

Programação Orientada a Objetos

Programação Orientada a Objetos

Roque Maitino Neto

© 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 Acadêmico de Graduação e de Educação Básica

Mário Ghio Júnior

Conselho Acadêmico

Ana Lucia Jankovic Barduchi

Camila Cardoso Rotella

Danielly Nunes Andrade Noé

Grasiele Aparecida Lourenço

Isabel Cristina Chagas Barbin

Lidiane Cristina Vivaldini Olo

Thatiane Cristina dos Santos de Carvalho Ribeiro

Revisão Técnica

Marcio Aparecido Artero

Walter Gima

Editorial

Camila Cardoso Rotella (Diretora)

Lidiane Cristina Vivaldini Olo (Gerente)

Elmir Carvalho da Silva (Coordenador)

Letícia Bento Pieroni (Coordenadora)

Renata Jéssica Galdino (Coordenadora)

Dados Internacionais de Catalogação na Publicação (CIP)

Maitino Neto, Roque

M232p Programação orientada a objetos / Roque Maitino
Neto. – Londrina : Editora e Distribuidora Educacional S.A.,
2018.
208 p.

ISBN 978-85-522-0766-5

1. Programação. I. Maitino Neto, Roque. II. Título.

CDD 600

Thamiris Mantovani CRB-8/9491

2018

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

Unidade 1 Fundamentos da orientação a objetos	7
Seção 1.1 - Histórico e introdução à orientação a objetos	9
Seção 1.2 - Conceitos básicos de orientação a objetos	22
Seção 1.3 - Construtores e sobrecarga	37
 Unidade 2 Estruturas de programação orientadas a objetos	 59
Seção 2.1 - Estruturas de decisão e controle em Java	61
Seção 2.2 - Estruturas de repetição em Java	76
Seção 2.3 - Reutilização de classes em Java	93
 Unidade 3 Exceções, classes abstratas e interfaces	 111
Seção 3.1 - Definição e tratamento de exceções	113
Seção 3.2 - Definição e uso de classes abstratas	126
Seção 3.3 - Definição e uso de interfaces	141
 Unidade 4 Aplicações orientadas a objetos	 155
Seção 4.1 - Arrays em Java	157
Seção 4.2 - Strings em Java	173
Seção 4.3 - Coleções e arquivos	188

Palavras do autor

Caro aluno, seja bem-vindo ao fascinante mundo da Programação: orientação a objetos.

É provável que você conheça pessoas — mesmo que apenas por nome — com capacidade e perspicácia suficientes para criar coisas que, de tal sorte interessantes e belas, são chamadas de obras de arte. A sutileza dos traços de um pintor, o ritmo das palavras de um poeta e a sensibilidade de um músico são expressões de habilidades que, colocadas à serviço de suas obras, tornam-nas únicas e dignas de serem conhecidas por muita gente.

Mas afinal, o que programar um computador tem a ver com isso? Em que uma solução computacional se relaciona com arte? Imagine então uma ferramenta que nos permita criar programas elegantes, eficientes e fáceis de serem compreendidos. Pois o paradigma da orientação a objetos é justamente essa ferramenta. Se bem usado, ele pode viabilizar a construção de soluções tão interessantes e únicas, que não estaríamos cometendo exagero algum se as chamássemos de obras de arte.

Assim como compor uma música ou criar um poema, o processo de aprendizagem da programação orientada a objetos requer empenho e prática, além da correta estruturação dos temas. Por isso, esse material é dividido em quatro unidades, cada qual com um objetivo a ser atingido, uma competência a ser desenvolvida e um produto a ser entregue.

A primeira unidade aborda os fundamentos da orientação a objetos, o que inclui a apresentação do seu histórico e de seus principais recursos. Em particular, o objetivo dessa unidade é habilitá-lo a conhecer as diferenças entre as linguagens orientadas a objetos e procedurais. A segunda unidade trata das estruturas de seleção e repetição da linguagem Java, além de abordar herança, polimorfismo e encapsulamento, três das principais características da orientação a objetos. O objetivo dessa unidade é proporcionar a você a habilidade para manipular corretamente tais estruturas e para construir aplicações básicas em Java. O livro então avança para a terceira unidade, que trata das exceções, classes abstratas e interfaces, conhecimentos essenciais para que seus programas sejam de fácil manutenção e com

bom grau de escalabilidade. O objetivo dessa unidade é justamente proporcionar as condições necessárias para que você desenvolva soluções que se utilizem corretamente desses recursos da orientação a objetos. Por fim, a quarta unidade tem como objetivo promover sua capacitação no uso de strings, arrays, coleções e arquivos. Embora não sejam recursos exclusivos da orientação a objetos, é por meio desse paradigma que todo o potencial dessas estruturas poderá ser desenvolvido.

Sua dedicação e seu potencial serão decisivos para que a competência em compreender e utilizar a orientação a objetos seja adquirida plenamente. A efetiva incorporação dos recursos da linguagem Java em seu cotidiano como programador passa pela execução das atividades propostas, pela leitura atenta dos textos propostos e, é claro, pela sua participação ativa nos encontros presenciais.

Bom estudo! E sucesso!

Fundamentos da orientação a objetos

Convite ao estudo

Iniciamos, aqui, a primeira unidade do nosso livro didático, desejando a você um ótimo aproveitamento do conteúdo que nela será desenvolvido.

Não há como negar: se você quer aprender um assunto novo, que inclua alguma complexidade e que seja composto por vários outros conceitos, é necessário, antes de tudo, que os fundamentos sejam muito bem abordados e fixados. É exatamente essa a proposta desta unidade: introduzir o paradigma de orientação a objetos, passear pelos motivos que levaram à sua criação e tratar das suas principais características, sempre usando estilo de comunicação apropriado para quem ainda não teve contato com orientação a objetos.

Embora a análise das diferenças entre linguagens orientadas a objetos e linguagens não orientadas a objetos seja o produto a ser criado nesta unidade, a orientação a objeto será tratada de forma autônoma e sem referências desnecessárias ao paradigma procedural. Estimamos que, dessa forma, a competência para conhecer e compreender os fundamentos de orientação a objetos seja atingida com maior assertividade e eficiência.

Já que cada linha desse material é escrita com o objetivo de trazer elementos da sua vida profissional para o ambiente acadêmico, o que segue é a descrição de uma situação que poderá muito bem ocorrer com você.

Apresentamos uma empresa de confecção e instalação de vidro, espelho, box para banheiro e vários outros artigos de vidro. A empresa é tradicional em seu ramo e conta com departamentos bem estruturados e com funções definidas.

Um desses departamentos, aliás, é responsável pela criação e manutenção dos programas de computador que dão suporte às operações da empresa. Ocupando o cargo de gerente de tecnologia da informação da empresa, sua atuação inclui a contratação de pessoal, definição de funções e a escolha da tecnologia utilizada na empresa.

Embora a operação dos programas de computador seja bem executada pelos funcionários, há muito a empresa não investe na atualização desses programas. Muitas funções que poderiam ser simplificadas e aprimoradas acabaram permanecendo inalteradas por deficiências da linguagem de programação em que foram construídas. Por isso, não há mais dúvida: é hora de mudar!

No decorrer desta unidade, você será desafiado a apresentar boas justificativas ao proprietário da empresa para que ele permita investimento na troca da linguagem atual para linguagem orientada a objetos. É natural que essas justificativas venham acompanhadas da correta apresentação da linguagem, de seus conceitos básicos e das vantagens em relação à tecnologia atualmente utilizada.

Qual a melhor forma de se abordar um superior quando se deseja propor mudanças? Elas se justificarão no decorrer do tempo?

Três desafios derivarão dessa situação. Para que você tenha plena condição de superá-los, as seções desta unidade abordarão aspectos introdutórios da orientação a objetos, apresentarão seus conceitos básicos e, por fim, apresentarão construtores e sobrecarga, conceitos importantes do paradigma.

Aí está colocada a proposta. Desafio aceito? Sigamos adiante.

Seção 1.1

Histórico e introdução à orientação a objetos

Diálogo aberto

Caro aluno, seja bem-vindo à primeira seção desta unidade. Curioso para conhecer orientação a objetos? Ansioso para saber se o paradigma procedural é semelhante a ela? Logo você conseguirá responder a essa pergunta com tranquilidade. Antes, porém, é necessário introduzir a situação-problema que servirá como base para nosso estudo nesta unidade.

É tempo de mudança no departamento de tecnologia da informação da empresa. Embora o programa atual desempenhe sua função, ele poderia ser mais bem escrito, mais facilmente mantido e — por que não? — mais atual. Para que sua proposta de mudança siga adiante, será necessário convencer o proprietário da empresa, quanto à efetividade do paradigma de orientação a objetos. A primeira providência nesse sentido é contar a ele um pouco das suas características, tratar dos motivos que acarretaram sua criação e abordar, desde já, suas vantagens em relação à tecnologia atualmente usada na empresa. A tarefa de explicar o paradigma ao proprietário deverá ser cumprida por meio de um relatório ou, melhor ainda, por meio de um vídeo gravado por você.

Caso você opte pelo relatório, ele deverá ter no máximo duas páginas. Caso prefira o vídeo (o que sugerimos com entusiasmo), conte com até 8 minutos de gravação. Qualquer que seja o formato, utilize seu espaço ou seu tempo para:

- Apresentar o paradigma de orientação a objetos e contar um pouco da sua história.
- Tratar das suas principais características.
- Introduzir suas vantagens em relação ao paradigma atual.

Lembre-se: você estará explicando conteúdo técnico a quem não domina o assunto. Cuidado com termos que podem ser incompreensíveis para leigos.

Com a leitura atenta do trecho “Não pode faltar” e com a resolução das atividades propostas, imaginamos que você estará a cumprir sua missão. Boa leitura!

Não pode faltar

Você já imaginou se pudéssemos representar coisas da vida real tal qual elas são em um programa de computador? Já imaginou se conseguíssemos reproduzir suas informações e seus comportamentos na resolução de um problema computacional? Pois, com a orientação a objetos, nós podemos. E, a partir de agora, começamos a obter respostas para essas questões.

Introdução ao paradigma de orientação a objetos

Uma das primeiras providências a serem tomadas quando se deseja aprender algo novo é entender os termos que compõem a expressão que a identifica. Em outras palavras, entenda o nome do novo assunto antes de se aprofundar nele. Embora o termo “paradigma” não seja tão comum assim em nosso cotidiano, seu significado é simples. De acordo com Houaiss, Franco e Villar (2001, p. 329), paradigma significa modelo, padrão. No contexto da programação de computadores, um paradigma é um jeito, uma maneira, um estilo de se programar.

O restante do termo — orientação a objetos — será esmiuçado nas próximas linhas e em todo este material. Então, não trataremos especificamente dele aqui. Ao contrário, vamos começar abordando algo que você já conhece.

O programa de computador é o meio pelo qual um computador executa cálculos e funções. Ele é escrito com o uso de comandos, operadores, variáveis e alguns outros recursos que, dispostos corretamente, formam uma linguagem de programação. O que de melhor um computador faz é receber, processar e disponibilizar nomes, valores, fórmulas e textos que, de forma genérica, chamamos de dados. Tudo isso, é claro, por meio de um programa.

O que nos interessa, por ora, são justamente os dados. Santos (2003) nos ensina que o paradigma de programação orientada a objetos considera que os dados a serem processados e os mecanismos de processamento desses dados devem ser considerados em conjunto.

Essa relação entre os dados e as operações neles aplicadas logo será mais bem detalhada.

Quando representamos elementos reais de forma simplificada e padronizada, estamos criando um modelo para esses elementos.

Modelos são representações simplificadas de objetos, pessoas, itens, tarefas, processos, conceitos, ideias etc., usados comumente por pessoas no seu dia a dia, independentemente do uso de computadores (SANTOS, 2003, p. 2).



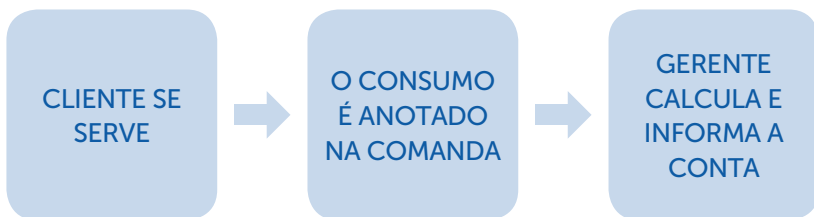
Refleta

Seria adequado, então, associarmos modelos a elementos que vemos no mundo real?

Os modelos geralmente agregam dados e as operações que se aplicam a eles. Para entendermos melhor esse conceito, devemos passar pelo exemplo que segue, adaptado de Santos (2003).

O restaurante POOeNaMesa, em seu procedimento diário, permite que o próprio cliente se sirva da refeição que deseja. O peso da comida e demais itens consumidos são então anotados no que chamaremos de comanda, cujo desenho pode ser visto na Figura 1.2. Quando o cliente pede a conta, o gerente calcula o valor devido por meio do que foi anotado na comanda e a apresenta ao cliente. O processo é representado na Figura 1.1.

Figura 1.1 | Processo do restaurante POOeNaMesa



Fonte: elaborado pelo autor.

Pronto, temos aqui um modelo do restaurante. De forma simplificada, ele é capaz de agregar as informações necessárias para a contabilização dos pedidos (peso da refeição, tipo e quantidade de refrigerantes solicitados e a sobremesa, por exemplo) e as operações ou os procedimentos associados a essas informações, como

encerramento do pedido, apresentação da conta para o cliente e inclusão de um novo item, entre outras.

Figura 1.2 | Comanda do restaurante POOeNaMesa

Mesa número _____

	Refeição (kg)
	Sobremesa
	Refrigerante (2l)
	Refrigerante (0,6l)
	Refrigerante (lata)
	Cerveja

Fonte: adaptado de Santos (2003).

A mesma forma de utilização de um modelo pode ser considerada para submodelos. Usando o POOeNaMesa como referência, imagine uma comanda específica para conter os dados da sobremesa ou das bebidas consumidas. Essa comanda específica conterá características gerais da comanda principal, como o nome do restaurante, mas tratará especificamente do armazenamento dos dados da sobremesa consumida, por exemplo.

Dependendo do contexto em que um modelo está inserido, é comum que ele assuma certas particularidades. Tomemos como exemplo a representação das informações de uma pessoa. Se estivermos tratando de um modelo em que uma pessoa é um eleitor, por exemplo, alguns dados serão relevantes e outros não. Observe os exemplos.

Pessoa considerada como eleitor: nesse caso, é necessário conhecermos seu nome, endereço, número de inscrição, zona de votação e seção. A operação de alteração de domicílio eleitoral poderia ser aplicada nesse modelo.

Pessoa considerada como aluno: para a composição do modelo de aluno, é necessário que se tenha o nome, o número de matrícula, a nota da primeira prova, a nota da segunda prova, as faltas e a nota final, entre outros dados. Operações como consultar nota e solicitar revisão de ausências são plausíveis nesse contexto.

Pessoa considerada como motorista: nesse caso, as informações de nome do condutor, número da CNH, histórico de multas e data da

revalidação devem compor o modelo, assim como as operações de consulta muitas e solicita revalidação.

Lembre-se: a inclusão ou não de dados ou operações no modelo depende fortemente do contexto. Não faria sentido colocarmos, nos respectivos modelos, o dado de salário do condutor do veículo ou a operação de consulta de nota para a representação de um eleitor. Embora ambos possam ser necessários em alguma circunstância de representação de uma pessoa, no contexto em que foram incluídos, não teriam aplicação prática alguma e contrariariam o princípio de simplificação ao qual todo modelo deve estar sujeito.

Antes de terminarmos nossa abordagem sobre modelos, vale a pena tratarmos da capacidade de serem reutilizados. Para representarmos, por exemplo, diferentes alunos — cada um com dados específicos e únicos, como o número de matrícula —, não precisamos, necessariamente, de um modelo para cada aluno. Ao contrário, cada um deles deve ser representado pelo mesmo modelo, respeitadas as especificidades de cada um. Essa característica fará todo o sentido em aulas futuras, pode apostar.

Histórico do paradigma de orientação a objetos

Para entender o que a orientação a objeto representa hoje, nada melhor do que vasculhar seus primórdios, mesmo que de modo breve. O criador da expressão programação orientada a objetos (POO) foi Alan Kay, o mesmo que criou a linguagem Smalltalk. No entanto, mesmo antes de o termo ter sido criado, ideias conceituais sobre orientação a objetos já estavam sendo aplicadas na linguagem de programação SIMULA 67 (DOUGLAS, 2015).

Como seu nome sugere, essa linguagem era usada para criar simulações. Alan Kay, que atuava na Universidade de Utah naquela época, gostou do que viu na SIMULA. Consta que ele teria vislumbrado um computador pessoal que pudesse fornecer aplicações orientadas a gráficos e intuiu que uma linguagem como a SIMULA poderia oferecer bons recursos para leigos criarem tais aplicações.

Kay então resolveu “vender sua visão” à Xerox e no início dos anos 1970; sua equipe criou o primeiro computador pessoal, o Dynabook. A linguagem Smalltalk, que era orientada a objetos e também orientada a gráficos, foi desenvolvida para programar o Dynabook. Ela existe até hoje, embora não seja largamente usada para fins comerciais.

A ideia da programação orientada a objetos ganhou impulso na década de 1970 e, no começo da década de 1980, Bjarne Stroustrup integrou a orientação a objeto na linguagem C, o que resultou no C++, tida como a primeira linguagem OO usada em massa (THE UNIVERSITY OF TENNESSEE, [s.d.]).

No início dos anos 1990, um grupo da Sun, liderado por James Gosling, desenvolveu uma versão mais simples do C++, que foi batizada de Java. O grupo esperava que o Java fosse usado para aplicações de vídeo sob demanda, mas o projeto não evoluiu. Foi quando Gosling resolveu voltar sua linguagem para aplicações de internet, e o resto da história você já conhece.



Assimile

Modelos são representações simplificadas de objetos, pessoas, itens, tarefas, processos, conceitos, ideias etc., usados comumente por pessoas no seu dia a dia, independentemente do uso de computadores (SANTOS, 2003, p. 2).

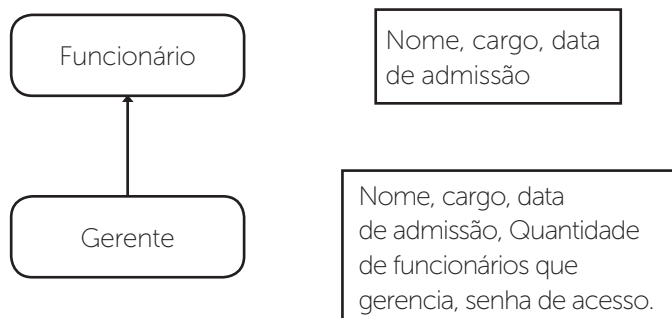
Características e princípios do paradigma de orientação a objetos

Machado (2015) entende que o paradigma da orientação a objeto é fundamentado por quatro características.

- **Abstração:** a abstração está relacionada à definição precisa de um objeto. Essa definição inclui sua identificação (nome), suas características (ou propriedades) e o conjunto de ações que ele desempenha. Tomemos como exemplo um cachorro: o objeto cachorro deve ser único e não poderá ser repetido. Nesse ponto, o objeto já tem identidade definida. Sua caracterização se dá pela cor do pelo, peso, raça e por aí vai. Por fim, as ações que ele é capaz de desempenhar incluem latir, farejar, pular etc. Pronto! Conseguimos abstrair o objeto cachorro e o temos perfeitamente definido.

- **Herança:** por meio dessa característica do paradigma OO, um objeto filho herdará características e comportamentos do objeto pai. Quando estiver criando classes, você vai perceber que essa possibilidade permite o reaproveitamento de código e torna o trabalho mais racional e otimizado. A ideia da herança é mostrada na Figura 1.3.

Figura 1.3 | Herança na orientação a objetos



Fonte: elaborado pelo autor.

Para que você possa entender a Figura 1.3, considere os dados colocados nos quadros como características de funcionário e gerente, respectivamente. Podemos ilustrar a herança com os dados do modelo "Funcionário". Eles devem identificá-lo, por exemplo, com nome, cargo e data de admissão, entre outros. No entanto, existe uma categoria especial de funcionário chamada "Gerente" que, além desses dados, deverá também conter a quantidade de funcionários que gerencia e a senha de acesso ao sistema da empresa.

O modelo "Gerente" herda características do modelo "Funcionário", especificamente os dados: nome, cargo e data de admissão. Essa é a ideia central da herança: aproveitar características e comportamentos gerais em modelos mais específicos. Essa particularidade nos permite entender que um objeto mais abaixo é um caso específico do objeto acima, ou seja, ele possui as características gerais do objeto pai, acrescidas de mais algumas especificidades que o diferem do seu ancestral.



Pesquise mais

Acesse <<https://www.youtube.com/watch?v=vhvmZfxZhPw>> (Acesso em: 5 out. 2017) para assistir ao vídeo de Danilo Filitto. Em pouco mais de 9 minutos, o autor sintetiza conceitos básicos do paradigma de orientação a objetos, com clareza e objetividade.

Polimorfismo: antes de entendermos esse conceito, melhor entendermos a composição de seu nome. O termo “poli” significa muitos, vários. Já “morfismo” remete a formas. Então, temos que polimorfismo significa “muitas formas”, e é exatamente essa definição que caracteriza com exatidão esse pilar da orientação a objetos.

Já sabemos que um objeto filho herda características e ações de seu objeto pai, situado hierarquicamente acima do primeiro. Contudo, em certos casos, precisaremos definir ações do objeto de outra forma. Assim, polimorfismo consiste em dar outra forma à alguma ação herdada do objeto pai.

Imagine um objeto chamado eletrodoméstico. Uma das suas ações consiste em ligar. No entanto, os objetos forno de micro-ondas e televisão — que são especializações de eletrodoméstico — são ligados de formas diferentes. Por isso, para cada um dos objetos filho, a ação ligar será descrita de modo diferente.

• **Encapsulamento:** nesse nosso contexto, o termo encapsulamento está relacionado à proteção ou ocultação dos dados do objeto. Para entender essa característica do paradigma OO, pense em uma câmera fotográfica automática. Santos (2003) ensina que, quando você clica o botão para tirar a foto, o processo de seleção de velocidade e de abertura apropriada do obturador é iniciado. Para quem está operando a câmera, os detalhes relacionados a velocidade, tipo de filme e ajuste à iluminação do ambiente não têm relevância alguma. O que importa, de fato, é que a foto seja tirada.

Ainda usando a analogia da máquina fotográfica, a ocultação do comportamento e dos dados relativos ao processo permite que o usuário se preocupe apenas em tirar a foto e o impede de modificar dados e comportamentos da câmera. Assim será quando você estiver escrevendo classes e utilizando objetos.



Pesquise mais

Observe este exemplo de modelo:

Lâmpada LED (*Light Emitting Diode*)

Dado básico: estado de ligada ou desligada.

Operações: ligar, desligar, alterar cor (Em alguns modelos, isso é possível pelo acionamento e desligamento da lâmpada via interruptor em menos de um segundo), mostrar estado (Operação desnecessária na vida real,

mas necessária durante a modelagem).

