. Vale criar uma ressalva ao conteúdo já visto na unidade 07: a assinatura do método é o conjunto de parâmetros que um determinado método recebe, levando em consideração apenas os tipos de seus dados, independente dos nomes das variáveis envolvidas.

Por ter o mesmo nome, um método invocado transferirá ao compilador a responsabilidade de escolher qual o método mais adequado de acordo com cada situação, conforme nos esclarece Furgeri (2015)¹:

A linguagem Java permite que vários métodos sejam definidos com o mesmo nome, desde que eles tenham uma assinatura diferente, ou seja, essas diferenças podem ser com base no número, nos tipos ou na ordem de parâmetros recebidos. Quando um método sobrecarregado é chamado, o compilador avalia e seleciona o método mais adequado à situação, examinando a assinatura correspondente, portanto os métodos sobrecarregados são utilizados para a realização de tarefas semelhantes sobre tipos de dados diferentes (FURGERI, 2015, p. 96).²

¹ FURGERI, Sérgio. **Java 8-ensino didático**: desenvolvimento e Implementação de aplicações. São Paulo: Érica, 2018.

² FURGERI, Sérgio. **Java 8-ensino didático**: desenvolvimento e Implementação de aplicações. São Paulo: Érica, 2018.



EXEMPLIFICANDO

Considere esse cenário, apresentado na Figura 1. O método `emitirSom` estará pronto para receber parâmetros vindos de alguma outra parte do código (seja uma classe externa ou a própria classe que o contém). Em determinado local do código, realizamos a declaração do método informando um valor do tipo inteiro, por exemplo: `emitirSom(17)`; o compilador executará esse comando e enviará o valor “17” como parâmetro ao método `emitirSom`, que observará a duplicidade de nomes e então encaixará o número 17 no método que puder receber dados do tipo inteiro (considerando que o número 17 é um inteiro).

A mesma lógica ocorreria caso a declaração possuísse essa outra estrutura: `emitirSom(true)`, porém, o método que seria alimentado seria aquele disposto na linha 8, da Figura 1, pois o parâmetro informado refere-se a um dado do tipo *boolean*. Adicionalmente, é importante ressaltar que, se porventura houvesse uma declaração do tipo `emitirSom(“teste”)`, uma exceção de erro seria imediatamente invocada, pois no escopo desenvolvido nenhuma assinatura está preparada para receber dados do tipo *String*.

Vejamos, na prática, como esta estrutura ficaria em um código de programação Java:

Figura 1 | Polimorfismo. Sobrecarga de métodos

```
1 package situacao.problema;  
2  
3 public class Cachorro extends Animal{  
4  
5     public Cachorro() {  
6     }  
7  
8     public void emitirSom(boolean a) {  
9         boolean latir = a;  
10    }  
11  
12    public void emitirSom(int a) {  
13        int alturaDoLatido = a;  
14    }  
15 }
```

Fonte: elaborado pelo autor (2018).

Conforme demonstrado na Figura 1, temos dois métodos com o mesmo nome **emitirSom** (linhas 8 e 12) e, ainda assim, o compilador não acusa erro por duplicidade. Isso ocorre porque os métodos possuem assinaturas diferentes entre si.

Torna-se claro que transferimos ao compilador da linguagem, a responsabilidade de escolher o melhor método a ser executado de acordo com o parâmetro recebido, cabendo ao programador apenas a responsabilidade de compor a estrutura desses métodos e zelar para que os dados enviados como parâmetros se encaixem em algum dos padrões esperados.

Observe que, ao tratarmos a respeito do polimorfismo de sobrecarga, o elemento que mais se destaca é a assinatura do método. Se observarmos com mais minúcia, o grande agente desse tipo de polimorfismo é propriamente o tipo de parâmetro que ele recebe, pois é por meio de seus parâmetros que o compilador discernirá entre um método e outro. Dessa forma, ao abordamos o polimorfismo de sobrecarga, estamos criando um link direto com o polimorfismo paramétrico.

ASSIMILE



Embora sejam adotadas duas nomenclaturas meramente conceituais (e a maioria das bibliografias acaba não abordando ambas), o polimorfismo de sobrecarga e o polimorfismo

paramétrico podem ser considerados como sendo apenas um, pois eles se completam mutuamente e um não existiria sem o outro.

Como um exemplo extra, temos a Figura 2, que implementa dentro da classe **Teste** dois métodos chamados **ligar()**: um do tipo inteiro, para registrar o número de um telefone; e outro do tipo *booleano*, para registrar uma resposta verdadeira ou falsa, como, por exemplo, ligar uma lâmpada.