A mudança de estado se dá pelo ato de ligar ou desligar a lâmpada. A Figura 1.4 mostra o modelo “Lâmpada”.

Figura 1.4 | O modelo “Lâmpada”, com seus dados e operações

Lâmpada	
-	Estado da lâmpada
-	Acende
-	Apaga
-	Altera cor
-	Mostra estado

Fonte: adaptado de Santos (2003, p. 7).

Essa representação coloca o nome do modelo (sem acento) na primeira divisão do retângulo. Logo em seguida é descrito o único dado (haverá mais que um, na maioria dos casos) do modelo. Na terceira divisão da representação, são exibidos os comportamentos ou as funções do modelo Lâmpada.

Bem, essa é apenas uma porção do que pode ser abordado em relação aos fundamentos da orientação a objetos. Quanto melhor esses fundamentos forem assimilados, mais bem-sucedida será a evolução da sua habilidade nesse paradigma. Não deixe de resolver os exercícios propostos e de fazer a leitura atenta dos textos sugeridos.

Bom estudo!

Sem medo de errar

Na descrição da situação-problema, você foi desafiado a contar ao proprietário da empresa um pouco das características do paradigma de orientação a objetos, tratar dos motivos que acarretaram sua criação e convencer-lhe das suas vantagens em relação à tecnologia atualmente usada na empresa. É com esse último item, aliás, que você começará a produzir o que se espera para essa unidade, que é a análise das diferenças entre linguagens orientadas a objetos e linguagens procedurais não orientadas a objetos.

Antes de prosseguirmos, duas observações devem ser feitas:

a) Não há apenas uma forma de se tratar o problema, já que não estamos diante de um mero exercício. Ao explicar um tema a alguém, você lança mão de seus conhecimentos previamente adquiridos e faz uso de recursos pessoais e apropriados para essas situações, o que torna única cada “resolução”.

b) A forma ideal para essa atividade é a gravação do vídeo, por ser um meio mais pessoal e menos propenso a mal-entendido. Por meio do vídeo, o alvo da sua explicação terá contato com sua expressão facial, sua entonação e seus gestos, algo impossível pelo meio escrito. E tudo isso, é claro, pode contar a seu favor.

Isso posto, passemos ao roteiro de uma solução possível.

1) **Apresentar o paradigma de orientação a objetos:** coloque o paradigma OO como uma forma mais apropriada para a criação de programas que se caracterizem pela flexibilidade, escalabilidade e facilidade de manutenção. Resuma essas qualidades contando que uma eventual mudança ou acréscimo que se deva fazer em um programa escrito sob esse padrão é de fácil implementação.

2) **Contar um pouco da história dele:** nessa parte da explicação, trate o paradigma OO como uma evolução do paradigma procedimental. A história da OO começa na década de 1960, o que nos faz supor que se trata de um padrão já bastante testado e devidamente sedimentado. Assim, haverá segurança de que ele não deixará de ser usado repentinamente pela comunidade de TI ou será trocado por outro padrão a qualquer momento.

3) **Tratar das suas principais características e introduzir suas vantagens em relação ao paradigma atual:** nesse tópico, pode ser destacada a facilidade com que a orientação a objetos cria e manipula modelos. Explique que os modelos servem para implementar coisas (no sentido amplo) do mundo real para um programa de computador, facilitando a simulação do comportamento dessa entidade. No mundo procedimental, essa facilidade não é tão expoente.

Avançando na prática

Um exemplo para esclarecer

Descrição da situação-problema

Vamos usar o “Avançando na prática” para apresentar uma circunstância que poderia derivar da situação-problema já apresentada e resolvida.

Durante sua explicação sobre as principais características da orientação a objetos, o proprietário ficou particularmente interessado na questão dos modelos. Para que ele entenda melhor do assunto, você deverá criar um exemplo de modelo que faça parte do dia a dia do proprietário. Apresente os dados e comportamentos da entidade da mesma forma que foi apresentado no quadro “Exemplificando”. Mãos à obra!

Resolução da situação-problema

A sugestão de entidade a ser desenvolvida é cliente. No âmbito de uma empresa que comercializa produtos e presta serviços relacionados a vidros, identifique os principais dados que devem caracterizar um cliente (nome, CPF ou CNPJ, endereço etc.) e seus principais comportamentos, como `realiza_pedido`, `confirma_serviço`, `realiza_pagamento`, entre outros.

Faça valer a pena

1. A programação orientada a objetos separa claramente a noção de o que é feito de como é feito. O que é descrito como um conjunto de métodos e suas semânticas associadas. O como de um objeto é definido pela sua classe, que define a implementação dos métodos que o objeto suporta (ARNOLD, GOSLING, HOLMES, 2007, p. 62).

Assinale a alternativa que contém a expressão que melhor define “modelo” em nosso contexto.

- a) Abreviação.
- b) Demonstração.
- c) Simplificação.
- d) Apresentação.
- e) Representação.

2. Programas processam dados: valores em uma conta bancária, caracteres entrados por um teclado, pontos em uma imagem, valores numéricos para cálculos. O paradigma de programação orientada a objetos considera que os dados a serem processados e os mecanismos de processamento desses dados devem ser considerados em conjunto (SANTOS, 2003, p. 1).

Entendidos como base para a criação de objetos no paradigma OO, os modelos apresentam algumas características próprias. Assinale a alternativa que contém apenas indicações de afirmações verdadeiras relacionadas a tais características.

I) Modelos são capazes de agregar informações e operações sobre certos objetos da vida real.

II) Um modelo genérico pode gerar modelos mais específicos, dependendo do contexto em que é analisado.

III) Não é possível a criação de modelos que contenham apenas operações e que não contenham dados do modelo, já que isso descaracterizaria o modelo.

a) Apenas as afirmações I e II são verdadeiras.

b) Apenas as afirmações II e III são verdadeiras.

c) Apenas as afirmações I e III são verdadeiras.

d) Apenas a afirmação I é verdadeira.

e) Apenas a afirmação II é verdadeira.

3. Em programação orientada a objetos, os dados pertencentes aos modelos são representados por tipos de dados nativos, ou seja, que são característicos da linguagem de programação. Dados também podem ser representados por modelos já existentes na linguagem ou por outros modelos criados pelo programador (SANTOS, 2003, p. 4).

Analise as afirmações referentes às quatro características fundamentais do paradigma de orientação a objetos.

I) Polimorfismo significa “muitas formas” e relaciona-se com a capacidade de um objeto assumir uma forma que oculte e proteja seus dados do acesso de outros objetos.

II) A abstração está relacionada à definição de um objeto, incluindo sua identificação, suas características e ações.

III) A característica da herança nos permite entender que um objeto mais abaixo na hierarquia é um caso genérico do objeto mais acima.

IV) É por meio do encapsulamento que um objeto consegue transmitir suas características e seu conjunto de ações a outro objeto.

Assinale a alternativa que contém apenas indicações de afirmações verdadeiras.

- a) Apenas as afirmações II e III são verdadeiras.
- b) Apenas as afirmações I, II e IV são verdadeiras.
- c) Apenas a afirmação II é verdadeira.
- d) Apenas as afirmações I e II são verdadeiras.
- e) Apenas a afirmação I é verdadeira.

Seção 1.2

Conceitos básicos de orientação a objetos

Diálogo aberto

Na primeira seção, você ficou incumbido de apresentar aspectos básicos do paradigma de orientação a objetos. É bom lembrarmos que a necessidade dessa apresentação surge da urgência em se adotar um novo paradigma de programação, visando à agilidade na manutenção dos programas e ao aumento da escalabilidade.

Por causa da sua competência em mostrar-lhe as vantagens em investir na atualização da tecnologia dos programas da empresa, o proprietário se sentiu propenso a acatar sua proposta. No entanto, antes de tomar qualquer decisão, ele gostaria de contar com a opinião da sua equipe do departamento de TI. Por isso, novamente você será responsável por esclarecer e convencer pessoas a aderirem à sua ideia.

Sua missão é explicar aspectos básicos e mais técnicos da orientação a objeto, dessa vez para seu time técnico. Quatro itens no mínimo serão obrigatórios em sua explicação:

- **Classes:** estruturas da orientação a objetos que servem para conter os dados que devem ser representados e as operações que devem ser efetuadas com esses dados para determinado modelo.
- **Objetos:** é como chamamos a materialização da classe, que assim poderá ser usada para representar dados e executar operações.
- **Atributos:** são as propriedades (ou dados) da classe.
- **Métodos:** os métodos expressam as funcionalidades da classe.

Para que sua explicação não se perca em palavras ditas, é necessário que você a registre em um relatório ou, preferencialmente, em um vídeo de até 10 minutos. Um lembrete se faz necessário: embora sua missão seja novamente a de esclarecer e convencer, você estará tratando com pessoas de formação técnica e isso muda a abordagem.

Não pode faltar

Caro aluno, seja bem-vindo à segunda seção de Programação orientada a objetos. Faremos, nesta ocasião, a abordagem dos conceitos fundamentais de classes, objetos, métodos e atributos, quatro importantes pilares da programação orientada a objetos (POO) e que estão fortemente relacionados com a ideia de modelos desenvolvida em nosso último encontro.

A compreensão desses elementos básicos da POO será muito importante para a continuidade dos nossos estudos. Sem isso, não conseguiremos tirar todo o proveito da herança, do encapsulamento e do polimorfismo, além de tantas outras características que tornam a orientação a objetos tão poderosa.

Vamos adiante e boa leitura!

Classes e objetos para programação orientada a objetos

Não há como negar: as classes e os modelos – estes últimos estudados em nossa primeira aula – relacionam-se diretamente. Se entendemos que modelos são representações simplificadas de um objeto ou de um processo, entre outras coisas, então devemos também entender que as classes são implementações desses modelos em uma linguagem orientada a objetos.

Classes são estruturas das linguagens de POO criadas para conter os dados que devem ser representados e as operações que devem ser efetuadas com esses dados para determinado modelo (SANTOS, 2003, p. 14). Repare como essa definição associa classe e modelo, passando pela ideia de representação de dados e operações em ambos.

Quando esboçamos nossos primeiros modelos, não foi mencionada a existência de padrão de representação naqueles objetos, embora ele estivesse presente. A escrita de uma classe, contudo, requer critério.



Pesquise mais

Os critérios de escrita de uma classe são bem explicados pela própria Oracle, empresa que mantém o Java. Visite <<http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html>>

(acesso em: 3 out. 2017) e saiba mais. Uma boa fonte em português pode ser encontrada em <<http://www.devmedia.com.br/convencoes-de-codigo-java/23871>> (acesso em: 3 out. 2017).

O que vem a seguir serve como introdução às regras de escrita de classes na linguagem Java. Vejamos:

- *Nome da classe*: início com letra maiúscula, sem acento e sem espaço entre palavras.
- *Nome de métodos e atributos*: início com letra minúscula; acento permitido, porém pouco recomendado; sem espaço entre palavras.

O procedimento de criação de uma classe segue as regras determinadas pela linguagem Java, incluindo:

- Uma classe é sempre declarada com a palavra reservada *class*, seguida do nome da classe.
- O corpo da classe é delimitado por { e } (leia-se “abre chave” e “fecha chave”). A cada “abre chave” deve corresponder um “fecha chave”.

Observe este exemplo:

// A classe Vazia não possui campos nem métodos, mas mesmo assim

// pode ser usada para exemplificar as regras sintáticas básicas

// Java.

```
public class Vazia {
```

```
    /* Se houvesse campos ou métodos para a classe Vazia, eles  
    deveriam ser declarados aqui dentro. */
```

```
}
```

- A palavra reservada *class* deve ser escrita em letra minúscula.
- A palavra reservada *public* é um modificador de acesso. Ela garante que o campo ou método da classe poderá ser acessado e/ou

executado a partir de qualquer outra classe. Mais detalhes a respeito de modificadores de acesso serão dados em aulas futuras.

- Embora o nome da classe possa ter acento, você poderá ter problemas com nomes de arquivos com acento, já que o compilador gera o arquivo com nome idêntico à classe.
- A sequência `//` indica que tudo o que segue na linha não será considerado pela linguagem Java, mas servirá apenas como comentário do programador.

Instâncias e atributos para programação orientada a objetos

Embora essa abordagem de classes seja apenas introdutória, ainda assim nos cabe fazer uma observação: já que podemos comparar uma classe a um modelo, então é razoável imaginarmos que, para se tornar utilizável, esse modelo precisa “ganhar vida” por meio da criação de um exemplo ou instância de si mesmo.

Um objeto (ou instância) é como chamamos a materialização da classe, que assim poderá ser usada para representar dados e executar operações. Fazendo uma analogia, uma classe poderia ser considerada a planta de um prédio, que descreve o prédio, mas não corresponde fisicamente a ele. Os prédios construídos de acordo com aquela planta seriam as instâncias (SANTOS, 2003).

Antes de focarmos objetos, devemos tratar dos atributos. Nesse ponto, vale a pena lembrarmos do conceito de classe, com destaque para o termo “dados”:



Assimile

Classes são estruturas das linguagens de programação orientada a objetos criadas para conter os **dados** que devem ser representados e as operações que devem ser efetuadas com esses dados para determinado modelo (SANTOS, 2003, p. 14).

Chamamos de atributos os **dados** associados a uma determinada classe. Será comum encontrar esse conceito ligado à palavra variável ou a campo. Arnold, Gosling e Holmes (2007, p. 38) definem campo como uma das espécies de membros da classe, como segue: campos são variáveis de dados associados com uma classe e seus objetos.

Campos armazenam resultados de computações realizadas pela classe.

A linguagem de programação Java é estaticamente tipada (*statically-typed*), o que significa que todas as variáveis devem primeiro ser declaradas para que depois possam ser usadas. Basta declarar o tipo de dado, seguido do nome do campo.

Exemplo:

```
int gear = 1; // Essa declaração informa ao compilador que seu
programa criou uma variável chamada "gear", que deverá armazenar
um dado numérico e que tem o valor inicial 1.
```



Refleta

Mesmo com restrições impostas pela linguagem de programação, o desenvolvedor tem liberdade para escolher os nomes de suas variáveis, seus métodos e suas classes. Genericamente, esses nomes são conhecidos como identificadores. No entanto, o bom senso deve sempre prevalecer nas escolhas. Não seria razoável criar os nomes desses elementos conforme a função que eles executam no contexto do programa? Qual o sentido, por exemplo, de chamar de "Multiplicação" uma classe que executa operações de adição?

Observe esse outro exemplo, adaptado de Santos (2003).

```
public class RegistroAcademicoSimples {
    String nomeDoAluno; //cadeia de caracteres que deverá
    conter o nome do aluno.
    int númeroDeMatrícula; // representará o RA do aluno.
    boolean éBolsista; //assume true ou false.
    short anoDeMatrícula; //representará o ano da matrícula.
}
```

Além do nome do campo (`nomeDoAluno`, `númeroDeMatrícula`, `éBolsista` e `anoDeMatrícula`), outra informação relevante associada a ele é seu tipo. De acordo com Oracle (2015), os oito tipos de dados primitivos suportados pelo Java são *byte*, *short*, *int*, *long*, *float*, *double*, *boolean* e *char*. Por ora, ficamos apenas na menção de seus nomes. Em encontros futuros, trataremos adequadamente dos tipos de dados.

Voltemos aos objetos. Objetos são criados por expressões contendo a palavra *new*. Criar um objeto a partir de uma definição de classe é também conhecido como instanciação. Por isso, é comum que os objetos sejam também chamados de instâncias (ARNOLD; GOSLING; HOLMES, 2007, p. 38).

O formato básico da inicialização de referências é *NomeDaClasse nome daReferência = new NomeDaClasse*.

Observe o exemplo que segue, adaptado de Santos (2003). Para simplificar as coisas, o código foi fragmentado.

```
public class DemoDataSimples {  
    /**  
        * O método main permite a execução desta classe. Este método  
        contém declarações de algumas instâncias da classe  
        * DataSimples e demonstra como seus campos podem ser  
        acessados diretamente, já que são públicos.  
  
        public static void main (String[] args)  
        {  
            // Criamos duas instâncias da classe DataSimples,  
            usando new.  
            // As instâncias serão associadas a duas referências,  
            que permitirão  
            // o acesso aos campos e métodos das instâncias.  
  
            DataSimples hoje = new DataSimples();  
            DataSimples    independênciaDoBrasil    =    new  
            DataSimples();  
            byte umDia, umMês; short umAno; // e três variáveis  
            para receberem o dia, mês e ano para as datas.  
            umDia = 40; umMês = 1; umAno = 2017;  
            hoje.inicializaDataSimples(umDia, umMês, umAno); //  
            inicializa os campos da instância.  
            hoje.mostraDataSimples(); //imprime 0/0/0.
```

```

        // Inicializando "independênciaDoBrasil" como uma
data válida

        umDia = 7; umMês = 9; umAno = 1822;
        independênciaDoBrasil.inicializaDataSimples(umDia,
umMês, umAno);
        independênciaDoBrasil.mostraDataSimples();        //
imprime 7/9/1822.

...

        hoje.mostraDataSimples(); //imprime 0/3/2017
        independênciaDoBrasil.mês = 13;
        independênciaDoBrasil.mostraDataSimples();        //
imprime 7/13/1822.

    } //fim do método main()
} // fim da classe DemoDataSimples.

```

Informações importantes sobre essa classe:

- Duas instâncias da classe `DataSimples` são declaradas. (`DataSimples hoje = new DataSimples();` `DataSimples independênciaDoBrasil = new DataSimples();`)
- Para receber dia, mês e ano, três variáveis de tipos compatíveis com os definidos na classe `DataSimples` são declaradas. Elas não devem ser inicializadas com a palavra-chave `new`.
- Métodos que foram definidos dentro das classes podem ser executados usando a notação `nomeDaReferência.nomeDoMétodo(argumentos)`, mas somente se a instância tiver sido criada com `new`.

Nas seções que virão, trataremos em detalhes dos objetos.

Criação e utilização de métodos

Neste ponto, já sabemos criar classes e representar campos nas

classes com tipos que estudamos previamente. Nossos esforços agora serão concentrados em uma maneira de efetuar operações nas classes.

As operações nas classes são feitas por meio de métodos. O nome do método deve iniciar por letra (e \$ também vale); deve ser composto por uma única palavra; pode conter números, desde que a partir do segundo caractere; e pode ter acento.

Métodos não podem ser criados dentro de outros métodos nem fora da classe à qual pertencem. Não podemos ter métodos isolados, fora de alguma classe (SANTOS, 2003).

Um método sempre tem de definir o que retorna, nem que defina que não há retorno. Nesse último caso, a palavra reservada `void` deve ser usada.

Um método pode retornar um valor para o código que o chamou. No caso do `dataÉVálida`, o método deve devolver um valor booleano indicando se a data é válida ou não. A palavra-chave `return` indica que o método vai terminar ali, retornando tal informação.

Outro elemento importante no contexto é o escopo. Ele determina a visibilidade dos elementos da classe. Em outras palavras, o escopo determina se variáveis ou campos podem ser acessados e/ou modificados em:

- Todos os métodos da classe.
- Somente em um determinado método.
- Ou somente em parte de um determinado método.

O exemplo que segue é oferecido para que classe, atributos e método possam ser visualizados em uma só estrutura. Considere o exemplo também apropriado para demonstrar o conceito de escopo, que será mais bem detalhado em encontros futuros.



Exemplificando

Adaptado de Santos (2003).

```
public class Triangulo {  
    float lado1;  
  
    boolean éEquilátero(){
```

```

        boolean igualdade12, resultado;
        igualdade12 = (lado1 == lado2);
        boolean igualdade23;
        igualdade23 = (lado2 == lado3);
        if (igualdade12 && igualdade23)
            resultado = true;
        else
            resultado = false;
        return resultado;
    }

    float calculaPerimetro() {
        float resultado = lado1 + lado2 + lado3;
        return resultado;
    }
    float lado2, lado3;
}

```

A classe chamada Triangulo contém o atributo lado1. Ela contém ainda os métodos éEquilátero e calculaPerimetro, cada qual com suas próprias operações.

Algumas regras relacionadas a escopo merecem destaque:

I) Campos declarados em uma classe são válidos por toda a classe, mesmo que os campos estejam declarados depois dos métodos que os usam (ver campos lado2 e lado3).

II) Variáveis e instâncias declaradas dentro dos métodos só são válidas dentro desses métodos.

III) Dentro de métodos e blocos de comandos, a ordem de declaração de variáveis e referências a instâncias é considerada. Se as linhas

```

        boolean igualdade23;
        igualdade23 = (lado2 == lado3);

```


fossem trocadas, o compilador acusaria erro, já que uma variável não pode receber valor antes que seja declarada.

Criação de aplicações em Java

As classes não formam um programa completo e executável em Java. Para que um código Java seja capaz de executar efetivamente um algoritmo, ele deverá conter um método especial em uma classe, que será considerado o ponto de entrada do programa. Esse método especial leva o nome de *main*. O método *main* deve conter, obrigatoriamente, os modificadores *public static*, nesta ordem, e deve retornar *void* e receber como argumento um *array* de instâncias *String*.

Um programa em Java inicia-se pelo método *main()*, e sua declaração é *public static void main (String [] args)*.

public: torna o método visível de qualquer outra classe.

static: ainda será mais bem estudado. Dispensa a criação de uma instância da sua classe para que possamos criá-lo.

void: indica que *main()* não retorna nada.

String [] args: argumentos passados para classe executável via linha de comando.

O exemplo que segue ilustra a utilização do método *main*. Não se preocupe se não entender linha por linha ainda. Saiba, no entanto, que a aplicação calcula o fatorial de um número fornecido pelo usuário.

```
public class Fatorial {  
    public static void main (String args[])  
    {  
        Scanner entrada = new Scanner(System.in); //criação  
de um objeto do tipo Scanner, passando como argumento o objeto  
System.in  
  
        int fator, num;  
        System.out.println("Entre com um valor: ");  
        num = entrada.nextInt();  
        if (num>=0) {  
            if (num<=1) { System.out.print("\nResposta:
```

```

1");
    }
    else
    {
        fator = num;
        while (num>=2)
        {
            num = num-1;
            fator = fator * num;
            // ou simplesmente fator *=
--num;

        }
        System.out.printf("\nResposta:  %d",
fator);
    }
} else System.out.print("\nNumero inválido!");
}

```



Pesquise mais

Acesse <<http://www.devmedia.com.br/entendendo-a-estrutura-de-um-codigo-java/24622#ixzz3GmnOTDLM>> (acesso em: 15 set. 2017) e entenda melhor a estrutura de um código Java. A familiarização precoce com a forma de se escrever um programa é essencial para a fixação da estruturação do código.

Santos (2003) nos ensina que o método main poderia pertencer a qualquer classe, até mesmo a uma classe que representa um modelo qualquer. Em vez de usar esse estilo, tentaremos fazer que main seja o único método de uma classe, separado da representação dos modelos.

Arquivo fonte

Em Java, cada classe é colocada em um arquivo fonte. Esses arquivos representam partes de uma aplicação ou toda a aplicação (no

caso de programas muito pequenos). Arquivos fonte são gerados com a extensão .java e devem possuir o mesmo nome da classe que representam.

Aí estão, portanto, as linhas iniciais de classes, objetos, métodos e atributos. É certo que esses temas serão revisitados muitas e muitas vezes durante todo o curso, mas o entendimento inicial de seus papéis no contexto é fundamental para a sequência das aulas. A literatura nos oferece farta abordagem desses itens introdutórios, e os livros (e os bons sites também, por que não?) devem se tornar referência constante em seus estudos.

Sigamos com a resolução da situação-problema e com os primeiros exercícios da seção.

Sem medo de errar

A estrutura da resolução dessa situação-problema é muito parecida com a da resolução anterior. Conforme relato da situação, novamente você será responsável por esclarecer conceitos e convencer pessoas a aderirem sua ideia. Sua missão é explicar aspectos básicos e mais técnicos da orientação a objeto, dessa vez para seu time técnico. Quatro itens no mínimo serão obrigatórios em sua explicação: **classes, objetos, atributos e métodos**.

Antes de prosseguirmos, as duas observações feitas na Seção 1 devem ser lembradas:

a) Não há uma só forma de se tratar o problema, já que não estamos diante de um mero exercício. Ao explicar um tema a alguém, você lança mão de seus conhecimentos previamente adquiridos e usa de recursos pessoais e apropriados para essas situações, o que torna única cada “resolução”.

b) A forma ideal para essa atividade é a gravação do vídeo, por ser um meio mais pessoal e menos propenso a mal-entendido. Por meio do vídeo, o alvo da sua explicação terá contato com sua expressão facial, sua entonação e seus gestos, algo impossível pelo meio escrito. E tudo isso, é claro, pode contar a seu favor.

Feito o esclarecimento, passamos ao roteiro de uma solução possível.

1. Apresente classes como uma forma de modelar objetos do mundo real. Ressalte a possibilidade de se controlar dados e simular comportamentos desses objetos com facilidade em uma classe, algo não tão imediato no mundo procedural.

2. Apresente objetos como a materialização da classe. Faça a equipe entender que, para ganhar vida, uma classe deve ser instanciada. Dessa forma, valores individuais e operações próprias podem ser atribuídas ao objeto da classe.

3. Apresente os atributos como os dados da classe, sobre os quais incidirão operações.

4. Por fim, apresente os métodos como os elementos da classe em que são efetivadas as funções do programa. Com o devido cuidado, associe métodos da orientação a objetos aos procedimentos das linguagens procedurais.

Avançando na prática

Mais um exemplo para esclarecer

Descrição da situação-problema

Novamente usaremos o “Avançando na prática” para apresentar uma circunstância que poderia derivar da situação-problema já apresentada e resolvida.

Durante sua explicação sobre classes, atributos, objetos e métodos, a equipe ficou particularmente interessada nas **classes**. Para que o pessoal técnico compreenda melhor seu funcionamento, você deverá criar o corpo da classe **cliente**, conforme as regras de escrita da linguagem Java e conforme os dados de atributos e métodos fornecidos para o modelo cliente da aula passada.

Resolução da situação-problema

Novamente, a resolução não será explicitada em todas as suas linhas. Usando os exemplos desenvolvidos durante a abordagem teórica do conteúdo, estruture a classe, defina os campos e crie apenas as assinaturas (cabeçalhos) dos métodos. Não é necessário implementar os métodos efetivamente. Assim como na situação exposta na aula anterior, considere nome, CPF ou CNPJ, endereço etc.

como dados e realiza_pedido, confirma_serviço e realiza_pagamento como métodos.

Faça valer a pena

1. A programação orientada a objetos separa claramente a noção de o que é feito de como é feito. O “o que” é descrito como um conjunto de métodos — e, às vezes, com dados publicamente disponíveis — e suas semânticas associadas. Essa combinação de métodos, dados e semântica é muitas vezes descrita como um contrato entre o projetista da classe e o programador que a usa (ARNOLD; GOSLING; HOLMES, 2007).

Um método é um componente da classe que:

- a) Representa a estrutura da classe e dos seus atributos.
- b) Armazena o estado de uma classe e de seus atributos.
- c) Representa as variáveis de dados associadas à classe.
- d) Contém as ações (ou funcionalidades) de uma classe.
- e) Define os valores iniciais das variáveis da classe.

2. Uma das boas referências em orientação a objetos ensina que classes são estruturas das linguagens de programação orientada a objetos criadas para conter os dados que devem ser representados e as operações que devem ser efetuadas com esses dados para determinado modelo (SANTOS, 2003, p. 14).

Analise as afirmações que seguem e assinale a alternativa que contém apenas indicações de afirmações verdadeiras relacionadas a classes.

- I) Embora não possua meio de representar dados e comportamentos juntos, uma classe ainda assim pode ser comparada a um modelo.
- II) O rigor formal com que se escreve uma classe em Java restringe-se à correta colocação de letras maiúsculas e minúsculas em seu nome.
- III) Classes são criadas por expressões contendo a palavra new. Criar uma classe a partir de uma definição de atributos é conhecido como classificação.
- IV) A unidade fundamental de programação da linguagem de programação Java é a classe. Classes fornecem a estrutura para os objetos.

- a) Apenas a afirmação I é verdadeira.
- b) Apenas a afirmação IV é verdadeira.
- c) Apenas as afirmações III e IV são verdadeiras.
- d) Apenas as afirmações II e III são verdadeiras.
- e) Apenas as afirmações II e IV são verdadeiras.

3. Variáveis locais são declaradas dentro de um bloco de código, tal como o corpo de um método, em contraste com campos, que são declarados como membros de uma classe. Cada variável deve ter um tipo que precede seu nome quando a variável é declarada (ARNOLD; GOSLING; HOLMES, 2007).

Considerando o contexto apresentado, avalie as seguintes asserções e a relação proposta entre elas.

I) Campos declarados em uma classe são válidos por toda extensão da classe, desde que declarados antes do método que utilizam esses campos.

PORQUE

II) A visibilidade dos elementos da classe varia em função do tipo de classe declarada.

A respeito dessas asserções, assinale a alternativa verdadeira.

- a) As asserções I e II são proposições falsas.
- b) A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.
- c) A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.
- d) As asserções I e II são proposições verdadeiras, mas a II não é uma justificativa da I.
- e) As asserções I e II são proposições verdadeiras, e a II é uma justificativa da I.

Seção 1.3

Construtores e sobrecarga

Diálogo aberto

Caro aluno, aceite nossas boas-vindas para mais uma seção do nosso material.

Ao abraçar a causa da mudança do paradigma de programação das soluções de software da empresa, você buscou aprovação do dono da empresa e o apoio do pessoal que você gerencia no departamento de TI. Embora você tenha se saído muito bem nas duas empreitadas, ainda é necessário que lhes seja oferecido o apelo final, algo que tornará irresistível a adoção da mudança que você propôs. Utilizando-se dos mesmos meios disponibilizados nas seções anteriores (sempre com preferência para o vídeo), elabore uma apresentação de como a utilização de construtores e sobrecarga trará inegáveis vantagens ao desenvolvimento e à manutenção dos programas, salientando a exclusividade desse recurso no paradigma OO.

Com a conclusão desse desafio, você terá construído uma eficiente análise das diferenças entre o paradigma procedural (atualmente em uso) e a orientação a objetos.

Desafio aceito? Então mãos à obra!

Não pode faltar

Se você já conhecia o paradigma procedimental antes de iniciar este curso, então certamente já notou semelhanças e diferenças entre as duas formas de criar programas. Já foi possível perceber, certamente, que a definição de dados e ações de um certo objeto da vida real em uma classe facilita bastante a simulação do comportamento desse objeto no programa. Por outro lado, é notável a semelhança conceitual entre procedimentos e métodos, por pouco que tenhamos estudado o segundo.

Longe de termos esgotado o tratamento das generalidades dos métodos, trataremos, nesta seção, de alguns tipos especiais deles, sempre com foco na superação do desafio que nos foi colocado pela situação-problema. Sigamos adiante.

Definição e utilização de métodos construtores

Já sabemos que a criação de instâncias requer a utilização da palavra-chave `new`. Além de criar a instância, ela a associará a uma referência para que os métodos da classe instanciada possam ser executados.

Exemplo: `Data novaData = new Data();`

Até o momento, deixamos a cargo do programador (geralmente o programador-usuário) a criação de métodos que inicializam os campos da instância criada. Pelo fato de o compilador não obrigar o uso de métodos de inicialização, o programador pode se esquecer de fazê-lo.

Observemos as classes `RegistroAcademicoSemConstrutor` e `De moRegistroAcademicoSemConstrutor`, adaptadas de Santos (2003).

//Esta classe contém campos que representam dados simples de um registro acadêmico.

```
public class RegistroAcademicoSemConstrutor {  
    //declarando os campos da classe  
    private String nomeDoAluno; //nome do aluno  
    private int númeroDeMatrícula; //número de matrícula  
    private byte códigoDoCurso; //código do curso (1 .. 4)  
    private double percentualDeCobrança; //percentual em  
    relação ao preço cheio, de 0 a 100%  
    /**  
     * O método inicializaRegistroAcademicoSemConstrutor  
     recebe argumentos para inicializar  
     * os campos da classe RegistroAcademicoSemConstrutor  
     * @param n o nome do aluno  
     * @param m o número de matrícula
```



```

    * @param c o código do curso
    * @param p o percentual da bolsa
    */
    public void inicializaRegistroAcademicoSemConstrutor
(String n, int m, byte c, double p){
        nomeDoAluno = n;
        númeroDeMatrícula = m;
        códigoDoCurso = c;
        percentualDeCobrança = p;
    }

    /**
     * O método calculaMensalidade calcula e retorna a
    mensalidade do aluno usando
     * o código do seu curso e o percentual de cobrança.
     * @return o valor da mensalidade do aluno.
     */
    public double calculaMensalidade() {
        double mensalidade = 0; //o valor deve ser
    inicializado

        //determinação do valor cheio da mensalidade,
    dependendo do curso.

        if (códigoDoCurso == 1) mensalidade = 450.00; //
    Arquitetura

        if (códigoDoCurso == 2) mensalidade = 500.00; //
    Ciência da Computação

        if (códigoDoCurso == 3) mensalidade = 550.00; //
    Engenharia da Computação

        if (códigoDoCurso == 4) mensalidade = 380.00; //
    Zootecnia

```

//calcula o desconto. Se o percentual for zero, a mensalidade também o será.

 if (percentualDeCobrança == 1) mensalidade = 450.00;

 else mensalidade = mensalidade * 100.0 / percentualDeCobrança;

 return mensalidade;

 }

}

public class DemoRegistroAcademicoSemConstrutor {

public static void main (String[] argumentos) {

 RegistroAcademicoSemConstrutor renato = *new* RegistroAcademicoSemConstrutor();

 RegistroAcademicoSemConstrutor roberto = *new* RegistroAcademicoSemConstrutor();

 //Apenas uma das instâncias vai ser inicializada.

 renato.inicializaRegistroAcademicoSemConstrutor("Renato Ribeiro", 34980030, (byte)2, 75.00);

 //O cálculo é feito para os dois alunos, embora uma das instâncias não tenha sido inicializada.

 System.out.println("A mensalidade do Renato é "+renato.calculaMensalidade());

 System.out.println("A mensalidade do Roberto é "+roberto.calculaMensalidade());

 }

}

Saída:

```
Problems @ Javadoc Declaration Console
<terminated> DemoRegistroAcademicoSemConstrutor [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (20/03/2017 10:54:57)
A mensalidade do Renato é 543.4782608695652
A mensalidade do Roberto é NaN
```

Tratam-se de classes que fazem, basicamente, o cálculo de uma mensalidade, considerando o curso e o percentual de desconto concedido.

A mensalidade do primeiro aluno foi calculada corretamente. No cálculo da mensalidade do segundo aluno, os dados não foram informados, e a divisão por zero foi representada pelo Java como NaN.

Por isso, é comum que o programador-usuário seja forçado a passar dados para as instâncias criadas para que elas tenham sentido, e isso pode ser feito por meio de construtores.

Construtores são métodos especiais que são chamados automaticamente quando instâncias são criadas por meio da palavra-chave `new`. Eles garantem que o código contido neles será executado antes de qualquer outro código em outros métodos, já que uma instância de uma classe só pode ser usada depois de ter sido criada com `new`, o que causa a execução automática do construtor.

Fatos importantes:

- **Construtores** devem ter exatamente o mesmo nome da classe a que pertencem, inclusive considerando maiúsculas e minúsculas.
- **Construtores** não podem retornar valor algum, nem mesmo `void`. Por isso, devem ser declarados sem tipo de retorno.
- **Construtores** não devem receber modificadores (exemplo: `public` ou `private`). Eles serão públicos se a classe for pública.

Observe a classe `EventoAcademico`, que representa um evento como congresso, simpósio etc., e a classe `DemoEventoAcademico`, que a executa.

```
/**
```

* A classe EventoAcademico representa um evento acadêmico, como um

* congresso ou encontro, que se realiza em determinado período entre datas, local

* e com certo número de participantes.

*

* @author RoqueMaitino

*

*/

```
public class EventoAcademico {
```

```
    private String nomeDoEvento, localDoEvento;
```

```
    private Data inicioDoEvento, fimDoEvento;
```

```
    private int númeroDeParticipantes;
```

```
    /**
```

* O construtor para a classe EventoAcademico, que recebe argumentos para inicializar

* os campos da classe.

* @param i o início do evento.

* @param f o fim do evento.

* @param n o nome do evento.

* @param l o local do evento

* @param num o número de participantes do evento.

* */

```
EventoAcademico (String n, String l, Data i, Data f, int num) {
```

```
    nomeDoEvento = n;
```

```
    localDoEvento = l;
```

```
    inicioDoEvento = new Data();
```

```

        inicioDoEvento.inicializaData(i.retornaDia(),
i.retornaMês(), i.retornaAno());

        fimDoEvento = new Data();

        fimDoEvento.inicializaData(i.retornaDia(),
i.retornaMês(), i.retornaAno());

        númeroDeParticipantes = num;

    } //fim do construtor


    public String toString() {

        // aqui é feita a impressão do relatório. Código a ser
desenvolvido.

    }

}

```

O que vale a pena saber sobre a classe `EventoAcademico`:

- Essa classe não poderá ser executada como se encontra, já que lhe falta o método principal. Ela serve apenas como base para abordagem do conteúdo sobre construtor.
- O construtor da classe é bem semelhante a um método que seria usado para inicializar dados encapsulados na classe.
- Quando houver instâncias de outras classes sendo usadas como campos de uma classe, essas instâncias também podem ser inicializadas no construtor (vide *private Data inicioDoEvento, fimDoEvento*);
- Na classe `EventoAcademico`, existem duas referências a instâncias da classe `Data` que **devem** ser inicializadas explicitamente com *new*. O método `inicializaData` da classe `Data` é chamado, tendo como argumentos os valores obtidos da instância passada como argumento para o construtor. Assim, teremos uma cópia exata dos dados da data passada como argumento e não uma referência apontando para a mesma instância (SANTOS, 2003).

/**

* A classe DemoEventoAcademico demonstra o uso de instâncias da classe EventoAcademico.

* @author RoqueMaitino

*

*/

```
public class DemoEventoAcademico {
```

```
    /**
```

* O método main permite a execução desta classe. Este método contém declarações de

* algumas instâncias da classe EventoAcademico e demonstra seu uso. Algumas instâncias

* da classe Data deverão ser criadas e passadas como argumento para o construtor

* da classe EventoAcademico.

* @param argumentos os argumentos que podem ser passados para o método via linha de comando.

*/

```
public static void main (String[] argumentos) {
```

```
    EventoAcademico MaratonaDeProgramação;
```

```
    EventoAcademico SemanaDeCurso;
```

```
    Data data1 = new Data();
```

```
    Data data2 = new Data();
```

```
    data1.inicializaData((byte)20, (byte)4, (short)2017);
```

```
    data2.inicializaData((byte)20, (byte)9, (short)2017);
```

```
    MaratonaDeProgramação = new EventoAcademico  
("Maratona de Programação", "Faculdade Anhanguera de Bauru", data1,  
data2, 48);
```

```
SemanaDeCurso = new EventoAcademico ("Semana  
de Curso da Ciência da Computação", "Faculdade Anhanguera de  
Bauru", data1, data2, 200);
```

```
//Imprimimos os dados das instâncias por meio de  
chamadas implícitas ao método toString.
```

```
System.out.println(MaratonaDeProgramação);
```

```
System.out.println(SemanaDeCurso);
```

```
}
```

```
}
```

O que vale a pena saber sobre a classe `DemoEventoAcademico`:

- As instâncias da classe `EventoAcademico` só podem ser inicializadas com `new` se argumentos forem passados para seu construtor. Por isso, para essa classe, não é possível criar instâncias que tenham valores não inicializados.



Assimile

Construtores são métodos especiais que são chamados automaticamente quando instâncias são criadas por meio da palavra-chave `new`. Eles garantem que o código contido neles será executado antes de qualquer outro código em outros métodos, já que uma instância de uma classe só pode ser usada depois de ter sido criada com `new`, o que causa a execução automática do construtor.

Duas instâncias temporárias da classe `Data` são criadas na classe `DemoEventoAcademico` somente para que sejam passadas como argumentos para os construtores da classe `EventoAcademico`. Essas instâncias são reaproveitadas ou reutilizadas com valores diferentes. Assim, mesmo depois de os conteúdos das instâncias `data1` e `data2` serem modificados `data1.inicializaData((byte)20, (byte)4, (short)2017); data2.inicializaData((byte)20, (byte)9, (short)2017)`, as datas representadas internamente na instância da classe `EventoAcademico` não são modificadas (SANTOS, 2003).

Definição e utilização de sobrecarga de métodos

De acordo com Arnold, Gosling e Holmes (2007, p. 86), cada método possui uma assinatura, que consiste de seu nome e número, mais tipos de seus parâmetros. Dois métodos podem ter o mesmo nome se eles tiverem diferentes número e tipos de parâmetros e, portanto, assinaturas diferentes. Essa característica é considerada sobrecarga (ou *overload*, em inglês) porque um único nome de método possui mais de um significado sobrecarregado.

A sobrecarga consiste em permitir, dentro da mesma classe, mais de um método com o mesmo nome. Entretanto, eles necessariamente devem possuir argumentos diferentes para funcionar (ALVES, [s.d.]).

Santos (2003) ressalta que o tipo de retorno do método não é considerado parte da assinatura. Assim, não podemos ter dois métodos com o mesmo nome e tipos de argumentos, mas com tipo de retorno diferente. Ao conceituar sobrecarga, o autor assim se expressa: a possibilidade de criar mais de um método com o mesmo nome e assinatura diferente é conhecida como sobrecarga de métodos. A decisão sobre qual método será chamado quando existem dois ou mais métodos será feita pelo compilador, com base na assinatura dos métodos.

Por ora, fiquemos com o conceito. Em aula futuras, quando nossa habilidade em tratar com métodos estiver desenvolvida, voltaremos ao assunto, com exemplos e aplicações.



Pesquise mais

Um recurso interessante no contexto dos métodos sobrecarregados e sobrepostos é a invocação `this()`. Para saber mais, acesse <<http://www.javaprogressivo.net/2012/10/Auto-referencia-com-o-this-Invocando-metodos-de-Classes-e-Objetos.html>> (acesso em: 27 set. 2017).

Um bom vídeo sobre métodos construtores pode ser assistido em <<https://www.youtube.com/watch?v=-dtekFZ50Yw>> (acesso em: 24 out. 2017).

Para saber mais sobre sobrecarga, assista a <<https://www.youtube.com/watch?v=YvTijjeXJZU>> (acesso em: 24 out. 2017).

Definição e utilização de sobreposição de métodos

Para tratarmos da sobreposição de métodos, comumente chamada também de *override* ou superposição, é necessário recordar o conceito de herança, desenvolvido na Seção 1 desta unidade. Em linhas gerais, é por meio da herança que um objeto filho herdará características e comportamentos do objeto pai.

A declaração de métodos com a mesma assinatura que métodos de classes ancestrais chama-se sobreposição. O motivo para usarmos a sobreposição é que métodos de classes herdeiras geralmente executam tarefas adicionais que os mesmos métodos das classes ancestrais não executam (SANTOS, 2003).

Novamente, aqui deixaremos o desenvolvimento de exemplos e a demonstração mais detalhada das possíveis aplicações para quando tratarmos especificamente de reutilização de classes.

Atributos e métodos estáticos em classes

Ao criarmos instâncias de uma classe por meio da palavra reservada *new*, cada instância da classe terá uma cópia de todos os campos declarados na classe. Por padrão, a modificação de um campo de uma instância de uma classe não afeta o valor do mesmo campo em outra instância. Esses campos são conhecidos como campos de instância.

É por meio da declaração e do uso de campos estáticos que conseguimos compartilhar um valor em todas as instâncias de uma mesma classe. Um campo estático é também conhecido como campo de classe, já que ele está associado à classe em que ele é definido, não à instância dessa classe.

Campos estáticos são declarados com o modificador *static*, que deve ser posicionado antes do tipo de dado do campo e pode ser combinado com modificadores de acesso (*public* e *private*, por exemplo).

Exemplos de declaração de campos:

private int registroAcademico; (campo de instância)

static private int registroAcademico; (campo estático ou campo de classe).



A aplicação que segue simula o mecanismo de atendimento de um caixa bancário. O atendimento é feito por meio de uma senha que indica a ordem de atendimento. O construtor da classe atribui valor inicial ao número do caixa, inicializa o contador de clientes e imprime uma mensagem. O parâmetro *n* é o número do caixa de banco. Observe este exemplo de Santos (2003).

```
public class SimuladorDeCaixaDeBanco0 {  
    private int númeroDoCliente;  
    private int númeroDoCaixa;  
  
    SimuladorDeCaixaDeBanco0 (int n) {  
        númeroDoCaixa = n;  
        númeroDoCliente = 0;  
        System.out.println ("Caixa " + númeroDoCaixa+" iniciou  
a operação.");  
    }  
    public void próximoAtendimento() {  
        númeroDoCliente = númeroDoCliente + 1;  
        System.out.print("Cliente    com    senha    número  
"+númeroDoCliente+", favor ");  
        System.out.print("dirigir-se        ao        caixa  
"+númeroDoCaixa+".\n");  
    }  
}  
  
public class DemoSimuladorDeCaixaDeBanco0 {  
    public static void main (String [] args) {  
        SimuladorDeCaixaDeBanco0    c1    =    new  
SimuladorDeCaixaDeBanco0(1);  
        SimuladorDeCaixaDeBanco0    c2    =    new  
SimuladorDeCaixaDeBanco0(2);  
        SimuladorDeCaixaDeBanco0    c3    =    new  
SimuladorDeCaixaDeBanco0(3);  
        SimuladorDeCaixaDeBanco0    c4    =    new  
SimuladorDeCaixaDeBanco0(4);  
        SimuladorDeCaixaDeBanco0    c5    =    new  
SimuladorDeCaixaDeBanco0(5);  
        c1.próximoAtendimento();  
    }  
}
```

```

        c2.próximoAtendimento();
        c3.próximoAtendimento();
        c4.próximoAtendimento();
        c5.próximoAtendimento();
        c1.próximoAtendimento();
        c2.próximoAtendimento();
        c3.próximoAtendimento();
        c1.próximoAtendimento();
        c2.próximoAtendimento();
        c1.próximoAtendimento();
        c1.próximoAtendimento();
        c1.próximoAtendimento();
        c1.próximoAtendimento();

    }
}

```

Saída:

Caixa 1 iniciou a operação.