Figura 2 | Polimorfismo de sobrecarga/ paramétrico

```
1 package polimorfismo;
2
3
4 public class Teste {
5     public void ligar(int numero){
6         System.out.println("Foi preenchido o número de telefone:" + numero);
7     }
8
9     public void ligar(boolean numero) {
10         System.out.println("Foi preenchido o valor booleano: " + numero);
11     }
12 }
```

Fonte: elaborado pelo autor (2018).

Mediante a execução do código, o sistema imprimirá uma mensagem na tela com o número de telefone que o método recebeu ou preencherá o valor *booleano* (ou 0 ou 1).

Por fim, vale ressaltar que no polimorfismo de sobrecarga, os métodos devem pertencer sempre à classe na qual são sobrecarregados.

► 2. Polimorfismo de sobreposição

Conforme abordado na unidade 07, existe a possibilidade de um método se sobrepor a outro com o mesmo nome. Nesse caso, essa sobreposição

criará implementações que o método inicial não possuía. Sobreposição pode ser considerada também como reescrita de métodos.

Na linguagem Java, ao sobrepormos um método, precisamos, necessariamente, apontar a notação `@Override`, ou seja, precisamos informar que estamos utilizando de forma intencional um nome de método já existente e que estamos cientes de que aquele método irá sobrepor-se a algum outro. Diferente do polimorfismo de sobrecarga, a sobreposição referencia-se a algum método que não esteja na mesma classe na qual ele é sobreposto.

É importante salientar que cada linguagem tem sua própria característica quanto à notação de sobreposição e caberá a você, como programador, localizar e adequar sua estrutura de sintaxe ao padrão da linguagem escolhida. Um bom exemplo disso é a linguagem *Kotlin* (amplamente utilizada como meio paliativo ao desenvolvimento em Java), cuja notação de sobreposição ocorre de forma explícita na própria linha de código:

Figura 3 | Notação Override em Kotlin

```
open class A {  
    open fun foo(i: Int = 10) { ... }  
}  
  
class B : A() {  
    override fun foo(i: Int) { ... } // no default value allowed  
}
```

Fonte: adaptado de <<https://www.kotlinlang.org/>> (2018). Acesso em: 22 nov. 2018.

LINK



Acesse o site da linguagem *Kotlin*, que vem ganhando muito espaço no mercado devido à sua adaptabilidade ao Java, porém com sintaxe amplamente mais compacta. Veja mais detalhes sobre a notação de sobreposição na página, em inglês, disponível em: <<https://kotlinlang.org/docs/reference/functions.html>>. Acesso em: 22 nov. 2018.

Para que consigamos compreender a funcionalidade da sobreposição, precisamos abordar o tema implementação. Implementar algo, de acordo com o dicionário, significa “por em execução; fazer o implemento de; efetuar, executar, fazer” (MICHAELIS, 2018)³. Dessa forma, ao tratarmos a respeito de implementação, estamos nos referindo de maneira direta ao ato de executar ou fazer o implemento de algo que já existe.

Por padrão, ao trabalharmos com implementações, usamos o mesmo conceito trabalhado nas unidades iniciais, quando tratamos a respeito das classes abstratas. Essas classes tinham o papel de prover um escopo padrão, pré-definido, para que suas subclasses criassem características mais específicas, implementando-as.

Com os métodos, esse conceito funciona de maneira similar: cria-se uma implementação que tem como objetivo atribuir novas funcionalidades a um método já existente ou preencher parâmetros que o método inicial tenha deixado propositalmente em aberto.

PARA SABER MAIS



Seguindo a linha de desenvolvimento Java, recomendamos a leitura do tema Atividades (ou Activities) do Google Android, na documentação oficial da linguagem. Este tema abordará os conceitos sobre implementação e auxiliará a compreender sua utilidade quando aplicada a métodos.

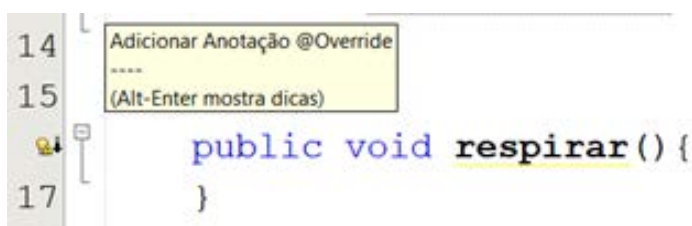
Em geral, é possível sobrepor métodos padrões do Java ou métodos criados pelo próprio programador. O conceito final acaba por ser o mesmo, embora em Java, o mais comum seja a sobreposição de métodos próprios (enquanto em Java Android, por exemplo, o mais comum seja a sobreposição de métodos nativos).