Caixa 2 iniciou a operação.

Caixa 3 iniciou a operação.

Caixa 4 iniciou a operação.

Caixa 5 iniciou a operação.

Cliente com senha número 1, favor dirigir-se ao caixa 1.

Cliente com senha número 1, favor dirigir-se ao caixa 2.

Cliente com senha número 1, favor dirigir-se ao caixa 3.

Cliente com senha número 1, favor dirigir-se ao caixa 4.

Cliente com senha número 1, favor dirigir-se ao caixa 5.

Cliente com senha número 2, favor dirigir-se ao caixa 1.

Cliente com senha número 2, favor dirigir-se ao caixa 2.

Cliente com senha número 2, favor dirigir-se ao caixa 3.

Cliente com senha número 3, favor dirigir-se ao caixa 1.

Cliente com senha número 3, favor dirigir-se ao caixa 2.

Cliente com senha número 4, favor dirigir-se ao caixa 1.

Cliente com senha número 5, favor dirigir-se ao caixa 1.

Cliente com senha número 6, favor dirigir-se ao caixa 1.

Cliente com senha número 7, favor dirigir-se ao caixa 1.

Problema: cada instância da classe `SimuladorDeCaixaDeBanco0` tem seus próprios campos. Assim, a modificação do campo `númeroDoCliente` de uma instância da classe não modifica os valores do mesmo campo em outras instâncias após o atendimento a um cliente. Por isso, cada caixa do banco funciona de maneira independente, de tal forma que os clientes teriam senhas que só valeriam para aquela caixa.

Solução: passar o número da senha para o método `iniciaAtendimento()` e fazer com que o método retorne o valor da senha incrementado de 1, ou seja, o próximo número da senha.



Refleta

Será mesmo que esta solução é a ideal? A programação orientada a objetos não nos oferece recurso adequado para casos como esse?

Essa abordagem apresenta duas contraindicações: (i) o controle do próximo atendimento ficaria sob responsabilidade do programador-usuário, sendo que o indicado é encapsular o comportamento do caixa na classe `SimuladorDeCaixaDeBanco0` e (ii) qualquer número poderia ser passado como senha, o que poderia comprometer a simulação. Essa solução, portanto, não é a ideal.

Curioso pela solução ideal para o caso? Continue a leitura e preste bastante atenção na seção "Avançando na prática". Agora é com você. Leia com atenção a resolução da situação-problema e resolva os exercícios propostos. Bom estudo.

Sem medo de errar

A estrutura da resolução dessa situação-problema é, mais uma vez, semelhante à resolução da situação anterior. Apenas para resgatar a situação-problema apresentada, você deverá explicar aos colegas de trabalho os conceitos e as vantagens do uso de **métodos construtores e sobrecarga de métodos**.

Novamente, é necessário salientar que a resolução aqui apresentada pode ser ligeiramente diferente daquela imaginada por você. É bom lembrar também que o vídeo é a forma mais eficiente de tratar do problema, pelos motivos que expusemos em seções anteriores.

Feitos os lembretes, passamos a apresentação do roteiro de uma solução possível.

1. Apresente os métodos construtores como uma forma de se iniciar valores do objeto. Ressalte que, da forma como é estruturado no paradigma de orientação a objetos, o recurso não tem semelhante no mundo procedimental.

A equipe conhece a palavra reservada *new()* como meio natural de se iniciar um objeto. No entanto, sem que um construtor seja usado, o compilador assume que os valores do objeto devem ser inicializados com os conteúdos padrão do Java, quais sejam 0 para variáveis numéricas e falso para variáveis lógicas.

Ressalte que, com a adoção de um método construtor na classe (ou mais de um, dependendo do caso), evita-se que o programador-usuário atribua os valores que quiser ao objeto recém-criado.

Por fim, escolha um exemplo — nos moldes daquele fornecido durante a abordagem do nosso conteúdo teórico — e o explique de modo que a teoria fique devidamente ilustrada.

2. Apresente a sobrecarga de métodos como recurso oferecido pelo Java para possibilitar a definição de vários métodos com o mesmo nome, porém com assinaturas diferentes. Um bom exemplo da utilidade da sobrecarga é dado por Furgeri (2013).

Considere `System.out.println()`. O método `println` pode receber diversos dados como parâmetro, isto é, ora pode ser enviado um dado inteiro para ser impresso, ora pode ser enviado um tipo *double*, ou ainda *string*. Isso foge das regras de definição dos métodos, uma vez que ele deve receber o mesmo tipo de dado declarado. Sendo assim, como o método `println()` pode receber dados de tipos diferentes? Isso é possível por meio da sobrecarga de métodos.

A qualidade da sua apresentação será decisiva para que a equipe compreenda e adote o conteúdo. Bom trabalho!

Uma solução melhor

Descrição da situação-problema

No decorrer do nosso conteúdo teórico, foi apresentado o problema do campo `númeroDoCliente` da classe `SimuladorDeCaixaDeBanco()`. Concluímos que a solução apresentada passava longe de ser a ideal, já que não haveria sincronismo adequado (por assim dizer) na geração das senhas para atendimento na agência.

Sendo assim, é necessário que você apresente solução adequada para o problema.

Resolução da situação-problema

A resolução é bastante simples, mas ilustra o poder de certos recursos próprios da orientação a objetos. Se o número do cliente a ser atendido for declarado com *static*, somente um único valor poderá ser armazenado nele, independentemente de quantas instâncias dessa classe sejam criadas. Em outras palavras, o valor da variável será sempre mantido, o que soluciona o problema da manutenção de uma senha sequencial para atendimento aos clientes. O trecho de código que segue é parte da solução:

```
class SimuladorDeCaixaDeBanco {  
    static private int númeroDoCliente;  
    //Este campo contém o número do cliente a ser atendido.  
    //Como é declarado como static, apenas um valor  
    //poderá ser armazenado.  
    private int númeroDoCaixa;  
    SimuladorDeCaixaDeBanco (int n)  
    {  
        númeroDoCaixa = n;  
        númeroDoCliente = 0;  
        System.out.println("Caixa "+númeroDoCaixa+" iniciou  
operação");
```

```

    }

}

```

Um outro uso para campos estáticos em classes se dá por meio da criação de constantes, que serão também compartilhadas por todas as instâncias daquela classe.

A classe `ConstantesMatematicas` contém vários valores que são declarados e usados como constantes. Ela não tem métodos, e todos os campos são declarados como *static*. Assim, não é necessário criar instâncias dessa classe para acessar os valores. Para que os valores possam ser acessados de fora da classe sem a necessidade de métodos, os campos serão declarados como *public*. Para garantir que os valores não poderão ser modificados, os campos também são declarados como *final*.

Observe a classe extraída de Santos (2003):

```

public class ConstantesMatematicas {
    final static public double raizDe2 = 1.414213562373095;
    final static public double raizDe3 = 1.732050807568877;
    final static public double raizDe5 = 2.23606797749979;
    final static public double raizDe6 = 2.449489742783178;
}

public class DemoConstantesMatematicas {
    public static void main (String[] argumentos) {
        ConstantesMatematicas const1 = new
ConstantesMatematicas();
        ConstantesMatematicas const2 = new
ConstantesMatematicas();
        System.out.println(const1.raizDe2 == const2.raizDe2);
//imprime true.
        System.out.println(const1.raizDe3 == const2.raizDe3);
//imprime true.
    }
}

```

```

        System.out.println(const1.raizDe5 == const2.raizDe5);
//imprime true.
        System.out.println(const1.raizDe6 == const2.raizDe6);
//imprime true.
        double raizDe10 = ConstantesMatematicas.raizDe2 +
ConstantesMatematicas.raizDe5;
        System.out.println ("A raiz quadrada de 10 é "+
raizDe10);
    }
}

```

Note que as constantes são declaradas com o modificador `static`. O modificador `final` faz com que os valores dos campos, depois de declarados, não possam mais ser modificados, o que é necessário e desejável para campos que representam constantes.

Faça valer a pena

1. Um método *static* é invocado em nome de uma classe inteira e não sobre um objeto específico instanciado a partir dessa classe. Tais métodos também são conhecidos como métodos de classe. Um método estático pode realizar uma tarefa geral para todos os objetos da classe, como retornar o próximo número serial disponível ou algo dessa natureza (ARNOLD; GOSLING; HOLMES, 2007, p. 76).

É sabido que um método estático pode acessar somente campos estáticos e outros métodos estáticos da classe. A razão dessa restrição deve-se ao fato de que:

- a) Membros não estáticos dispensam o uso da palavra reservada *new*, já que a criação de um objeto se dá automaticamente, independentemente da vontade do programador.
- b) Membros estáticos podem ser acessados sem referência a um objeto, desde que o controle de acesso aos membros da classe seja omitido na assinatura do método.
- c) Membros não estáticos devem ser acessados por meio de uma referência a um objeto, e nenhuma referência a objeto está disponível em um método estático.
- d) Membros não estáticos devem ser acessados sem referência explícita ao um objeto, mas sempre alguma referência a um objeto estará disponível no método estático.
- e) Membros não estáticos dispensam a criação de objetos da classe, razão pela qual sempre alguma referência a um objeto estará disponível no método estático.

2. Para objetivos além da simples inicialização, classes podem ter construtores. Construtores são blocos de comandos que poder ser usados para inicializar um objeto antes que a referência ao objeto seja retornada por *new*. Construtores possuem o mesmo nome da classe que eles inicializam (ARNOLD; GOSLING; HOLMES, 2007, p. 70).

Em relação aos métodos construtores, analise as afirmações que seguem e assinale a alternativa que contém apenas indicações de afirmações verdadeiras.

- I) Métodos construtores devem ter exatamente o mesmo nome da classe a que pertencem, mas não se exige coincidência de letras maiúsculas e minúsculas.
- II) Métodos construtores devem retornar valor do tipo que se espera que os atributos sejam inicializados por ele.
- III) Métodos construtores não devem receber modificadores de acesso. Eles serão públicos se a classe for pública e assim por diante.

- a) Apenas as afirmações I e II são verdadeiras.
- b) Apenas as afirmações II e III são verdadeiras.
- c) Apenas as afirmações I e III são verdadeiras.
- d) Apenas a afirmação III é verdadeira.
- e) As afirmações I, II e III são verdadeiras.

3. Você deve usar sobrecarga prudente e cuidadosamente. Enquanto que sobrecarregar com base no número de argumentos é geralmente bastante claro, pode ser confuso sobrecarregar com base no mesmo número, mas diferentes tipos de argumentos (ARNOLD; GOSLING; HOLMES, 2007, p. 87). Assinale a alternativa que expressa correta utilização de sobrecarga de métodos.

- a) A sobrecarga é usada quando um mesmo método tem de ser chamado duas vezes, pois evita a repetição da assinatura do método.
- b) A sobrecarga é usada para que o programador possa dispensar a criação explícita de um objeto na classe em uso.
- c) A sobrecarga é usada para inicializar um objeto antes que a referência ao objeto seja retornada pela palavra reservada *new*.
- d) A sobrecarga é usada quando um método é declarado sem atributos, já que eles impediriam a sobreposição das informações.
- e) A sobrecarga é usada quando um método pode aceitar a mesma informação apresentada em formas diferentes.

Referências

ALVES, L. **Sobrecarga e sobreposição de métodos em orientação a objetos**. Disponível em: <<http://www.devmedia.com.br/sobrecarga-e-sobreposicao-de-metodos-em-orientacao-a-objetos/33066>>. Acesso em: 24 out. 2017.

ARNOLD, K., GOSLING, J., HOLMES, D. **A linguagem de programação Java**. 4. ed. Porto Alegre: Bookman, 2007.

DOUGLAS, M. **Você sabe, com certeza, o que é orientação a objetos?** Object Pascal Programming, 2015. Disponível em: <<http://objectpascalprogramming.com/posts/o-que-e-orientacao-a-objetos/>>. Acesso em: 31 ago. 2017.

FURGERI, S. **Java 7 ensino didático**. 2. ed. São Paulo: Érika, 2013.

HOUAISS, A.; FRANCO, F. M. M.; VILLAR, M. S. **Dicionário Houaiss da Língua Portuguesa**. São Paulo: Objetiva, 2001.

MACHADO, H. **Os 4 pilares da programação orientada a objetos**. DevMedia, 2015. Disponível em: <<http://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264>>. Acesso em: 30 ago. 2017.

MATTANO, G. **Entendendo a estrutura de um código Java**. Disponível em: <<http://www.devmedia.com.br/entendendo-a-estrutura-de-um-codigo-java/24622#ixzz3GmnOTDLM>>. Acesso em: 15 set. 2017.

ORACLE. **Primitive data types**. Disponível em <<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>>. Acesso em: 14 set. 2017.

SANTOS, R. **Introdução à programação orientada a objetos usando Java**. Rio de Janeiro: Campus, 2003.

THE UNIVERSITY OF TENNESSEE. **A brief history of object-oriented programming**. Disponível em: <<http://web.eecs.utk.edu/~huangj/CS302S04/notes/oo-intro.html>>. Acesso em: 31 ago. 2017.

Estruturas de programação orientadas a objetos

Convite ao estudo

Aqui iniciamos a segunda unidade do nosso livro didático. Mais uma vez, você é nosso convidado especial, e esperamos que o bom aproveitamento do conteúdo leve-o a um substancial acréscimo em seu conhecimento em Java.

Na primeira unidade, você teve contato com os fundamentos da orientação a objetos, e imaginamos que quase tudo lá tenha sido novidade, e pouco associado àquilo que você já conhecia sobre programação de computadores. Nas próximas duas seções, no entanto, os pontos de contato entre o novo conhecimento e aquele que você já tem serão bem mais evidentes.

Nesta unidade em particular, será por meio do desenvolvimento da sua capacidade em usar as estruturas condicionais e de repetição que você será capaz de implementar uma ordem de serviço utilizando o que a orientação a objeto oferece como recurso. Em resumo, estudaremos nesta unidade:

- Estruturas de decisão e controle em Java, incluindo operadores aritméticos, lógicos e relacionais, bem como os comandos *if* e *switch*;
- Estruturas de repetição em Java, incluindo contadores, acumuladores e os comandos *while*, *do-while* e *for*;
- Reutilização de classes em Java, com modificadores de acesso, herança, polimorfismo e encapsulamento.

Por falar nisso, foi por causa dos seus valiosos recursos técnicos e interpessoais que o dono da empresa autorizou o investimento necessário para a mudança do paradigma e da linguagem de programação utilizados na empresa. Além disso,

you conseguiu fazer com que seu time técnico aderisse com entusiasmo o desafio de aprender outra forma de se fazer programas. Embora uma etapa importante do processo tenha sido cumprida, ele ainda se encontra longe de estar finalizado. Sua competência em transmitir às equipes seus conhecimentos na linguagem Java será agora colocada à prova. Como os recursos colocados à disposição do departamento de TI não são vultosos, o treinamento da equipe deverá ser feito pelo próprio gerente do setor que, no caso, é você mesmo. Nesta unidade, você deverá habilitar seu time a usar recursos básicos da linguagem Java, incluindo estruturas de seleção, repetição, herança e polimorfismo. Como resultado desta etapa do processo, o programa Java que implementa o controle da ordem de serviço deverá ser criado, conforme já mencionado.

Será que as estruturas de seleção e de repetição do Java são mera repetição das que conhecemos do paradigma procedimental? As características próprias da orientação a objetos se manifestam, por exemplo, na estrutura condicional? Essas questões serão respondidas em breve.

Persevere nos estudos, e sucesso em mais esta unidade.

Seção 2.1

Estruturas de decisão e controle em Java

Diálogo aberto

O contexto de aprendizagem desafia o gerente de TI, que é você mesmo, a treinar a equipe nos recursos básicos da linguagem Java, habilitando-a para a criação da nova ordem de serviço da empresa.

Derivada desse contexto de aprendizagem, a situação-problema propõe o seguinte: o treinamento da equipe começará com a apresentação da estrutura de decisão em Java e sua implementação por meio dos comandos *if-else* e *switch*. O domínio dessa estrutura é parte do que a equipe precisa para superar o desafio desta seção.

Como responsável pela criação de novos programas, você será desafiado a criar o esboço da classe (ou classes) que implementa a ordem de serviço. As informações mínimas que devem constar da ordem de serviço é seu número, a data, o nome do cliente requisitante, a descrição do serviço, o responsável (ou responsáveis) pela execução do serviço, o produto utilizado, sua respectiva quantidade e preço e, por fim, valor total da ordem de serviço. Seu principal propósito é formalizar e registrar um serviço a ser prestado a um determinado cliente. Nesse sentido, é de se esperar que ela quem? ou o quê? se relacione com a classe dos clientes e dos materiais a serem utilizados no serviço.

Esse exercício compõe a primeira parte do treinamento que você ministrará à sua equipe. O código-fonte, juntamente com breve explicação de seus principais itens, deverá ser apresentado em um relatório impresso ou em forma de arquivo.

Desafio aceito? Adiante, pois.

Não pode faltar

Ao tratarmos de “programa de computador” logo nas primeiras linhas do conteúdo de nossa aula inicial, definimos a expressão como o *meio pelo qual um computador executa cálculos e funções*. Mais adiante, durante a exposição do conteúdo sobre métodos, entendemos serem

eles os principais responsáveis pela realização das funções dos nossos programas.

Assim, falta entender, em sentido amplo, como os cálculos são feitos em um programa. Será que contamos com recursos para compararmos logicamente duas expressões? Podemos relacionar grandezas? Depois de cumprida essa etapa inicial — e respondidas essas perguntas —, passaremos a tratar da estrutura de seleção (ou de decisão) por meio de seus comandos *if-else* e *switch*. Sigamos adiante!

Operadores aritméticos, lógicos e relacionais

Operadores aritméticos são aqueles capazes de realizar operações aritméticas entre operandos. É bom que se registre, aliás, que um operando é uma entrada (ou argumento) de um operador. Por exemplo, na operação $8 * 5 = 40$, $*$ é o *operador* de multiplicação, e 8 e 5 são os *operandos*. A quantidade de operandos determina a *aridade* da operação. O nome pode parecer estranho, mas fará sentido quando identificarmos operadores como *unários*, *binários*, *terciários* e assim por diante.

O Quadro 2.1 resume os operadores aritméticos em Java e que operam sobre qualquer um dos tipos primitivos numéricos.

Quadro 2.1 | Operadores aritméticos do Java

Símbolo	Operação
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto

Fonte: elaborado pelo autor.

Da utilização desses operadores e operandos nascem as expressões em Java. Nas palavras de Arnold, Gosling e Holmes (2007, p. 209), uma expressão consiste de operadores e seus operandos, que são avaliados para fornecer um resultado. Esse resultado pode ser uma variável, um valor ou mesmo nada.



A quantidade de operandos de uma operação determina se ela é unária (um operando apenas), binária (dois operandos), terciária (três operandos) e assim por diante.

Antes de abordarmos exemplos de expressões, é necessário que tratemos do operador de atribuição em Java. O operador de atribuição = atribui o valor da expressão de seu operando à direita ao seu operando da esquerda, que deve ser uma variável (ARNOLD; GOSLING; HOLMES, 2007). Por enquanto, devemos entender que o tipo da expressão precisa ser compatível com o tipo da variável para efeito de atribuição de valor.

Se tivermos, por exemplo, declarado uma variável **a** como sendo do tipo inteiro, uma forma de atribuição possível é **a = 7**. Se **b**, **c** também forem declaradas como variáveis do tipo inteiro, então **c = b = a = 7** também será uma atribuição válida. Nesse caso, todas as variáveis assumem o valor 7.

**Exemplificando**

Existe uma expressão aritmética tão comum nos programas, que acabou recebendo até um nome próprio: **contador**. Embora ele possa ser escrito em mais de uma forma na linguagem Java, a mais usual é **c = c + 1**, em que **c** é uma variável do tipo inteiro e com nome definido pelo programador. Essa expressão incrementa em 1 o valor atual da variável (**c**, nesse exemplo) a cada ocorrência de um certo evento. Embora ela faça sentido em um programa de computador, não tente resolvê-la como se fosse uma equação de primeiro grau.

Os operadores lógicos são expressos, como segue na Quadro 2.2.

Quadro 2.2 | Operadores lógicos do Java

Símbolo	Descrição
&&	Retorna verdadeiro se os dois valores (à esquerda e à direita do operador) forem verdadeiros.
	Retorna verdadeiro se ao menos um dos dois valores (à esquerda e à direita do operador) for verdadeiro.
!	Negação.

Fonte: elaborado pelo autor.

O símbolo && implementa o operador “e”. O operador || implementa o operador “ou”. Ambos são usados, via de regra, para combinarem mais de uma operação relacional, que será abordada na sequência.

Operadores relacionais são capazes de estabelecer relação entre duas grandezas (ou valores). Em outras palavras, os operadores relacionais comparam dois valores, e essa comparação gera uma resposta do tipo booleano, ou seja, verdadeira ou falsa.

As comparações possíveis são descritas no Quadro 2.3.

Quadro 2.3 | Comparações em Java

Símbolo	Descrição
==	Retorna verdadeiro se os dois valores comparados forem iguais.
!=	Retorna verdadeiro se os dois valores comparados forem diferentes.
>	Retorna verdadeiro se o valor à esquerda do sinal for maior do que o valor à direita do sinal.
<	Retorna verdadeiro se o valor à esquerda do sinal for menor do que o valor à direita do sinal.
>=	Retorna verdadeiro se o valor à esquerda do sinal for maior ou igual do que o valor à direita do sinal.
<=	Retorna verdadeiro se o valor à esquerda do sinal for menor ou igual do que o valor à direita do sinal.

Fonte: adaptado de Santos (2003, [s. p.]).



Refleta

O símbolo = pode ser considerado uma simplificação do símbolo ==? A resposta é um retumbante não! O símbolo = (um símbolo de igual apenas) realiza uma operação de atribuição, ao passo que o símbolo == realiza uma comparação entre valores.

Uma expressão que pode ser entendida como exemplo para a combinação “valor operador_relacional valor” é distância >= 200. Nesse caso, a variável distância tem o seu valor atual comparado com o valor constante 200. Essa expressão retornará verdadeiro somente se o valor contido em distância for maior ou igual a 200. Caso contrário, o retorno será falso.

A abordagem detalhada do comando *if-else* apresentará vários outros exemplos de como os operadores relacionais são utilizados na prática.

Bem, estamos quase terminando a abordagem das operações possíveis em Java. No entanto, faltaria algo importante se não tratássemos de tipos de dados antes de finalizarmos. No decorrer das aulas, você terá um contato mais próximo com tipos de dados, mas cabe aqui uma explicação introdutória do tema.

De acordo com Oracle (2015), os oito tipos de dados primitivos suportados pelo Java são:

- *byte*: trata-se de um tipo que suporta números inteiros sinalizados (armazena positivos e negativos) de 8 *bits*. Tem um valor mínimo de -128 e um valor máximo de 127 (inclusive). O tipo de dados *byte* pode ser útil para “economizar” memória, já que ocupa pouco espaço.

- *short*: o tipo de dados *short* é um inteiro sinalizado de 16 *bits*. Tem um valor mínimo de -32.768 e um valor máximo de 32.767 (inclusive). Da mesma forma que no tipo *byte*, você pode usar um *short* para economizar memória em situações nas quais isso realmente importa.

- *int*: por padrão, o tipo de dados *int* é um inteiro sinalizado de 32 *bits* que tem um valor mínimo de -2³¹ e um valor máximo de 2³¹ -1. No Java SE 8 e posterior, você pode usar o tipo de dados *int* para representar um inteiro sinalizado de 32 *bits*, que tem um valor mínimo de 0 e um valor máximo de 2³². Use a classe *Integer* para usar o tipo de dados *int* como um inteiro sem sinal.

- *long*: o tipo de dados *long* armazena um número inteiro de 64 *bits*. O tipo *long* sinalizado tem um valor mínimo de -2⁶³ e um valor máximo de 2⁶³ -1. No Java SE 8 e posterior, você pode usar um *long* para representar um valor de 64 *bits* sem sinal que tenha um valor mínimo de 0 e um valor máximo de 2⁶⁴ -1. Use esse tipo de dados quando precisar de um intervalo maior do que aqueles fornecidos pelo *int*.

- *float*: o *float* é um tipo que suporta ponto flutuante de precisão de 32 *bits*. Sua faixa de valores está além do escopo dessa discussão. Como com as recomendações para *byte* e *short*, use um *float* (em vez de duplo) se você precisar economizar memória em certas aplicações de ponto flutuante.

- *double*: o *double* é um tipo que suporta ponto flutuante de precisão de 64 *bits*. Sua faixa de valores está além do escopo dessa discussão. Para valores decimais, esse tipo de dados geralmente é a opção padrão.

- *boolean*: o tipo de dados *boolean* tem apenas dois valores possíveis: *true* e *false*. Use esse tipo de dado para sinalizadores simples que assumam condições verdadeiras/falsas. Esse tipo de dados representa um *bit* de informação.

- *char*: o tipo de dados *char* é um único caractere Unicode de 16 *bits*. Tem um valor mínimo de `'\u0000'` (ou 0) e um valor máximo de `'\uffff'` (ou 65.535 inclusive).

Introdução a estruturas de decisão

A menção a estruturas de decisão (ou estruturas condicionais) faz referência aos comandos de uma linguagem que permitem que o programa, em tempo de execução, decida entre dois caminhos possíveis. Dependendo da situação, é cabível que a escolha recaia em um caminho entre vários possíveis. Conforme veremos em detalhes mais adiante, são essas variações que determinarão o comando a ser utilizado.

Furgeri (2013) nos ensina que as estruturas condicionais existem em todas as linguagens de programação e possibilitam que a execução de um programa seja desviada de acordo com certas condições. Essas condições mencionadas pelo autor são expressas em comparações entre grandezas. Essas comparações, é bom lembrarmos, dão-se por meio dos operadores “igual”, “diferente”, “maior que”, “menor que”, “maior ou igual a” e “menor ou igual a”.

As comparações entre duas variáveis e/ou a comparação entre uma variável e uma constante servem de “portas de entrada” para que o programa decida qual caminho tomar. Observe este exemplo escrito em pseudo-linguagem:

Se peso < 500 então elevadorLiberado senão elevadorBloqueado.

Ainda que seja prematuro entendermos o comando em si (aguarde mais algumas linhas para compreender o exemplo por completo), é necessário que você compreenda a comparação feita no trecho “peso < 500”. A variável peso contém determinado valor que já lhe fora atribuído previamente em linhas anteriores a essa, seja por entrada pelo teclado, seja por atribuição via programa.

O valor atual de peso será comparado com a constante 500. Se, e somente se, a variável peso contiver valor menor que 500, o elevador será liberado. Em outras, palavras, se peso for menor que 500, o

fluxo do programa seguirá pelo caminho do elevadorLiberado. Caso contrário, seguirá pelo caminho do elevadorBloqueado.

Ainda nas palavras de Furgeri (2013), os comandos condicionais (ou ainda instruções condicionais) usados em Java são *if-else* e *switch-case*. Essas duas estruturas de desvio existentes na linguagem possibilitam executar diferentes trechos de um programa com base em certas condições, conforme exemplificado.

Na sequência, trataremos em detalhes dos dois comandos de decisão que o Java nos oferece. Tudo do que tratamos nesta seção até o momento fará mais sentido a partir do estudo dos comandos *if-else* e *switch*, pois eles darão sentido prático às operações que acabamos de estudar. Sigamos adiante.

O comando *if-else* para programação orientada a objetos

Não seria exagero tratar o *if-else* como um comando universal, já que ele está disponível — com uma ou outra variação de formato — em todas as linguagens de programação. Sinteticamente, ele possibilita que o programa tome uma decisão sobre qual caminho o fluxo do programa deverá tomar com base naqueles mesmos preceitos que estudamos em “Introdução a estruturas de decisão”. Contudo, mais importante do que “teorizar” o comando é cuidar para que não falte um exemplo esclarecedor da sua real função.

A primeira variação — e menos comum — do comando é seu formato sem o complemento *else*. Vejamos o formato geral do comando *if*

```
if (expressão_de_teste)
{
    Comando1;
    Comando2;
    Comandon;
}
```

Comando1, comando2, comandon só serão executados se a expressão_de_teste retornar verdadeiro. Expressão_de_teste será sempre uma comparação entre duas grandezas (valores) por meio de uma ou mais expressões relacionais.

Vamos ao nosso primeiro exemplo. A aplicação que segue emite

em tela a mensagem "Você está apto(a) a obter CNH" se a idade informada for maior ou igual a 18 anos.

```
import java.util.Scanner;

public class IFSimples {

    public static void main(String args[]) {

        Scanner entrada = new Scanner (System.in);
        byte idade;
        System.out.print("Informe sua idade: ");
        idade = entrada.nextByte();
        if (idade >=18) System.out.print("Você está apto(a) a
obter CNH.");
    }

}
```

Repare que o programa não executará qualquer ação se a comparação retornar falso, ou seja, se a idade for menor que 18 anos. O recurso que supre essa deficiência é justamente a segunda variação do comando, mais completa e largamente utilizada: o *if-else*. Observe seu formato geral:

```
if <expressão_de_teste>
{
    comando1;
    comando2;
    comandon;
}

else
{
    comando1;
    comando2;
    comandon;
}
```

Caso a *expressão_de_teste* seja avaliada como verdadeira, o bloco logo abaixo do cabeçalho do comando será executado. Caso a *expressão_de_teste* seja avaliada como falsa, o bloco situado logo abaixo do *else* será executado.

Uma observação importante deve ser feita nesse ponto: em nenhum caso o bloco do *if* e o bloco do *else* serão exercitados numa mesma execução do programa. A execução do bloco de comandos do *if* exclui a execução do bloco de comandos do *else*.

A aplicação que segue lê dois valores inteiros e os imprime em ordem crescente. Observe a utilização do complemento *else*, ele determina a impressão primeiro do valor de *a* e depois do valor de *b* se o teste (*a >= b*) retornar falso.

```
import java.util.Scanner;

public class Crescente {
    public static void main(String args[]) {
        Scanner entrada = new Scanner (System.in);
        int a, b;
        System.out.println("Informe o primeiro valor: ");
        a = entrada.nextInt();
        System.out.println("Informe o segundo valor: ");
        b = entrada.nextInt();
        if (a >= b) System.out.printf("%d - %d",b,a); else System.
out.printf("%d - %d",a,b);

    }
}
```

Seriam essas as duas únicas formas de expressarmos o comando condicional *if*? Leia com atenção o próximo quadro.



Pesquise mais

Nos casos em que o objetivo de uma avaliação de expressão pelo *if* é o de determinar que valor será atribuído a uma variável, o operador *?* pode ser usado com vantagem. O operador *?* é conhecido também como operador ternário. Trata-se de uma simplificação do comando *if-else*, mas que usa apenas um comando e não blocos de comandos.

Pesquise e saiba mais em <<https://www.dm.ufscar.br/~waldeck/curso/java/part26.html#ifcompact>>. Acesso em: 17 out. 2017.

O comando *switch* para programação orientada a objetos

Se o comando *if-else* é usado nos casos em que é necessário decidir por um caminho entre dois possíveis, o comando *switch* (ou *switch-case*) simplifica as coisas para o programador quando existem mais de dois caminhos na decisão.

Forma geral do comando:

switch (variável inteira)

```
{  
    case valor1:comando1; break;  
    case valor2:comando2; break;  
    case valorn:comandon;  
    default: comando;  
}
```

Importante:

- O comando *break* é opcional. Ele faz com que o fluxo de execução seja transferido para a primeira linha após o bloco do comando *case*.
- Para avaliar mais de um valor válido no *case*, faça, por exemplo, *case 'e': case 'E':*.
- A constante associada ao *case* deve ser um *int*, *char* ou *byte*.

Aí está o conteúdo básico de estrutura condicional. Dispense toda a sua atenção agora à resolução da situação-problema e não deixe de resolver os exercícios.

Sem medo de errar

A situação-problema desta seção propõe o esboço de uma ou mais classes que implementam o modelo da ordem de serviço da empresa. Como dados mínimos da ordem de serviço, foram sugeridos: número, data, nome do cliente, requisitante, descrição do serviço, responsável pela execução do serviço, produto utilizado e sua respectiva quantidade e preço e, por fim, valor total da ordem de serviço. É necessário registrar que essa atividade é parte importante da construção do produto desta unidade.

A resolução desse problema deve ser desenvolvida no IDE Eclipse,

ferramenta que já lhe é familiar por ter sido utilizada em atividades no laboratório. No Eclipse, será possível editar a classe e corrigir eventuais (e prováveis) erros de escrita do código, além de colocar a resolução no formato de entrega requerido na descrição da situação.

O nome da classe deve refletir, na medida do possível, sua funcionalidade. Não se acanhe em escolher nomes extensos, mas se lembre sempre das convenções utilizadas para criar identificadores.

Uma vez criada a estrutura da classe, defina os atributos da classe e seus tipos, com base nos dados mínimos sugeridos no enunciado do problema.

Pronto! Basta revisar a edição do código e apresentá-lo.

Avançando na prática

Simplificando o código

Descrição da situação-problema

Um colega de trabalho recém-contratado foi designado para revisar alguns programas da empresa. A revisão inclui a simplificação e padronização de alguns trechos de código Java sem que, no entanto, seja incluída ou alterada alguma funcionalidade.

Em sua primeira tarefa, ele se deparou com um programa simples que retorna ao usuário a categoria da qual um nadador faz parte, com base nos seguintes critérios

Quadro 2.1 | Categoria dos nadadores

Faixa etária	Categoria
10 a 13	Infantil
14 a 16	Juvenil
17 a 20	Júnior
21 a 30	Adulto
31 a 99	Veterano
Fora das faixas	Idade inválida

Fonte: elaborado pelo autor.

Eis o código:

```
import java.util.Scanner;

public class Categoria {
    public static void main (String args[]) {
        int idade;
        String cat;
        Scanner entrada = new Scanner(System.in);
        System.out.print("Informe a idade do competidor: ");
        idade = entrada.nextInt();
        if (idade >= 10 && idade <=13) cat = "Infantil";
        else
            if (idade >= 14 && idade <= 16) cat = "Juvenil";
            else
                if (idade >= 17 && idade <= 20) cat =
"Júnior";
                else
                    if (idade >= 21 && idade <=
30) cat = "Adulto";
                    else
                        if (idade >= 31 &&
idade <= 99) cat = "Veterano";
                        else
                            cat = "Idade
inválida";
                    System.out.printf("A categoria do competidor é %s.",
cat);
                }
        }
    }
```

Por meio da alteração de um comando Java por outro, você deverá ajudá-lo a tornar o código mais legível, simples e menos propenso a erros quando sua lógica ou funcionalidade precisarem ser alteradas.

Mãos à obra!

Resolução da situação-problema

Fica claro que o encadeamento excessivo de *if-else* acarreta complexidade no código, mesmo que ele esteja funcionalmente perfeito. O comando *switch* foi pensado para que situações como essa, em que uma opção deve ser escolhida entre várias, tenha sua implementação facilitada. Uma solução possível para o problema é a que segue:

```
import java.util.Scanner;

public class CategoriaSwitch {
    public static void main (String args[]) {
        int idade;
        String cat;
        Scanner entrada = new Scanner(System.in);
        System.out.print("Informe a idade do
competidor: ");

        idade = entrada.nextInt();
        switch (idade)
        {
            case 10 : case 11 : case 12 : case 13
: cat = "Infantil"; break;

            case 14 : case 15 : case 16
: cat = "Juvenil"; break;

            case 17 : case 18 : case 19 : case 20 :
cat = "Júnior"; break;

            case 21 : case 22 : case 23 : case 24 :
case 25 : case 26 : case 27 : case 28 :
case 29 : case 30
: cat = "Adulto"; break;

            default : if (idade >= 31 && idade <=
99) cat = "Veterano";

            else cat = "Idade inválida";

        }
    }
}
```

```

        System.out.printf("A categoria do competidor
é %s.", cat);
    }
}

```

Não é difícil perceber que o *switch-case* não se sai bem com intervalos. Na verdade, não há como tratar facilmente com intervalos, como demonstrado na seleção da categoria "Adulto". Usando esse comando, não há meio simples de resolver essa deficiência.

Apesar disso, o *switch-case* é ótima opção na maioria dos casos semelhantes.

Faça valer a pena

1. A estrutura *switch-case* se refere a uma outra modalidade de desvio da execução do programa de acordo com certas condições, semelhante ao uso da instrução *if*. Na primeira linha do *switch*, é avaliado o resultado da expressão, que é comparado nas diretivas *case*, executando o bloco de instruções quando a expressão coincidir com o valor colocado ao lado direito do *case* (FURGERI, 2013).

O comando *switch-case* simplifica as comparações que seriam feitas pelo *if-else* quando:

- a) Existirem mais do que dois caminhos a serem percorridos no caso de a expressão de teste retornar falso.
- b) Existir a necessidade de selecionar um caminho entre vários possíveis.
- c) Existirem mais do que dois caminhos a serem percorridos no caso de a expressão de teste retornar verdadeiro.
- d) Existir a necessidade de selecionar vários caminhos entre um possível.
- e) O comando *if-else* se mostrar insuficiente para resolver o problema da seleção do caminho.

2. O *if*, em conjunto com o *else*, forma uma estrutura que permite a seleção entre dois caminhos distintos para execução, dependendo do resultado (verdadeiro ou falso) de uma expressão lógica. Assim como a maioria das instruções em Java, o conjunto *if-else* deve ser utilizado com minúsculas e, caso haja apenas uma instrução a ser executada, tanto no *if* como no *else*, o uso das chaves é desnecessário (FURGERI, 2013).

Com base no funcionamento do comando *if-else*, analise as afirmações que seguem:

I) Quando o fluxo do programa é direcionado para o bloco de comandos do *else*, é necessária nova avaliação da condição de teste.

II) Numa mesma execução, quando o fluxo do programa é direcionado para o bloco de comandos do *if*, o bloco de comandos do *else* também é executado.

III) O comando *if*, ao contrário da sua variação *if-else*, não requer a avaliação prévia da expressão de teste.

IV) O comando *if*, assim como sua variação *if-else*, requer a avaliação prévia da expressão de teste.

Assinale a alternativa que contém apenas indicações das afirmações verdadeiras.

- a) Apenas a afirmação IV é verdadeira.
- b) Apenas as afirmações I e IV são verdadeiras.
- c) Apenas as afirmações I, II e III são verdadeiras.
- d) Apenas as afirmações I, II e IV são verdadeiras.
- e) Apenas as afirmações II e IV são verdadeiras.

3. Os operadores relacionais possibilitam comparar valores ou expressões, retornando um resultado lógico verdadeiro ou falso. Os operadores lógicos são operadores que permitem avaliar o resultado lógico de diferentes operações aritméticas em uma expressão (FURGERI, 2013).

Em relação aos operadores lógicos e relacionais do Java, assinale a alternativa que contém afirmação verdadeira.

- a) O operador `&&` retornará verdadeiro se ao menos uma das asserções que compõem a sentença for falsa.
- b) O operador `&&` retornará verdadeiro se ao menos uma das asserções que compõem a sentença for verdadeira.
- c) O operador "maior que" retorna verdadeiro se o valor à direita do sinal for maior do que o valor à esquerda do sinal.
- d) O operador "maior que" retorna verdadeiro se o valor à esquerda do sinal for maior do que o valor à direita do sinal.
- e) A única função da operação de negação é inverter o sinal do operador `||`.

Seção 2.2

Estruturas de repetição em Java

Diálogo aberto

Aqui estamos novamente! Neste encontro, você terá a oportunidade de estudar os comandos de repetição da linguagem Java. Não será difícil concluirmos que as repetições nos programas são essenciais para a criação da maioria dos algoritmos, o que confere grande importância ao bom aprendizado das estruturas de repetição em nosso contexto.

Pois bem! Cumprida a primeira parte do treinamento e tendo concluído o esboço da classe (pode ter sido criada mais que uma) que implementa a nova ordem de serviço da empresa, o desafio agora passa a ser definir os métodos que implementarão as ações na Ordem de serviço (OS). Considere a criação de uma OS, a inclusão de materiais e a totalização do valor do serviço como exemplos de ações a serem implementadas na classe.

Seu trabalho, portanto, é incluir o esboço dos métodos nas classes anteriormente iniciadas. Assim como o desafio anterior, este também deverá ser apresentado em código-fonte.

As condições para que você supere esse desafio serão dadas por meio da abordagem de quatro assuntos relacionados às estruturas de repetição em Java:

- Contadores e acumuladores: expressões que realizam contagem de determinados eventos no programa e somas sucessivas de valores respectivamente.
- Comando *while*: comando que repete determinado trecho de código enquanto certa condição estiver sendo satisfeita. Teste de condição no início do bloco.
- Comando *do-while*: comando que repete determinado trecho de código enquanto certa condição estiver sendo satisfeita. Teste de condição no final do bloco.
- Comando *for*: comando que repete determinado trecho do código em um número conhecido de vezes

Mãos à obra!

Não pode faltar

Aqui estamos para mais uma seção de Programação orientada a objetos. A última seção nos esclareceu como o Java trata as decisões que devem ser tomadas durante a execução de um programa. O comando *if-else* é capaz de selecionar um bloco de comandos entre dois existentes, dependendo da avaliação da expressão de comparação dada.

O comando *switch*, que também compõe a estrutura de decisão da linguagem, é capaz de selecionar um caminho entre vários possíveis, o que facilita o trabalho do programador quando é necessário o uso de vários comandos *if-else* encadeados.

A importância desse recurso é incontestável e, sem ele, não conseguiríamos tornar nossos programas realmente úteis. No entanto, ainda falta algo. O que o Java nos oferece quando precisamos repetir determinado trecho de um programa? Considerando que sabemos como provocar essas repetições, como definiremos a condição de parada da repetição? Só existe uma forma de usar o recurso da repetição? Continue a leitura e fique apto a responder essas e muitas outras questões. Adiante!

Contadores e acumuladores para programação orientada a objetos

Antes de iniciarmos de fato a abordagem dos comandos de repetição do Java, é necessário o entendimento dos contadores e acumuladores. Em sua essência, eles são apenas expressões aritméticas que operam em alguma variável, na ocorrência de um certo **evento**.

Complicado demais? Então imagine a seguinte situação que exemplifica o funcionamento de um contador: sua missão é descobrir quantos alunos existem em uma sala de aula. Para tanto, você resolveu se colocar na porta da sala, com papel e caneta em mãos. Para cada aluno que deixa a sala (o aluno deixando a sala é o **evento**), você anota um número sequencial. O primeiro aluno a sair é o número 1, o segundo é o número 2 e assim por diante. Pronto! O último número anotado corresponde à quantidade de alunos na sala.

Para ilustrar o funcionamento dos acumuladores (ou somadores),

a situação é parecida: cada aluno na sala tem um número inteiro estampado na camisa. Sua missão é descobrir a soma dos números e, para isso, você também se postou na porta da sala, com papel e caneta em mãos. Quando o primeiro aluno sair da sala, você deve anotar o número que aparece na camisa. Quando o segundo deixar a sala, você deverá somar o número da camisa dele com o número da camisa do aluno anterior. O procedimento então se repete: o número do aluno atual deve ser somado com o valor sucessivamente **acumulado**.

Para que as situações fiquem mais próximas da realidade computacional, entenda que as ações de anotar no papel, contar e somar se traduzem em atribuição de valor a uma variável e processamento de expressões aritméticas.

Observe o exemplo que segue:

```
import java.util.Scanner;

public class Contador {
    public static void main (String args[]) {
        Scanner entrada = new Scanner(System.in);

        int i, c=0, valor;
        for (i=1;i<5;i++) {
            System.out.printf("\nDigite o %d valor: ",i);
            valor = entrada.nextInt();
            if(valor > 10) c = c + 1;
        }

        System.out.printf("A quantidade de valores maiores que 10
é %d\n",c);
    }
}
```

Ele ilustra o uso de um contador. A cada ocorrência do evento `valor > 10`, uma unidade é incrementada na variável `c`. Ao término do programa, a variável `c` conterá a quantidade de valores digitados maiores que 10.

O segundo exemplo ilustra o uso de uma expressão acumuladora:

```
import java.util.Scanner;

public class Acumulador {
```



```

public static void main (String args[]) {
    Scanner entrada = new Scanner(System.in);
    int i, a=0, valor;
    for (i=1;i<5;i++) {
        System.out.printf("\nDigite o %d valor: ",i);
        valor = entrada.nextInt();
        a = a + valor;
    }
    System.out.printf("A soma dos valores informados é
%d\n",a);
}
}

```

A variável *a*, que é inteira e tem valor inicial 0, recebe seu próprio valor adicionado do conteúdo da variável *valor* cada vez que o fluxo do programa passa por ela. Trata-se, portanto, da obtenção do resultado por meio de somas sucessivas.

Essas duas expressões serão muito úteis por todo nosso curso. Vale registrar ainda que o comando *for* e sua função, em ambos os exemplos, serão explicados mais além nesta seção.

A linguagem Java disponibiliza muitos recursos para serem utilizados em expressões aritméticas, principalmente como contadores e acumuladores. O Quadro 2.4 resume esses recursos.

Quadro 2.4 | Operadores Java

Operador	Descrição
++	Aplicado a inteiros ou ponto flutuante. Incrementa a variável em 1. Exemplo: <i>a++</i> equivale a <i>a=a+1</i> . O operador ++ pode ser usado antes ou depois do nome da variável que vai receber o incremento. Como o operador, além de incrementar o conteúdo da variável, retorna o valor modificado, a ordem dos termos é relevante. Imagine que a variável <i>a</i> tenha o valor 2. A expressão <i>soma = a++</i> faz com que a variável <i>soma</i> permaneça com o valor 2, já que a atribuição foi feita antes do incremento. Já pela expressão <i>soma = ++a</i> , ambas as variáveis valerão 3.
+=	Simula um somador (ou acumulador). Exemplo: <i>soma+=n</i> equivale a <i>soma = soma + n</i> .