³ MICHAELIS, Dicionário Brasileiro da Língua Portuguesa. Disponível em <<http://michaelis.uol.com.br/busca?i-d=ZNQp2>>. Acesso em: 10 nov. 2018.

Na prática não há diferença entre os dois tipos de sobreposição; salvo que na sobreposição de métodos nativos, o programador precisa conhecer o escopo do método para que os dados a serem implementados se acoplem com aqueles esperados.

Em Java, ao tentarmos sobrepor um método de alguma classe, que seja uma superclasse da classe atual, receberemos um aviso sobre essa tentativa, conforme observa-se na Figura 4:

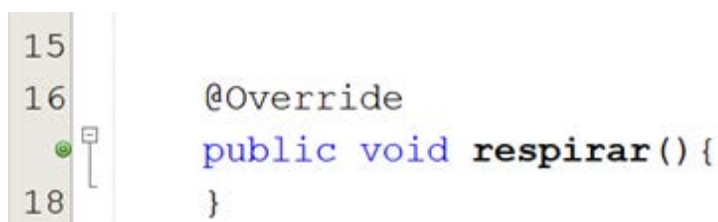
Figura 4 | Aviso quanto à sobreposição de método



Fonte: elaborado pelo autor (2018).

Esse aviso tem como objetivo alertar o programador que está prestes a sobrepor um método já existente. Após a inserção do `@Override`, o erro é sanado e um símbolo (dependendo da IDE utilizada) é apontado na linha da sobreposição, informando que aquele elemento implementa um outro método já existente:

Figura 5 | Método sobreposto



Fonte: elaborado pelo autor (2018).

PARA SABER MAIS



Na IDE *Netbeans* ou *Android Studio* (caso você esteja realizando testes com dispositivos móveis ou na linguagem *Kotlin*), é possível navegar entre os métodos que foram sobrepostos

para ver sua estrutura e entender seu funcionamento. No *Netbeans*, basta clicar sobre o símbolo (linha 17, da Figura 5) ou se preferir, paire o cursor do mouse em qualquer área do método e pressione CTRL+SHIFT+P. Já no *Android Studio*, basta pressionar CTRL e clicar sobre o método que deseja visitar.

Em geral, o polimorfismo de sobreposição é o mais utilizado que a sobrecarga ou inclusão. É bem claro que seus conhecimentos devem englobar todas as formas possíveis para se trabalhar com métodos, porém, a nível prático, a sobrecarga acaba por ser, a mais utilizada, seguida pela sobreposição e depois pela inclusão.

SITUAÇÃO-PROBLEMA

Para que você possa colocar em prática todo o conteúdo aprendido durante essa disciplina, propomos uma situação-problema, que envolverá todos os conceitos aprendidos ao longo dessas oito unidades.

Para que você consiga desenvolver a situação com eficácia, será necessário acompanhar todos os materiais de Leitura Fundamental, assistir às vídeo-aulas e tentar responder às questões propostas, pois assim você alcançará o nível de conhecimento requerido para o desenvolvimento dessa atividade.

Embora você tenha a opção de resolver a situação com qualquer linguagem de sua escolha, recomendamos fortemente que você utilize a linguagem Java, pois, além de ser uma linguagem nativamente Orientada a Objetos, foi a linguagem utilizada durante nossas aulas, e isso dará um respaldo maior na hora do desenvolvimento.

Pressuponha que você esteja desenvolvendo uma aplicação para um canil de sua cidade. Você precisará criar uma estrutura que seja capaz de armazenar, de maneira eficiente, os dados desses animais. Considerando que esses cachorros são animais, talvez seja interessante pensar numa hierarquia de classes, de forma que o código possa ser reutilizado futuramente.

A (s) classe (s) que você criar precisará ter, no mínimo, os atributos:

- Nome.
- Se o animal é mamífero ou não.
- Seu *habitat* (urbano ou rural).

A (s) classe (s) deverá (ão) ainda ter (em) um método chamado *respirar*.

De forma a deixar seu código conciso, lembre-se de pré-alimentar a (s) classe (s) com algum conteúdo padrão, caso algum objeto seja instanciado sem informar parâmetros, mas deixe-a pronta também para receber os parâmetros de um objeto recém-criado.

Como alguns cachorros sabem nadar, você deverá criar um método chamado *nadar* do tipo *booleano* (sim ou não), sobrecarregado por um outro método do tipo *String*, que informará o estilo de nado que o cachorro sabe fazer.