--	Decrementa o valor da variável em 1. Pode ser aplicado antes ou depois da variável, e a regra do operador ++ continua valendo aqui.
-=	Semelhante ao +=.
=	Multiplica o valor atual da variável pela constante ou variável dada. Exemplo: $m=4$ equivale a $m = m * 4$.
/=	Divide o valor atual da variável pela constante ou variável dada.

Fonte: adaptado de Santos (2003, p. 89).

O comando *while* para programação orientada a objetos

Aqui iniciamos a série de três comandos que compõem a estrutura de repetição da linguagem Java. Antes de detalharmos cada um deles, convém que algumas observações sejam feitas. Vamos a elas:

- (i) As repetições se dão em tempo de execução, ou seja, quando o código estiver sendo executado.
- (ii) As repetições atingem um determinado trecho de código, que pode ser composto por um ou mais comandos.
- (iii) Sempre que o trecho de código contar com mais de uma linha de código, o bloco deve ser delimitado por sinalizadores de início e fim de bloco. No Java, esses delimitadores são { e } (lê-se “abre chaves” e “fecha chaves”).
- (iv) Todos os comandos de repetição contam com uma certa condição de parada, que pode ser controlada pelo programador ou pelo próprio comando. Você verá que essa condição (e seu correto gerenciamento) é fundamental para que o comando funcione como se espera.

O comando *while* repete determinado trecho do código **enquanto** determinada condição estiver sendo satisfeita. O termo **enquanto**, aliás, é destacado justamente por ser o significado, em português, da palavra *while*.

O formato geral do comando é o que segue:

Inicialização da variável de controle;

while (teste)

{

Alteração na variável de controle;

<comando>

```
<comando>  
}
```

A estrutura do comando indica claramente que o teste de validação da condição é feito **antes de o laço ser executado**. Por isso, é legítimo entender que nenhuma repetição pode ser executada, pois o teste pode retornar falso antes mesmo que o comando possa repetir o trecho que vem logo após o teste.

Se (teste) retornar verdadeiro, o corpo do laço é executado uma vez, a expressão de teste é avaliada novamente e, se ainda retornar verdadeiro, o trecho é novamente executado. A inicialização e a alteração da variável de controle são necessárias para que, em algum momento e sob certa condição, o teste retorne falso e o laço não seja mais executado.

Observe o exemplo que segue. Ele ilustra uma forma de como **não** se deve criar um laço while:

```
public class ContraExemploWhile {  
  
    public static void main (String args[]) {  
        int a=100, b=100;  
        while (a>0) {  
            b = b - 1;  
            a = a + 1;  
            System.out.printf("\n%d", a);  
            System.out.printf("\n%d", b);  
        }  
    }  
}
```

Repare na variável *a*. Ela foi inicializada com o valor 100 e, antes de o laço de repetição se iniciar, seu conteúdo foi comparado com 0. A leitura que se faz do cabeçalho do comando é “enquanto o valor de *a* for maior que 0, repita...”. No interior do bloco que será repetido, “enquanto o valor de *a* for maior que 0”, o valor de *a* é incrementado em 1 a cada iteração. Essa operação de incremento faz com que a variável jamais seja menor ou igual a 0, o que mantém o retorno

da comparação $a > 0$ sempre verdadeiro. Qual o resultado disso? A ocorrência do que chamamos “loop infinito”. Por causa do “loop infinito” você vai precisar interceder para interromper o programa, já que ele nunca será concluído pela forma normal.



Refleta

É legítimo entender o comando *while* semelhante ao comando *if*, mas com o poder de repetir o trecho do código?

Uma característica bem forte do comando *while* é que, via de regra, não se conhece antecipadamente a quantidade de iterações (ou repetições) que será desenvolvida no laço. Vamos a um exemplo:



Exemplificando

O código que segue demonstra a utilização do comando *while* para cálculo do fatorial de um número informado pelo usuário. As repetições (ou a repetição, no singular, dependendo do número informado) criadas pelo comando *while* serão executadas enquanto o valor informado for maior ou igual a 2. No corpo do bloco, a variável que contém o valor informado é decrementada em 1 a cada repetição do laço. Na mesma iteração, a variável que conterá o fatorial ao final do processamento é multiplicada pelo valor atual da variável *num*.

```
import java.util.Scanner;

public class Fatorial {

    public static void main (String args[])
    {

        Scanner entrada = new Scanner(System.in);
        int fator, num;
        System.out.println("Entre com um valor: ");
        num = entrada.nextInt();
        if (num >= 0) {
            if (num <= 1) {
                System.out.print("\nResposta: 1");
            }
            else
```

```

        {
            fator = num;
            while (num>=2)
            {
                num = num-1;
                fator = fator * num;
            }
            System.out.printf("\nResposta:  %d",
fator);
        }
    } else System.out.print("\nNúmero inválido!");
}
}

```

Na sequência trataremos do segundo comando de repetição desta seção. Funcionalmente, ele é bem parecido com o *while*, mas com uma diferença fundamental. Vamos descobrir qual?

O comando *do-while* para programação orientada a objetos

O funcionamento do laço condicional *do-while* guarda muita semelhança com o do comando *while*. No entanto, segundo Furgeri (2013), o conjunto de instruções é executado antes da avaliação da expressão lógica. Isso faz com que essas instruções sejam executadas pelo menos uma vez.



Assimile

O comando *do-while* executa ao menos uma vez o trecho a ser repetido. Após a execução, o teste da expressão lógica é feito.

Observe que o formato geral do comando *do-while* é bem parecido com o do comando *while*:

```
do
{
    Instrução
    Instrução
} while (teste);
```

Não há quantidade máxima de instruções que compõem o laço. No entanto, esse comando também requer que blocos com mais de uma instrução sejam delimitados por { e } (lê-se “abre chaves” e “fecha chaves”).

Antes de desenvolvermos um exemplo que ilustre o comando, vale uma observação: as aplicações não obrigam o uso do comando *while* no lugar do comando *do-while* ou vice-versa. Em outras palavras, o programador pode escolher a utilização de um ou outro, dependendo do seu estilo de programação ou, quando muito, da conveniência. Por exemplo, se o algoritmo demanda que ao menos uma execução do bloco de repetição seja feita, então a escolha natural e mais conveniente é a *do-while*.



Exemplificando

O exemplo que segue ilustra a utilização do comando *do-while* para o cálculo da média dos valores digitados pelo usuário. Ele deverá informar o valor -1 para que a entrada termine.

```
public class LacoDoWhile {
    public static void main (String args[]) {
        Scanner entrada = new Scanner(System.in);
        float media = 0, valor = 0, soma = 0;
        int qtd=0;

        do
        {
            System.out.printf("Digite o %d valor [-1 encerra]: ",qtd+1);
            valor = entrada.nextInt();
```

```

        if (valor != -1) {
            qtd=qtd+1;
            soma=soma+valor;
        }

    } while (valor != -1);
    if (qtd != 0) media = soma/qtd;
    System.out.printf("\nA média entre o(s) %d valor(es) digitado(s)
foi = %.2f", qtd, media);
    System.out.print("\n");
}
}

```

O exemplo nos revela aspectos importantes dessa estrutura:

- i) Como o funcionamento do algoritmo está atrelado ao valor informado pelo usuário, é de se esperar que ao menos um valor deva ser informado, obrigatoriamente. Daí a utilização do comando *do-while* em vez do comando *while*;
- ii) A média dos valores informados é calculada com base na quantidade de valores informados e em sua soma. Daí a necessidade de usarmos um contador de quantidade ($qtd=qtd+1$) e um somador de valores ($soma=soma+valor$);
- iii) O valor informado -1 não deve ser considerado como entrada, mas como condição de parada do laço. Também, por isso, os comandos de contagem de quantidade e de somatória de valores não devem ser executados na ocorrência da digitação do -1.

Bem, até o momento, o controle das repetições foi feito pelo programador por meio dos comandos *while* e *do-while*. A interrupção do laço de repetição, portanto, se deu pela não satisfação de determinada condição.



Pesquise mais

Assista ao vídeo a seguir e observe a explicação prática sobre a diferença na utilização do comando *while* e do comando *do-while*.

Disponível em: <<https://www.youtube.com/watch?v=jywFb1-rlcA>>.

Acesso em: 30 out. 2017.

A pergunta que se faz agora é: o Java oferece comando que realiza quantidade conhecida e previamente determinada de repetições, sem que o programador precise controlá-las?

O comando *for* para programação orientada a objetos

O comando *for* é conhecido por executar repetições controladas por variável. Essa variável é controlada pelo próprio comando. Ele repete determinado trecho de código em um número determinado de vezes.



Assimile

A principal característica do comando *for* é a quantidade previamente conhecida de repetições que o bloco de comandos sofrerá.

Dessa característica, deriva outra: como as repetições acontecem em um número sabido e determinado de vezes, espera-se que não haja interrupção durante as repetições. Em outras palavras, se não houver interferência externa, a quantidade de repetições será a que o programador determinou, independentemente do valor da entrada fornecida pelo usuário durante a execução do programa.



Exemplificando

O exemplo que segue calcula a tabuada de um número fornecido pelo usuário. Independentemente da entrada fornecida pelo usuário, a quantidade de repetições será sempre 10 neste caso. A leitura do comando *for* (*i*=1; *i*<11; *i*++) pode ser feita assim: “faça a variável *i* iniciar com o valor 1 e, enquanto a variável *i* for menor que 11, incremente-a em 1”.


```
import java.util.Scanner;
public class Tabuada {
    public static void main (String args[]) {
        Scanner entrada = new Scanner(System.in);
        byte i, num;
        System.out.print("Digite o número do qual você deseja
calcular a tabuada [1..10]: ");
        num = entrada.nextByte();
        for (i=1;i<11;i++)
            System.out.printf("\n%d x %d = %d", num, i, i*num);
        }
    }
}
```

É isso! Sempre conte com a certeza de conseguir material extra sobre esse conteúdo em bons livros de Java, vídeos e artigos. Passaremos agora para a resolução da situação-problema e, na sequência, para a proposição de exercícios.

Sem medo de errar

Depois de terminado o esboço da classe que implementa a nova OS da empresa, sua missão agora é definir os métodos que implementarão as ações na OS. A situação-problema descrita no início da seção nos orienta a descrever a inclusão de materiais na OS e a totalização do valor do serviço. A apresentação, mais uma vez, vai se dar por meio do código-fonte.

A resolução desse problema deve ser desenvolvida no IDE Eclipse, ferramenta que já lhe é familiar por ter sido utilizada na criação da classe. No Eclipse, será possível editar os métodos e corrigir eventuais (e prováveis) erros de escrita do código, além de colocar a resolução no formato de entrega requerido na descrição da situação.

Os nomes dos métodos devem refletir, na medida do possível, a principal funcionalidade de cada um deles. Não se acanhe em escolher nomes extensos, mas lembre-se sempre das convenções utilizadas para criar identificadores.

Um exemplo possível para a ordem de serviço da empresa é o que segue:

Quadro 2.3 | Exemplo de Ordem de Serviço

ORDEM DE SERVIÇO			
Empresa		Ordem nº	125
		Data	13/11/2017
CNPJ 00.000.000/00001-00		Cliente	
Rua Dois, 2		Fulano de Tal	
Ararinhas, SP.		CPF 999.999.999-99	
Telefone (99) 9999-5555		Rua das Palmeiras, 1	
		Ararinhas, SP	
Descrição do serviço	Local da prestação	Endereço de cobrança	
Instalação de porta de vidro	Rua das Palmeiras, 1	O mesmo	
Instalação de vidro em janelas	Ararinhas, SP		
Quantidade	Serviço Prestado	Preço unitário	Total
		Subtotal	
		Desconto	
Forma de pagamento		Acréscimos	
Entrada para 30 dias mais prestações		Total	

Fonte: elaborado pelo autor.

As informações que constam nessa OS podem servir como base para você criar seu código. Observe que os campos reservados para quantidade, serviço prestado, preço unitário e total contêm várias linhas, o que sugere a utilização de estrutura de repetição na elaboração do algoritmo.

Avançando na prática

Ajuda ao colega de trabalho

Descrição da situação-problema

Seu colega de trabalho recebeu a incumbência de criar uma aplicação que, dados os valores de base e expoente, devolvesse o valor de b^e (lê-se b elevado a e). Depois de várias tentativas e muito tempo despendido, ele chegou ao código que segue:

```

import java.util.Scanner;

public class BaseExpoente {
    public static void main (String args[]) {
        Scanner entrada = new Scanner(System.in);
        int i, base, expoente, res=1;
        System.out.print("Informe o valor da base: ");
        base = entrada.nextInt();
        System.out.print("Informe o valor do expoente: ");
        expoente = entrada.nextInt();
        for(i=0;i<expoente;i++)
            res = res + base;
        System.out.printf("O valor da operação é %d", res);
    }
}

```

No entanto, embora ele estivesse certo que as entradas dos valores estavam sendo feitas corretamente, o programa não retornava o valor correto. O valor correto, aliás, era obtido em operação feita numa calculadora comum. Estava claro que o processamento não estava de acordo com as especificações para o cálculo de uma base elevada a um expoente.

Sua missão, então, é ajudar o colega a obter o valor correto da operação por meio da correção do algoritmo dado.

Resolução da situação-problema

A operação essencial para a obtenção do resultado desejado é a multiplicação sucessiva da base por ela mesma, "expoente vezes". Por exemplo, o cálculo de 43 implica na multiplicação sucessiva do valor quatro, três vezes. Esse pressuposto já torna inválida, para fins de resolução do problema, a linha que contém a expressão $res = res + base$;

Em vez de um somador, a linha deveria conter um multiplicador, o que faria que ela contivesse $res = res * base$. Vale ressaltar que o valor de res é iniciado com 1 no ato da sua declaração, já que 1 é o elemento neutro da multiplicação.

Por último, é necessário registrar que, antes da colocação do

comando *for*, é preciso incluir um comando *if* que vai avaliar se o expoente informado é 0. Se essa comparação retornar verdadeiro, o algoritmo deverá retornar 1 como resultado da operação, já que toda base elevada a 0 resulta em 1.

Feitas essas alterações, o programa vai se parecer com o que segue:

```
import java.util.Scanner;
```

```
public class BaseExpoente {
```

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

```
        Scanner entrada = new Scanner(System.in);
```

```
        int i, base, expoente, res=1;
```

```
        System.out.print("Informe o valor da base: ");
```

```
        base = entrada.nextInt();
```

```
        System.out.print("Informe o valor do expoente: ");
```

```
        expoente = entrada.nextInt();
```

```
        if (expoente == 0) res = 1;
```

```
        else
```

```
            for(i=0;i<expoente;i++)
```

```
                res = res * base;
```

```
        System.out.printf("O valor da operação é %d", res);
```

```
    }
```

```
}
```

Faça valer a pena

1. Uma instrução de repetição (ou de loop) permite especificar que um programa deve repetir uma ação enquanto alguma condição permanece verdadeira. A instrução de pseudocódigo

Enquanto houver mais itens em minha lista de compras

Compre o próximo item e risque-o da minha lista.

descreve a repetição que ocorre durante um passeio de compras (DEITEL; DEITEL, 2010).

Analise as afirmações que seguem sobre a utilização do comando *while* ou do comando *do-while* em determinada aplicação.

I) A utilização de um ou outro comando depende da quantidade de comandos que o programador inclui no bloco de repetição.

II) A utilização de um ou de outro comando depende da necessidade ou não de ao menos uma execução dos comandos do bloco.

III) A utilização de um ou de outro comando depende da existência ou não de um outro laço de repetição no interior do primeiro laço.

Assinale a alternativa que contém apenas indicação de afirmações verdadeiras.

- a) Apenas a afirmação I é verdadeira.
- b) Apenas as afirmações I e II são verdadeiras.
- c) Apenas a afirmação II é verdadeira.
- d) Apenas as afirmações I e III são verdadeiras.
- e) Apenas as afirmações II e III são verdadeiras.

2. A linguagem Java oferece três instruções de repetição (também chamadas instruções de loop), que permitem que programas executem instruções repetidamente, contanto que uma condição (chamada condição de continuação de loop) permaneça verdadeira. As instruções de repetição são as instruções *while*, *do-while* e *for* (DEITEL; DEITEL, 2010).

Analise o código que segue e assinale a alternativa que contém o valor final da variável *n* e impresso por meio do comando `System.out.println(n)`.

```
public class Questao {  
    public static void main(String[] args) {  
        int i, n=0;  
        for (i=2;i<5;i++)  
            if (i % 2 != 0) n++;  
            else n = n + 2;  
        System.out.println(n);  
    }  
}
```

- a) 5.
- b) 4.
- c) 3.
- d) 6.
- e) 2.

3. Você usa o comando *for* básico para fazer um laço sobre um intervalo de valores, do início até o fim. A expressão de inicialização permite que você declare e/ou inicialize variáveis de laço e é executada somente uma vez.

Em seguida, a expressão de laço é avaliada e, se ela for *true*, o comando no corpo do laço é executado (ARNOLD; GOSLING; HOLMES, 2007).

Analise o código que segue e assinale a alternativa que contém a correta descrição do seu funcionamento.

```
import java.util.Scanner;

public class Questao3 {
    public static void main (String args[]) {
        Scanner entrada = new Scanner(System.in);
        int i, x=0, valor;
        for (i=1;i<9;i++) {
            System.out.printf("\nDigite o %d valor: ",i);
            valor = entrada.nextInt();
            if(valor > x) x = valor;
        }
        System.out.printf("%d\n",x);
    }
}
```

- a) O código avalia 8 valores informados pelo usuário e imprime o mais próximo de x.
- b) O código avalia 9 valores informados pelo usuário e imprime o menor deles.
- c) O código avalia 9 valores informados pelo usuário e imprime o maior deles.
- d) O código avalia 8 valores informados pelo usuário e imprime o menor deles.
- e) O código avalia 8 valores informados pelo usuário e imprime o maior deles.

Seção 2.3

Reutilização de classes em Java

Diálogo aberto

Caro aluno, aqui estamos às voltas com mais um desafio.

Quem já desenvolveu programas de computador certamente já desejou reaproveitar, no programa atual, algumas estruturas criadas em programas anteriores, com o mínimo de adaptações. Os recursos para reutilização de classes que o Java oferece permitem ao programador não só reutilizar classes e seus membros, como também, a possibilidade de obter vantagens com isso. Nesta seção, é exatamente a reutilização de classes que será nosso tema central, dividido nos seguintes tópicos.

- **Modificadores de acesso:** característica da orientação a objeto que permite ao programador ocultar classes e seus membros, tornando controlada sua utilização por outros programadores.

- **Herança:** mecanismo que permite a criação de uma nova classe que estende uma outra já definida pelo programador, o que torna possível a reutilização de dados e comportamentos da classe ancestral.

- **Polimorfismo:** por meio do polimorfismo, um objeto pode ser referenciado de várias formas, o que confere flexibilidade ao desenvolvimento da solução.

- **Encapsulamento:** característica que permite que as variáveis da classe e seus métodos sejam agrupados em conjuntos, segundo o seu grau de relação.

Depois de ter aprendido a criar uma aplicação em Java, chegou a hora de você construir a ordem de serviço executável. Entenda “ordem de serviço executável” como a aplicação completa que cria e permite a utilização da ordem de serviço, com classes, métodos e atributos completamente definidos.

Assim, você deve criar o método `main()` e, uma vez fornecidos os dados do cliente e os materiais e serviços que compõem o pedido, o programa deverá apresentar em tela a ordem de serviço, com número gerado automaticamente.

Para efeito de aplicação imediata, esse exercício será considerado o trabalho final do treinamento que você tem ministrado à sua equipe. Como esta é a última atividade da unidade, é necessário que a evolução do trabalho fique evidente. Para isso, você deve entregar, além do código-fonte, um breve texto com relato do que foi codificado nesta tarefa e que permitiu a efetiva execução da aplicação. Inclua nesse texto ao menos uma imagem da tela de cadastramento da ordem de serviço.

Preparado? Ao trabalho!

Não pode faltar

Aqui estamos para mais uma seção de programação orientada a objetos, a última desta unidade. Neste encontro, trataremos de assuntos de grande importância no contexto das novidades do paradigma de orientação a objetos. Vamos adiante!

Modificadores de acesso para programação orientada a objetos

Se cada membro de cada classe e objeto fosse acessível a qualquer outra classe e objeto, então o entendimento, a depuração e a manutenção de programas seriam tarefas quase impossíveis. Um dos pontos mais interessantes da programação orientada a objetos é seu suporte ao encapsulamento e ocultação de dados.

Essa característica é implementada por meio dos modificadores de acesso, aplicados em classes, métodos e campos. Com ela, os campos das classes ficam protegidos de alterações indevidas de outros programadores. Observe o código que segue, adaptado de Santos (2003):

```
/**
```

```
 * A classe DemoDataSimples demonstra usos da classe
 * DataSimples, em especial os problemas que podem ocorrer
```

```
 * quando os campos de uma classe podem ser acessados
 * diretamente.
```

```
 *
```

```
 *
```



```

*/
public class DemoDataSimples {
/**
    * O método main permite a execução desta classe. Este método
    contém declarações de algumas instâncias da classe
    * DataSimples e demonstra como seus campos podem ser
    acessados diretamente, já que são públicos.
    ** @param args
    */
    public static void main (String[] args)
    {
        // Criamos duas instâncias da classe DataSimples,
        usando new.
        // As instâncias serão associadas a duas referências,
        que permitirão
        // o acesso aos campos e métodos das instâncias.

        DataSimples hoje = new DataSimples();
        DataSimples    independênciaDoBrasil    =    new
DataSimples();
        byte umDia, umMês; short umAno; // e três variáveis
        para receberem o dia, mês e ano para as datas.
        umDia = 40; umMês = 1; umAno = 2017;
        hoje.inicializaDataSimples(umDia, umMês, umAno);
        // inicializa os campos da instância.
        hoje.mostraDataSimples(); //imprime 0/0/0.
        // Inicializando "independênciaDoBrasil como uma
        data válida
        umDia = 7; umMês = 9; umAno = 1822;
        independênciaDoBrasil.inicializaDataSimples(umDia,
umMês, umAno);
        independênciaDoBrasil.mostraDataSimples(); //
        imprime 7/9/1822.
    }
}

```

```
//testando o método élgual()
if (hoje.élgual(independênciaDoBrasil)) System.out.
println("As datas são iguais");
else System.out.println("As datas são diferentes");
// o problema: podemos facilmente "invalidar" datas
válidas acessando os seus campos diretamente.
```

```
hoje.dia = 0;
hoje.mês = 3;
hoje.ano = 2017;
```

```
hoje.mostraDataSimples(); //imprime 0/3/2017
independênciaDoBrasil.mês = 13;
independênciaDoBrasil.mostraDataSimples(); //
imprime 7/13/1822.
```

```
} //fim do método main()
} // fim da classe DemoDataSimples.
```

Informações importantes sobre essa classe:

- Duas instâncias da classe DataSimples são declaradas. (DataSimples hoje = new DataSimples(); DataSimples independênciaDoBrasil = new DataSimples();)
- Para receber dia, mês e ano, três variáveis de tipos compatíveis com os definidos na classe DataSimples são declarados. Eles não devem ser inicializados com a palavra-chave new.
- Métodos que foram definidos dentro das classes podem ser executados usando a notação nomeDaReferência.nomeDoMétodo(argumentos), mas somente se a instância tiver sido criada com new.



Pesquise mais

Para que você possa de fato executar o exemplo anterior, consulte a obra: SANTOS, Rafael. **Introdução à programação orientada a objetos usando Java**. Rio de Janeiro: Campus, 2003. Na página 25 do livro, está disponível à classe `DataSimples`. Mãos à obra!

A execução de um método é também conhecida como envio de mensagem a objetos. Na linha `hoje.inicializaDataSimples(umDia, umMês, umAno)`, a classe está enviando a mensagem `inicializaDataSimples` ao objeto `hoje` com os argumentos `umDia`, `umMês`, `umAno`.



Refleta

Note que os campos das classes também podem ser acessados diretamente a partir de referências a instâncias das classes usando o operador ponto na forma `nomeDaReferência.nomeDoCampo`. Esse seria um problema? Não tenha dúvida que sim!

A solução para o caso passa pela restrição de acesso aos campos para que o programador usuário não possa acessá-los diretamente. Os campos poderiam apenas ser modificados por meio dos métodos. No exemplo, isso garantiria que datas inválidas teriam dia, mês e ano valendo zero.

Segue a descrição dos quatro modificadores de acesso disponibilizados pelo Java:

public: garante que o campo ou método da classe declarado com esse modificador poderá ser acessado ou executado a partir de qualquer outra classe.

private: campos ou métodos declarados como `private` só podem ser acessados, modificados ou executados por métodos da mesma classe, sendo completamente ocultos para outros programadores.

protected: funciona como o `private`, mas permite que **classes herdeiras** também tenham acesso ao campo ou método declarado com esse modificador de acesso.

sem modificadores: nesse caso, serão considerados como pertencentes à categoria *package* ou *friendly*. Seus campos e métodos podem ser acessados em todas as classes de um mesmo pacote.

Introdução à herança para programação orientada a objetos

A herança é uma das características mais interessantes da orientação a objetos, e seu efeito imediato é tornar possível a reutilização de classes. Por meio de um procedimento bastante simples, será possível, por exemplo, criarmos uma nova classe que estende uma outra já definida pelo programador e, assim, aproveitarmos todo o potencial desse recurso.

Herança é uma característica do paradigma de orientação a objetos por meio da qual um objeto filho herda características e comportamentos do objeto mãe. É pela implementação da herança que conseguimos estabelecer relação entre uma classe mãe e uma classe filha, de modo que a segunda se torne uma extensão da primeira, herdando diretamente os métodos e os atributos públicos da classe mãe.



Assimile

Herança é uma forma de reutilização de software na qual uma nova classe é criada, absorvendo membros de uma classe existente e aprimorada com capacidades novas ou modificadas (DEITEL, 2005).

Alguns exemplos de herança podem ser observados no Quadro 2.5. Note que as superclasses tendem a ser mais gerais, e as subclasses mais específicas. Por exemplo, as classes `AlunoDeGraduacao` e `AlunoDePosGraduacao` herdam características da classe `Aluno`.

Quadro 2.5 | Exemplos de herança

Superclasse	Subclasses
Aluno	AlunoDeGraduacao, AlunoDePosGraduacao
Forma	Circulo, Retangulo, Triangulo
Financiamento	FinanciamentoDeCarro, FinanciamentoDeCasa
Funcionário	Gerente, ChefeDeSetor
ContaBancaria	ContaCorrente, ContaPoupanca

Fonte: adaptado de DEITEL (2005, p. 302).

Ao tomarmos a superclasse `Funcionário` e suas subclasses, `Gerente` e `ChefeDeSetor` são descritas como uma especialização de `Funcionário`. Usando nomenclatura mais usual, teremos que a classe `Funcionário` é uma superclasse de `Gerente` e `ChefeDeSetor`, e

Gerente e ChefeDeSetor são subclasses de Funcionário. É interessante destacar que herança é sempre utilizada em Java, mesmo que não explicitamente. Quando uma classe é criada e não há nenhuma referência à sua superclasse, implicitamente a classe criada é derivada diretamente da classe *Object* (RICARTE, 2000).

O conceito não parece ser complicado, não é mesmo? Como se dá a implementação em Java? Seu conhecimento sobre classes e objetos adquiridos em aulas passadas será decisivo na implementação de herança em Java. Tomaremos como primeiro exemplo uma classe chamada Ponto. Em seu estado mais simples, ela serve para representar pontos em um plano bidimensional (ARNOLD; GOSLING; HOLMES, 2007).

```
public class Ponto {  
    public double x, y;  
    public void inicializa() {  
        Ponto inferiorEsquerdo = new Ponto();  
        Ponto superiorDireito = new Ponto();  
        Ponto pontoCentral = new Ponto();  
        inferiorEsquerdo.x = 0.0;  
        inferiorEsquerdo.y = 0.0;  
        superiorDireito.x = 1280.0;  
        superiorDireito.y = 1024.0;  
        pontoCentral.x = 640.0;  
        pontoCentral.y = 512.0;  
    }  
}
```

Fonte: adaptado de Arnold, Gosling e Holmes (2007, p. 39).

Cada objeto Ponto é único e possui sua própria cópia de x e y, que são inicializados em três regiões diferentes do plano.

Pois bem, imagine agora que nossa necessidade seja mostrar um pixel em tela em vez de um ponto no plano. Além das coordenadas x e y, esse pixel requer mais um dado: a cor. É evidente o estreito relacionamento que existe entre a tela e o plano e entre o pixel e o ponto,

de modo que podemos aproveitar características e comportamentos da classe ponto na classe pixel. Vejamos como:

```
public class Pixel extends Ponto {  
    Cor cor;  
    public void limpa() {  
        super.limpa();  
        cor = null;  
    }  
}
```

Fonte: adaptado de Arnold, Gosling e Holmes (2007, p. 49).

Repare na palavra reservada *extends*, colocada na linha de descrição da classe Ponto. Ela torna a classe Pixel uma extensão da classe Ponto e permite que seus dados e comportamentos sejam usados na primeira. Os demais elementos da classe Pixel, incluindo a palavra reservada *super*, serão tratados adiante.

Nosso segundo exemplo estabelece relação entre as classes funcionário e gerente, conforme introduzido no começo desta seção. O código que segue foi retirado de Caelum – Apostila Java e Orientação a Objetos.

```
public class Funcionario {  
    String nome;  
    String cpf;  
    double salário;  
    // Os métodos devem vir aqui.  
}
```

Fonte: <<https://www.caelum.com.br/apostila-java-orientacao-objetos/heranca-reescrita-e-polimorfismo/#repetindo-codigo>>. Acesso em: 31 out. 2017.

A classe Funcionario contém campos que são comuns a qualquer tipo ou variação de funcionário que se possa encontrar em uma organização. Esses campos são nome, cpf e salário. Sem a aplicação da herança, teríamos uma classe gerente, como a que segue:

```

public class Gerente {
    String nome;
    String cpf;
    double salario;
    int senha;
    int numeroDeFuncionariosGerenciados;

    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
    }

    // outros métodos
}

```

Fonte: <<https://www.caelum.com.br/apostila-java-orientacao-objetos/heranca-reescrita-e-polimorfismo/#repetindo-codigo>>. Acesso em: 31 out. 2017.



Reflita

Repare que os campos nome, cpf e salário tiveram de ser repetidos na classe Gerente. E se tivéssemos que repetir boa parte do código a cada criação de tipos de funcionários?

Caelum (s.d.) completa: se um dia precisarmos adicionar uma nova informação para todos os funcionários, precisaremos passar por todas as classes de funcionário e adicionar esse atributo. A solução natural para esse caso é a aplicação da herança. Observe a classe Gerente, reescrita para estender a classe Funcionario.

```

public class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;
    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
    }
}

```

Fonte: <<https://www.caelum.com.br/apostila-java-orientacao-objetos/heranca-reescrita-e-polimorfismo/#repetindo-codigo>>. Acesso em: 31 out. 2017.

Apenas os campos `senha` e `numeroDeFuncionariosGerenciados` foram acrescentados, já que fazem parte do contexto da classe `Gerente`.



Pesquise mais

Assista ao vídeo e note que em pouco menos de 17 minutos, a relação de herança entre as classes `professor` e `aluno` é minuciosamente exemplificada. Disponível em: <[link](#)>. Acesso em: 31 out. 2017.

Fica claro, então, que a utilização da herança permite que classes mais genéricas sejam reaproveitadas para compor classes mais específicas, o que confere segurança e racionalidade ao processo de desenvolvimento de aplicações.

Polimorfismo em programação orientada a objetos

Compreendemos herança como a transmissão de dados e operações de uma classe genérica para uma classe específica. O

que não foi dito é que a criação de classes derivadas estabelece uma relação *é-um-tipo-de*. Por exemplo, um Funcionário *é-um-tipo-de* Pessoa e um Aluno De Pós-Graduação *é-um-tipo-de* Aluno.

É justamente a relação *é-um-tipo-de* que permite a existência do **polimorfismo**. O polimorfismo permite a manipulação de instâncias de classes que herdam de uma mesma classe ancestral de forma unificada: podemos receber métodos que recebam instâncias de uma classe C, e os mesmos métodos serão capazes de processar instâncias de qualquer classe que herde da classe C, já que qualquer classe que herde de C *é-um-tipo-de* C (SANTOS, 2003, p. 140)

Caelum (s.d.) assevera que polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas, o que não quer dizer que o objeto fica se transformando. Ao contrário disso, um objeto nasce de um tipo e morre daquele tipo. O que pode mudar é a maneira como nos referimos a ele.



Exemplificando

Tomemos este trecho de código adaptado de Santos (2003) como exemplo:

```
public class ConcessionariaDeAutomoveis {
    public static void main (String[] args) {
        Automovel a1 = new Automovel ("Fiat", "Bege", Automovel.
        MOVIDOAAALCOOL);
        AutomovelBasico a2 = new AutomovelBasico ("Corsa",
        "Cinza", Automovel.MOVIDOAGASOLINA);
        AutomovelBasico a3 = new AutomovelBasico ("Gol",
        "Branco", Automovel.MOVIDOAGASOLINA, false, false, true);
        AutomovelDeLuxo a4 = new AutomovelDeLuxo
        ("Citroen", "Preto", Automovel.MOVIDOAGASOLINA);
        AutomovelDeLuxo a5 = new AutomovelDeLuxo ("Toyota",
        "Prata", Automovel.MOVIDOAGASOLINA, true, true, false,
        true, true, true);

        imprime (a1);
        imprime (a2);
        imprime (a3);
        imprime (a4);
```

```

        imprime (a5);
    }
    public static void imprime (Automovel a){
        System.out.println ("Seguem os dados do automóvel
escolhido:");

        System.out.print (a);
        System.out.println("Valor: "+a.quantoCusta());
        System.out.println(a.quantasPrestações() + " prestações de
"+ (a.quantoCusta() / a.quantasPrestações()));
    }
}

```

Imagine que nossa concessionária de automóveis informe o preço e o número de prestações de seus carros por meio de impressão dos dados em tela. Imagine também que a concessionária é bem pequena e tem a venda apenas cinco automóveis. O método main – aquele que permite a execução da classe – contém declaração de instâncias da classe Automovel, AutomovelBasico e AutomovelDeLuxo e as usa para mostrar o valor da prestação de cada automóvel.

O método imprime mostra os dados de uma instância de qualquer classe filha de Automovel, e serão chamados os métodos quantoCusta, quantasPrestações e toString para compor a informação no momento da impressão. Está, assim, demonstrada a capacidade da programação orientada a objetos de implementar métodos que podem assumir várias formas. Está demonstrado o polimorfismo.

Encapsulamento em programação orientada a objetos

Iniciamos este último tópico da seção com as seguintes questões: encapsulamento é sinônimo de ocultação de informações? Há ao menos alguma relação entre os dois termos? Como nosso enfoque sobre o tema será conceitual, melhor será reunirmos algumas citações.

No contexto em que tratam de elementos básicos de **métodos**, Arnold, Gosling e Holmes (2007, p. 41) ensinam que os verdadeiros benefícios da orientação a objetos provêm do ocultamento da implementação de uma classe atrás das operações realizadas sobre seus dados. Ocultar dados atrás de métodos de modo que sejam inacessíveis a outros objetos é a base fundamental do encapsulamento de dados.

Os mesmos autores, em outro contexto, relacionam os **modificadores de acesso** ao encapsulamento e à ocultação de dados como meios de controle de acesso às classes e seus membros.

Para iniciar a diferenciação entre ocultação e encapsulamento, Fernandes (s.d.) desenvolve o seguinte raciocínio: o encapsulamento é um conceito da programação orientada a objetos em que o estado de objetos (as variáveis da classe) e seus comportamentos (os métodos da classe) são agrupados em conjuntos, segundo o seu grau de relação. O autor completa que a ocultação está relacionada à restrição de acesso a alguns componentes de objetos, tornando-os disponíveis apenas por meio de métodos.

De qualquer maneira, tanto a ocultação quanto o encapsulamento serão bastante úteis em encontros futuros e serão mais bem desenvolvidos na prática. Por ora, continue firme na leitura e dispense máxima atenção aos exercícios. Bom estudo!

Sem medo de errar

Depois de termos estudado os fundamentos da linguagem Java e suas estruturas de decisão e repetição, já temos condição de esboçar a ordem de serviço da empresa.

O que vem a seguir neste item do nosso livro didático compõe as diretrizes gerais de uma resolução, mas não o código-fonte em si. Assim como nas outras seções, não há aqui uma solução definitiva e única para o problema. Dependendo do seu estilo de programação e dos recursos que souber explorar da linguagem Java, sua abordagem será mais ou menos sofisticada, por assim dizer.

O que se espera do programa, no mínimo, é que:

- A aplicação tenha interatividade com o usuário, já que os dados da OS deverão ser informados pelo usuário.
- A ordem de serviço deve ser apresentada em tela, com seu número de identificação gerado automaticamente.
- As funcionalidades da OS devem ser agrupadas em um ou mais métodos.

No mais, sua criatividade e capacidade de utilizar os recursos estudados darão brilho à sua entrega. Bom trabalho!

Reutilizando código

Descrição da situação-problema

Novamente, estamos às voltas com a necessidade de ajudar um colega de trabalho. A esse colega, foi dada a incumbência de otimizar o código do programa que vem sendo usado na empresa. Ele sabe que a otimização passa pela reutilização de código, mas não sabe bem como proceder. Especificamente, o desafio inclui a remodelagem das classes hoje em funcionamento. As classes são as que seguem:

Quadro 2.5 | Classes

<pre>public class FuncionarioIntegral { String nome, matricula; int idade; float salario; String retornaNome (String nome) { return nome; } String retornaMatricula (String matricula) { return matricula; } int retornalIdade(int idade) { return idade; } float retornaSalario (float salario) { return salario; } }</pre>	<pre>public class FuncionarioHorista { String nome, matricula; int idade, totalHoras; float salarioHora; String retornaNome (String nome) { return nome; } String retornaMatricula (String matricula) { return matricula; } int retornalIdade(int idade) { return idade; } float retornaSalario (float salario) { return salario; } int retornaHoras(int totalHoras) { return totalHoras; } }</pre>
--	--

Fonte: elaborado pelo autor.

Elas foram criadas para a mesma aplicação Java e servem para modelar um funcionário de regime integral e um funcionário horista. Sua ajuda consiste em dar a ele a devida orientação de como proceder para otimizar o código.

Resolução da situação-problema

A solução para o caso inclui a utilização do recurso da herança. Por meio da criação de uma classe mais genérica (que deverá ser tida como a classe pai) chamada *Funcionario*, as classes filha *FuncionarioIntegral* e *FuncionarioHorista* receberão as características gerais da classe pai, o que resultará em maior portabilidade e facilidade geral para manutenções.

Faça valer a pena

1. A extensão de classes pode ser usada para diversos objetivos. É mais comumente usada para especialização – em que a classe estendida define novos comportamentos e assim se torna uma versão especializada da superclasse. A extensão de classe pode envolver apenas a alteração de um método herdado, talvez para torná-lo mais eficiente (ARNOLD, GOSLING e HOLMES, 2007, p. 92).

Assinale a alternativa que contém a sentença que melhor define herança no contexto de orientação a objetos.

- a) Reutilização de código por meio da supressão de métodos e atributos repetidos.
- b) Passagem automática de atributos para classes mãe.
- c) Extensão de métodos e atributos de uma classe.
- d) Reutilização de código por meio de classes abstratas.
- e) Representação de modelos com os mesmos métodos e atributos.

2. Você poderá escrever códigos realmente flexíveis. Códigos que serão mais claros (mais eficientes e simples). Códigos que não serão apenas mais fáceis de desenvolver, mas também muito mais fáceis de estender, de maneira que você nunca imaginou no momento que originalmente os escreveu. (SIERRA; BATES, 2005, p. 135).

Assinale a alternativa que contém a sentença que caracteriza corretamente o que se obtém ao se utilizar o recurso do polimorfismo.

- a) Com o polimorfismo — e apenas com ele — é possível implementar o conceito de herança.
- b) Com o polimorfismo — e apenas com ele — é possível escrever código com boa legibilidade.
- c) Apenas o polimorfismo impede que uma referência e um objeto sejam distintos, o que provocaria erro de compilação.
- d) Com o polimorfismo, é possível escrever um código que não precisa ser alterado quando novos tipos de subclasses forem introduzidos no programa
- e) Com o polimorfismo — e somente com ele — é possível implementar herança e criar código com boa legibilidade no mesmo programa.

3. Com o polimorfismo, podemos projetar e implementar sistemas que são facilmente extensíveis – novas classes podem ser adicionadas a partes gerais do programa com pouca ou nenhuma modificação, contanto que as novas classes façam parte da hierarquia de herança que o programa processa genericamente (DEITEL, 2003, p. 336).

Considerando o contexto apresentado, avalie as seguintes asserções e a relação proposta entre elas.

II) A criação de classes derivadas estabelece uma relação *é-um-tipo-de* e só pode ser feita por meio do polimorfismo.

PORQUE

II) É justamente a relação *é-um-tipo-de* que permite a implementação do polimorfismo.

A respeito dessas asserções, assinale a alternativa correta.

- a) As asserções I e II são proposições verdadeiras, e a II é uma justificativa da I.
- b) As asserções I e II são proposições verdadeiras, mas a II não é uma justificativa da I.
- c) A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.
- d) A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.
- e) As asserções I e II são proposições falsas.

Referências

ARNOLD, K.; GOSLING, J.; HOLMES, D. **A Linguagem de Programação Java**. 4. ed. Porto Alegre: Bookman, 2007.

CAELUM. **Apostila Java e Orientação a Objetos**. Disponível em: <<https://www.caelum.com.br/apostila-java-orientacao-objetos/heranca-reescrita-e-polimorfismo/#repetindo-codigo>>. Acesso em: 31 out. 2017.

DEITEL, H. M. **Java: como programar**. 6. ed. São Paulo: Pearson Education do Brasil, 2005.

———, H. M., DEITEL, P. J., **Java – como programar**. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

FERNANDES, J. **Encapsulamento em Java: primeiros passos**. Disponível em: <<https://www.devmedia.com.br/encapsulamento-em-java-primeiros-passos/31177>>. Acesso em: 6 nov. 2017.

FURGERI, S. **Java 7 ensino didático**. 2. ed. São Paulo: Érika, 2013.

ORACLE. **Primitive Data Types**. Disponível em: <<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>>. Acesso em: 17 out. 2017.

RICARTE, I. L. M. **Herança**. Disponível em: <<http://www.dca.fee.unicamp.br/courses/PooJava/heranca/index.html>>. Acesso em: 31 out. 2017.

SANTOS, R. **Introdução à programação orientada a objetos usando Java**. Rio de Janeiro: Campus, 2003.

SIERRA, K.; BATES, B. **Use a cabeça: Java**. 2. ed. Rio de Janeiro: Alta Books, 2005.

Exceções, classes abstratas e interfaces

Convite ao estudo

Aqui iniciamos a terceira unidade da nossa disciplina e, como nas anteriores, você colocará seu talento à prova a fim de resolver as situações que apresentaremos. Antes delas, porém, é preciso resgatar o contexto evolutivo da sua aquisição de conhecimento.

Soa muito natural que, ao passar da metade do curso, você imagine que tenha amadurecido um bocado na linguagem Java e que já conheça o suficiente dela para criar aplicações interessantes. O estudo dos fundamentos da linguagem (Unidade 1) e de suas estruturas (Unidade 2) ratifica sua percepção e lhe confere boas credenciais. Sim, você já é capaz de criar bons códigos!

No entanto, o Java pode oferecer mais...

A simples ideia de que a linguagem não fornece meio eficiente de controlar erros em tempo de execução (numa chamada de método, por exemplo) ou que não possibilita agrupar métodos ainda sem implementá-los poderia desencorajar quem estivesse criando aplicações mais complexas e que fossem usadas como ferramenta de trabalho para usuários e outros programadores.

Acontece que o tratamento de exceções, a criação de classes abstratas e a utilização de interfaces compõem aqueles recursos que tornam o Java poderoso e o paradigma de orientação a objetos tão, digamos, elegante.

É justamente destes recursos que trataremos nesta terceira

unidade e esperamos que, depois de tê-los estudado, você passe a conhecer e compreender a manipulação de exceções, classes abstratas e interfaces, e desenvolva a capacidade de bem utilizar estes recursos, especialmente o tratamento de exceções, que será base para o produto a ser gerado ao final da unidade.

Você acaba de ser contratado uma empresa que cria programas sob encomenda e presta manutenção em produtos já criados. Em seu primeiro dia de trabalho, o gestor da equipe em que você foi alocado lhe propõe o seguinte: refazer funções específicas de um sistema antigo utilizando recursos do paradigma de orientação a objetos, por meio da linguagem Java. Ele deseja que você programe em Java algumas funções previamente selecionadas que o sistema atual já implementa, sem, contudo, refazê-lo por completo. Com isso, ele se certificará – ou não – que a utilização de exceções, classes abstratas e interfaces trará mais facilidade de manutenção e aumentará a escalabilidade do sistema.

Instigante, não acha? Será mesmo que vale a pena refazer código que funciona? Será que a tal manutenibilidade justifica o risco de se introduzir erro em código bom? Chegaremos a uma boa conclusão, pode apostar.

A Seção 1 desta unidade define e coloca em prática o tratamento de exceções, cuja função é assegurar o correto tratamento de situações anômalas ocorridas durante a execução da aplicação. A Seção 2 aborda as classes abstratas, suas características e como elas podem nos ser úteis. Por fim, a última seção define e coloca em prática o uso de interfaces, que alguns autores definem como a maneira por meio da qual podemos “conversar” com um objeto.

Imaginamos que você esteja curioso para conhecer o que se seguirá. E não é para menos.

Bom estudo!

Seção 3.1

Definição e tratamento de exceções

Diálogo aberto

Poucas surpresas são tão desagradáveis para um usuário quanto a interrupção abrupta da aplicação que ele está operando. Se o usuário for o próprio desenvolvedor, então a surpresa deve anteceder as devidas providências para tratar o problema antes que ele se manifeste. E são exatamente dessas providências que trata esta seção, ou seja, dos recursos de que o desenvolvedor dispõe para tratar as exceções que podem ocorrer durante a execução da aplicação.