Para que as informações possam ser exibidas em tela, será necessário criar um método que faça a impressão desses dados. Para isso, deverá existir um método denominado *imprimirDados*.

Em sua classe principal, você deverá exibir todos os dados do cachorro, além de alimentar o método *nadar* das duas maneiras possíveis (com atributo *booleano* e atributo *String*).

Dica: lembre-se dos critérios de segurança em seu código; você não deverá acessar ou modificar diretamente os seus atributos, utilize um intermediador.

Ao executar as ações acima, o resultado a ser exibido deverá ser similar à Figura 6, colocada a seguir (não é obrigatório utilizar essa mesma estrutura; é apenas uma sugestão).

Figura 6 | Sugestão para resultado da Situação problema

```
run:
Nosso cachorro se chama Doguinho
Ele é mamífero
Seu habitat é urbano
Ele nada no estilo 'Cachorrinho'
Que legal que o Doguinho sabe nadar!
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Fonte: elaborado pelo autor (2018).

QUESTÃO PARA REFLEXÃO

Considerando o conteúdo sobre polimorfismo, você consegue exemplificar um cenário para a aplicação de cada um dos tipos que vimos, entre as unidades 07 e 08? Tente utilizar exemplos que não foram abordados nas leituras fundamentais ou nas vídeo-aulas.

► 4. Considerações Finais

- Sobrecarga de métodos utiliza a assinatura dos métodos para diferenciá-los.

- O Polimorfismo paramétrico e o polimorfismo de sobrecarga atuam sobre o mesmo sentido.
- A sobreposição de um método precisa conter uma notação denominada *override*, e essa é aplicada de maneiras diferentes de acordo com a linguagem de programação utilizada
- Na sobrecarga de métodos, os elementos que compõem sua assinatura devem estar na mesma ordem e quantidade esperadas pelos atributos enviados.



VERIFICAÇÃO DE LEITURA

TEMA 8

1. Ao declarar um método com o seguinte escopo `acao(String a, int b, boolean c)`, o método executado será:
 - a) `Public void acao(String x, int y, boolean z){}`.
 - b) `Public void acao(int a, String b, boolean c){}`.
 - c) `Public void acao(boolean a, int b, String c){}`.
 - d) `Public void acao(String x, int b, String c){}`.
 - e) `Public void acao(String x, int y, String z){}`.
2. Sobre a notação `@Override`, pode-se afirmar que:
 - a) É obrigatória quando ocorrer sobrecarga de métodos.
 - b) É opcional quando ocorrer reescrita de métodos.
 - c) Deve sempre ter o símbolo de arroba (@), exceto na linguagem Kotlin.
 - d) É obrigatória quando houver sobreposição de métodos.

- e) Geralmente, é escrita da mesma maneira nas linguagens Java e C#.
3. A diferença entre o polimorfismo de sobrecarga e o de sobreposição é que:
- a) A sobrecarga implementa funções em um método já existente em outra classe.
 - b) A sobreposição implementa métodos declarados na mesma classe em que são sobrepostos.
 - c) A sobreposição, diferente da sobrecarga, tem a notação `@Override` como opcional.
 - d) A sobreposição tem todos os métodos sobrecarregados em uma mesma classe pai.
 - e) A sobrecarga tem todos os métodos sobrecarregados em uma mesma classe.

Glossário

- **Override:** na tradução literal do Inglês para o Português, *override* significa sobrepor.

Referências Bibliográficas

FURGERI, Sérgio. **Java 8–ensino didático**: desenvolvimento e Implementação de aplicações. São Paulo: Érica, 2018.

MICHAELIS, Dicionário Brasileiro da Língua Portuguesa. Disponível em <<http://michaelis.uol.com.br/busca?id=ZNQp2>>. Acesso em: 10 nov. 2018.



Gabarito – Tema 8

Questão 1 - Alternativa A

Independente dos nomes das variáveis passadas como parâmetros, o que é levado em consideração é a quantidade e tipos de dados enviados (que devem ser iguais aos que esperam ser recebidos).

Questão 2 - Alternativa D

Sobreposição e reescrita de métodos são a mesma coisa e sempre que houver uma sobreposição, é necessário informar que a sobreposição ocorreu intencionalmente. Informa-se isso por meio da notação *@Override* (que pode ser escrita de maneiras distinta, de acordo com a linguagem).

Questão 3 - Alternativa E

A sobrecarga tem como característica diferenciadora, o fato de todos os métodos sobrecarregados pertencerem à mesma classe, enquanto a sobreposição trabalha com métodos de classes externas.



Bons estudos!