O contexto de aprendizagem apresentado na abertura da unidade nos remete a situações em que códigos já existentes deverão passar por manutenções que permitam mudar – para melhor – funções já existentes.

Este primeiro desafio consiste em implementar em Java uma função do sistema que executa a checagem da divisão de um número por outro. Tome como exemplo a divisão do valor total do serviço pela quantidade de parcelas solicitadas pelo cliente. A quantidade de parcelas, no caso, é inserida pelo operador. O sistema atual não faz a checagem de eventual divisão por zero e, caso ela ocorra, o fluxo de execução é interrompido. Você deverá apresentar o código-fonte do tratamento desta situação com exceções em Java.

O suporte para a superação deste desafio será dado no decorrer desta seção. Ela começa introduzindo o conceito de exceções, prossegue abordando os tipos de exceções e termina tratando de asserções.

Não pode faltar

Introdução a exceções em programação orientada a objetos

Numa linguagem procedimental de concepção mais antiga é comum que o tratamento de erros em tempo de execução de um programa fique restrito a uma entrada incorreta de usuário ou tentativa de gravação em arquivo de leitura apenas.

No mundo da orientação a objetos, no entanto, a quantidade

e diversidade dos erros que podem ocorrer são muito maiores. Furgeri (2013) nos ensina que exceções em Java se referem aos erros que podem ser gerados durante a execução de um programa. Como o nome sugere, trata-se de algo que interrompe a execução normal do programa.

O tratamento da exceção serve justamente para que o programa possa continuar sendo executado em vez de ser encerrado repentinamente, o que confere confiabilidade e robustez às aplicações.



Assimile

“Uma exceção é uma indicação de um problema que ocorre durante a execução de um programa. O nome “exceção” significa que o problema não ocorre frequentemente” (DEITEL; DEITEL, 2010, p. 336).

Debrucemo-nos sobre este exemplo adaptado de Deitel e Deitel (2010):

```
import java.util.Scanner;

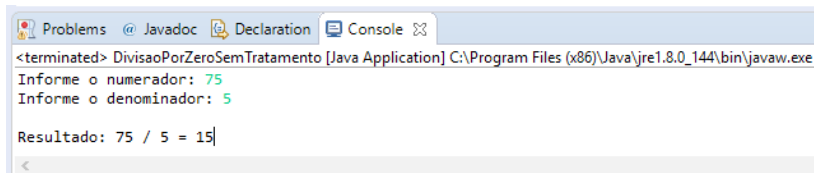
public class DivisaoPorZeroSemTratamento {
    public static int quociente (int numerador, int denominador)
    {
        return numerador / denominador;
    }

    public static void main(String args[]) {
        Scanner entrada = new Scanner(System.in);
        System.out.print("Informe o numerador: ");
        int numerador = entrada.nextInt();
        System.out.print("Informe o denominador: ");
        int denominador = entrada.nextInt();
        int resultado = quociente(numerador, denominador);
        System.out.printf("\nResultado: %d / %d = %d",
            numerador, denominador, resultado);
    }
}
```

Trata-se de um programa que recebe dois valores inteiros e

calcula, por meio do método quociente, o resultado da divisão entre os dois números.

Uma execução possível – e sem entradas com potencial para causar erro – para este programa é a que segue:



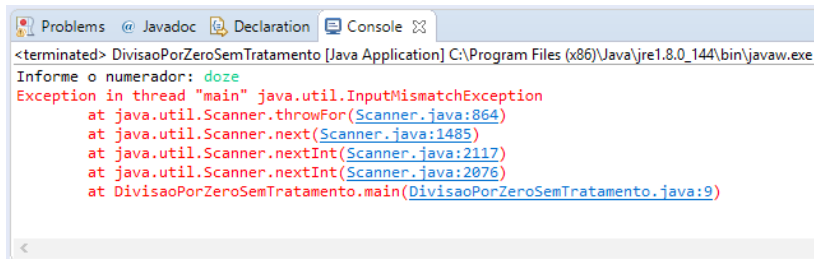
```
<terminated> DivisaoPorZeroSemTratamento [Java Application] C:\Program Files (x86)\Java\jre1.8.0_144\bin\javaw.exe
Informe o numerador: 75
Informe o denominador: 5

Resultado: 75 / 5 = 15
```

Duas observações se fazem necessárias:

- i. O resultado da operação será um valor inteiro, mesmo que o quociente não o seja.
- ii. A operação é feita sem checagem de ocorrência de erro e sem o devido tratamento em caso de ocorrência.

No entanto, o programa terminaria de forma anormal e sem retornar resultado plausível se o usuário resolvesse fornecer entradas incomuns para o algoritmo. Se, por exemplo, o numerador fosse informado como um literal ou cadeia de caracteres, a saída seria como segue:



```
<terminated> DivisaoPorZeroSemTratamento [Java Application] C:\Program Files (x86)\Java\jre1.8.0_144\bin\javaw.exe
Informe o numerador: doze
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:864)
    at java.util.Scanner.next(Scanner.java:1485)
    at java.util.Scanner.nextInt(Scanner.java:2117)
    at java.util.Scanner.nextInt(Scanner.java:2076)
    at DivisaoPorZeroSemTratamento.main(DivisaoPorZeroSemTratamento.java:9)
```

Uma exceção aritmética (`java.lang.ArithmeticException`) seria informada pelo compilador caso o numerador 0 (zero) fosse informado. Antes de entendermos como podemos contornar tais situações pelo tratamento dos erros cabe uma observação.

Em Java, as exceções são divididas em duas categorias (FURGERI, 2013):

- *Unchecked Exception*: significa “exceção não verificada”. Neste tipo de exceção o Java não verifica o código-fonte para determinar se a exceção está sendo capturada. Por isso, o

tratamento aqui é opcional. Fazem parte destas exceções de tratamento opcional, por exemplo, a verificação de acesso a um índice inexistente num vetor (trataremos de vetores na Unidade 4), a tentativa de se usar um método de um objeto ainda não instanciado e a conversão de um String em inteiro.

- *Checked Exception*: significa “exceção verificada”. Neste tipo de exceção o compilador Java obriga o programador a tratá-la. O Java verifica o código-fonte com a finalidade de determinar se a exceção está sendo capturada.

Arnold, Gosling e Holmes (2007) observam que as exceções verificadas forçam o programador a pensar sobre o que fazer com erros nos locais em que podem ocorrer no código. O não tratamento de uma exceção verificada é notificado durante a compilação e não durante a execução.

O programador pode não querer, em algumas situações, fazer o controle sobre uma exceção. A linguagem Java permite a ele que um erro seja descartado. No entanto, é preciso que essa previsão seja informada na declaração do método. Observe o exemplo que segue, adaptado de Furgeri (2013).



Exemplificando

Este trecho de código representa alternativa para entrada de dados pelo teclado sem a obrigatoriedade do tratamento da eventual ocorrência da exceção `IOException`.

```
import java.io.*;

public class UsoDoThrows {

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

        BufferedReader dado;

        System.out.println("Informe seu nome: ");

        dado = new BufferedReader(new InputStreamReader
(System.in));
```

```

        System.out.println("Seu nome é: " + dado.readLine());
    }
}

```

A cláusula `throws IOException` determina que eventual erro de entrada ou saída (IO, de Input Output) não será tratado pelo método `main`.

No Java, a estrutura que trata as exceções é formada pelos comandos `try-catch-finally`, assim dispostos:

```

try {
    comandos
} catch (exceção_tipo1 identificador1) {
    comandos
} catch (exceção_tipo2 identificador2) {
    comandos
...
} finally {
    comandos
}

```

Esta estrutura pode ser usada tanto com *Unchecked Exceptions* como com *Checked Exceptions* e tem como função desviar a execução de um programa caso ocorram certos tipos de erros, predefinidos durante o processamento das linhas. Isso evita que o programador deva criar testes de verificação ao codificar certas operações.

Façamos a análise de cada item:

`try {..}`: neste bloco são escritas todas as linhas de código que podem vir a lançar uma exceção.

`catch (tipo_excessao e) { ... }`: neste bloco é descrita a ação que ocorrerá quando a exceção for capturada.

Furgeri (2013) afirma que toda vez que a estrutura try é utilizada, obrigatoriamente em seu encerramento (na chave final) deve existir pelo menos um catch, a não ser que ela utilize a instrução finally.

Uma leitura possível para esta estrutura é: "tente executar o conjunto de instruções contidas no bloco do try. Na ocorrência de erro, execute seu tratamento no bloco do catch. Depois de tratado o erro, a execução do programa deve continuar a partir do final do último catch. O finally é opcional e fornece um conjunto de códigos que é sempre executado, independentemente da ocorrência da exceção. O uso do finally pode ser exemplificado por meio de operações de banco de dados. Algumas ações de encerramento de transação, tais como o fechamento de arquivos ou da conexão, serão sempre executadas, mesmo se uma exceção é lançada.

A cláusula throws é utilizada para delegar o tratamento de exceções para o próximo nível da pilha. Assim, o próximo método é obrigado a usar a estrutura try-catch ou delegar novamente a exceção.



Reflita

E se o programador preparar a estrutura try-catch e nenhuma exceção ocorrer durante a execução do programa?

Antes de continuarmos duas questões se fazem necessárias: todas as exceções são iguais? O programador tem controle sobre seus eventuais tipos? Prossigamos.

Criação de tipos de exceções

A linguagem Java oferece controle para geração e tratamento de muitas exceções. Nossa intenção passa longe de abordagem de todas elas, mas devemos saber que o desenvolvedor Java pode criar suas próprias exceções e dispará-las quando necessitar.

A instrução throw (que não deve ser confundida com a cláusula throws) serve para forçar a ocorrência de uma determinada exceção. Exceções são objetos e todos os seus tipos devem estender à classe Throwable ou a uma de suas subclasses. A classe Throwable possui uma cadeia de caracteres que pode servir como descritor da exceção e recuperada com o método getMessage. O método printStackTrace,

também desta classe, retorna o tipo de exceção gerado e informa em que linha da classe ocorreu o erro.



Pesquise mais

A hierarquia de exceções no Java contém muitas classes, cada uma dedicada a um tipo de exceção. Saiba mais sobre esta hierarquia no capítulo 11 da obra DEITEL, Harvey M., DEITEL, Paul J., **Java – Como Programar**. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

Como você pode imaginar, o estudo das exceções não se esgota com a abordagem do que pontuamos aqui. Antes de terminarmos, contudo, é necessário um rápido tratamento das asserções.

Definição de asserções em programação orientada a objetos

A partir de uma certa versão do Java foi criado mecanismo para que o desenvolvedor pudesse testar hipóteses que, em condições normais, não aconteceria. Nas palavras de Arnold, Gosling e Holmes (2007, p. 278), “uma asserção é usada para verificar uma invariante, ou seja, uma condição que deve ser sempre verdadeira. Se a asserção é verificada como falsa, então uma exceção é lançada.”



Refleta

Asserções são normalmente usadas para assegurar que coisas que não podem acontecer sejam notificadas quando acontecerem. A falha de uma asserção significa que o estado atual de uma operação (uma entrada, por exemplo) é anormal.

É legítimo, então, que não se cogite usar asserções para testar situações que as vezes acontecem, tais como uma IOException?

A sintaxe de uma asserção é `assert expressão_de_avaliação [: expressão_detalhe;`

Onde `expressão_de_avaliação` é uma expressão que retorna verdadeiro ou falso e `expressão_detalhe` é uma expressão opcional que será passada para o construtor de `AssertionError` para ajudar na descrição do problema.

Um breve exemplo de uma asserção nos é dado por Sabino (s.d.) no trecho de código que segue:

```
private void metodo(int num) {  
    assert (num>=0); // gera um AssertionError caso o teste não  
    seja verdadeiro  
    // aqui vem o resto do código supondo que num é positivo  
}
```

Ligando e desligando asserções

Por padrão, a avaliação das asserções permanece desligada. O desenvolvedor, no entanto, pode ligar toda a avaliação de asserções para classes e pacotes específicos. Por causa da possibilidade de serem ligadas e desligadas, o desenvolvedor deve tomar precauções a fim de evitar efeitos colaterais que afetem partes do código que não sejam de asserção (ARNOLD; GOSLING; HOLMES, 2007).

O controle de liga/desliga das asserções deve ser feito preferencialmente na linha de comando de execução da aplicação, embora possa também ser feito no código, conforme demonstrado em nosso último exemplo.

Pela linha de comando a forma geral dos comandos é a que segue:

-enableassertions /-ea [descritor]: habilita a avaliação das asserções conforme definido em descritor. Caso descritor não seja definido, as asserções são carregadas para todas as classes.

-disableassertions /-da [descritor]: desabilita a avaliação das asserções para todos os membros definidos pelo descritor. Caso o descritor não exista as asserções serão desabilitadas para todas as classes.

Então é o que temos para asserções exceções. Contudo, você não deve contentar-se apenas com este conteúdo. Busque mais informação, resolva exercícios e coloque em prática o que puder. Bom estudo!

Sem medo de errar

Conforme abordamos no início desta seção, o desafio que se apresenta é o de checar eventual divisão por zero no cálculo do valor das parcelas devidas por determinado serviço. Como no sistema atual a quantidade de parcelas é informada pelo usuário, é possível que o valor 0 seja digitado por descuido.

O esboço da solução que segue inclui apenas o código sem o tratamento da exceção de divisão por 0.

```
import java.util.Scanner;

public class Parcelas {

    public static float calculaValorParcela (float valorServiço, int
qtdParcelas) {

        return valorServiço / qtdParcelas;

    }

    public static void main (String args[]) {

        Scanner entrada = new Scanner(System.in);

        System.out.print("Informe o valor do serviço: ");

        float vs = entrada.nextFloat();

        System.out.print("Informe a quantidade de parcelas:

");

        int qp = entrada.nextInt();

        float valorParcela = calculaValorParcela (vs, qp);

        System.out.printf ("\nO valor de cada parcela é : %f",

valorParcela);

    }

}
```

Você deve criar um laço de repetição que inclua a leitura da quantidade de parcelas e o tratamento de eventual exceção por meio da estrutura *try-catch*.

Bom trabalho!

Prevenindo entradas incorretas

Descrição da situação-problema

Novamente estamos diante de um caso com potencial para ocorrer em seu trabalho atual ou no próximo. Chamado a fazer testes em programas que logo deverão ser liberados para execução em ambiente de produção, você se depara com um erro relativamente comum: o método em que se faz a entrada e processamento dos produtos da empresa permite que o usuário informe um valor literal onde deveria ser digitado uma quantidade inteira.

O trecho de código que contém o erro foi reproduzido de modo que pudesse ser executado:

```
import java.util.Scanner;

public class Produto {
    public static void main (String args[]) {
        Scanner entrada = new Scanner(System.in);
        System.out.print("Informe o código do produto [0 -
200]: ");

        int codigo = entrada.nextInt();
    }
}
```

Ao ser informado um valor literal no lugar de um inteiro, o programa simplesmente é interrompido e um erro é informado.

Sua missão é prever a exceção de entrada inválida e solicitar nova digitação do usuário.

Resolução da situação-problema

Como era de se esperar, a resolução do problema é simples, tanto quanto a detecção do problema durante um teste bem feito. O desenvolvedor deveria ter previsto – o fará agora – é a ocorrência de entrada inválida, que o usuário pode fornecer por engano ou simplesmente para “testar” o programa, a seu modo. Uma solução

possível é a que segue:

```
import java.util.Scanner;
import java.util.InputMismatchException;
public class Produto
{
    public static void main (String args[])
    {
        boolean continuacao = true;
        Scanner entrada = new Scanner(System.in);
        do
        {
            try
            {
                System.out.print("Informe o código
do produto: ");

                int codigo = entrada.nextInt();
                continuacao = false; //entrada
bem-sucedida implica em fim da leitura.
            }
            catch (InputMismatchException
erroEntradaIncorreta)
            {
                entrada.nextLine(); //ignora entrada
anterior.

                System.out.println("Você deve
informar número inteiro apenas.\n");
            }
        } while (continuacao);
    }
}
```

Leituras sucessivas do código do produto serão feitas até que um valor inteiro for informado.

Faça valer a pena

1. O objetivo das asserções é capturar erros precocemente, antes que eles corrompam coisas e causem efeitos colaterais bizarros. Quando alguém estiver executando o seu código, o rastreamento de causas de problemas é muito mais difícil, e assim capturá-las mais cedo e reportá-las claramente é muito mais importante [...] (ARNOLD; GOSLING; HOLMES, 2007, p. 282).

Assinale a alternativa que contém sentença que expressa a funcionalidade das asserções no contexto da linguagem Java.

- a) Possibilitar que as exceções sejam tratadas em tempo de execução, liberando o compilador de testá-las durante o desenvolvimento.
- b) Tornar efetivo o controle de erros, pois apenas as exceções não seriam suficientes para prever todos os tipos de erros durante a execução.
- c) Prover ao desenvolvedor forma de substituir o controle de erros via exceções, por causa da complexidade das operações envolvidas.
- d) Possibilitar testes de situações durante o desenvolvimento que sabidamente não irão ocorrer durante a execução.
- e) Transmitir ao usuário a sensação de segurança em relação à robustez e à confiabilidade da aplicação.

2. Em alguns momentos, pode ocorrer de o programador não querer realizar controle sobre uma exceção, isto é, não desejar tratar um erro. A linguagem Java permite ao programador que um ou mais erros sejam descartados, mesmo que de fato ocorram [...] (FURGERI, 2013).

Considerando as definições das palavras reservadas `throw` e `throws` do Java, assinale a alternativa que contém as expressões que completam corretamente as lacunas do texto que segue.

_____ serve para declarar as _____ e deve constar _____. Por sua vez, _____ é um comando que pode provocar um tipo de exceção.

- a) `Throws`, exceções não verificadas, no corpo do código, `throws`.
- b) `Throw`, exceções verificadas, no corpo do código, `throws`.
- c) `Throws`, exceções verificadas, na declaração do método, `throw`.
- d) `Throws`, exceções não verificadas, na declaração do método, `throw`.
- e) `Throw`, exceções verificadas, na declaração do método, `throws`.

3. O tratamento de exceções permite-lhe remover da “linha principal” de execução do programa o código de tratamento de erro, aprimorando a clareza do programa e destacando sua capacidade de modificação. Você pode decidir tratar a exceção que escolher [...] (DEITEL; DEITEL, 2010, p. 337).

Considerando o conceito e as funcionalidades de exceções em Java, analise as afirmações que seguem.

- I. A palavra reservada *finally* é opcional e fornece bloco de código que é sempre executado, independente de uma exceção ocorrer ou não.
- II. Na ocorrência do erro apontado durante o processamento das instruções presentes no *try*, automaticamente a execução do programa é direcionada para o bloco do *catch*.
- III. Na exceção não verificada o Java não verifica o código-fonte para determinar se a exceção está sendo capturada. Por isso, o tratamento do erro neste caso é opcional.
- IV. Quando uma exceção ocorre, automaticamente o Java deixa de considerar o código que vem abaixo dela e retoma a execução a partir do próximo método encontrado.

Assinale a alternativa que contém apenas indicações de afirmações verdadeiras.

- a) Apenas as afirmações I e III são verdadeiras.
- b) Apenas as afirmações I, II e III são verdadeiras.
- c) As afirmações I, II, III e IV são verdadeiras.
- d) Apenas as afirmações I e II são verdadeiras.
- e) Apenas as afirmações II e III são verdadeiras.

Seção 3.2

Definição e uso de classes abstratas

Diálogo aberto

Aqui estamos às voltas com mais uma situação a ser resolvida. Antes, porém, de tratarmos especificamente dela, vale a pena resgatarmos nosso último encontro. Visando habilitá-lo a resolver problemas relacionados a prevenção de erros e o desenvolvimento de seu raciocínio crítico, a Seção 3.1 abordou as **exceções em Java**. Por meio delas tornou-se possível preparar sua aplicação para que tratasse eventuais erros em tempo de execução, em vez de proporcionar a interrupção abrupta da execução.

Nesta seção, continuaremos a tratar de características oferecidas pela orientação a objetos que são de grande utilidade na construção de aplicações flexíveis, reutilizáveis e de manutenção facilitada. Nas próximas linhas do nosso texto serão apresentadas as classes abstratas, sua relação com herança e breve abordagem de herança múltipla. Tudo isso para habilitá-lo a superar o desafio que ora apresentamos.

Dando prosseguimento à demanda do seu gestor, neste segundo desafio você deverá criar um conjunto de classes abstratas que implementem as estruturas de dados de cliente de pessoa física e pessoa jurídica, segundo o paradigma de orientação a objetos. Em outras palavras, você deve utilizar o recurso da herança para simplificar a implementação feita pelo programa original destes dois tipos de clientes. A apresentação também deverá ser feita em código-fonte.

Não pode faltar

Seja bem-vindo(a) a mais uma seção do nosso curso!

Parece-nos evidente que o Java já deixou de ser um “bicho de sete cabeças” (será que um dia o foi?) e passou a ser um objeto de estudo bastante interessante. A dificuldade inicial na compreensão da orientação a objetos ficou no passado e, conforme você avança na teoria, também sua habilidade em resolver problemas práticos se torna mais concreta.

Por falar em coisas concretas, haverá pouco delas nesta segunda seção, já que o tema central aqui é classe abstrata, com suas características e suas relações com herança.

Não seria absurdo supor que, embora você tenha tentado, nunca tenha conseguido definir bem o termo “abstrato”? O conceito artístico de abstrato se relaciona pouco com o nosso cotidiano. O dicionário também não ajuda muito: “Resultante de abstrações. Que utiliza abstrações, que opera com qualidades e relações, e não com a realidade sensível” (FERREIRA, 1999, p. 18). Então não há outro jeito senão dedicar toda a sua atenção no que vem a seguir para se tornar um perito em abstração. Preparado?

Introdução a classes abstratas

Você sabe que o principal motivo pelo qual se modela uma classe é a criação de objetos por meio da sua instanciação. Pelo menos é assim que temos feito. No entanto, por alguns motivos plenamente justificáveis – e que serão analisados na sequência – poderemos criar classes a partir das quais não se pode gerar instâncias.

A classe abstrata é uma classe que não permite a geração de instâncias a partir dela, isto é, não permite que sejam criados objetos; ao contrário, uma classe concreta permite a geração de instâncias (FURGERI, 2013).

Usando classes abstratas o desenvolvedor pode declarar classes que definem somente parte de uma implementação, deixando para as classes estendidas o oferecimento de implementações específicas.



Assimile

Uma classe abstrata não se destina a ser instanciada, logo não precisa fornecer uma implementação completa. Em vez disso, funciona como um modelo ou padrão a partir do qual outras variáveis e métodos podem ser adicionados em subclasses (RUSSEL; ROBERTS, 2009).

Antes de tratarmos das características das classes abstratas vale dispensar nossa atenção para este exemplo: imagine que estamos modelando a classe Pessoa, com todos aqueles atributos que já conhecemos: nome, data de nascimento, endereço e estado civil, entre vários outros. Se considerarmos a necessidade da divisão do modelo “pessoa” em outros dois (pessoa física e pessoa jurídica, por exemplo), chegaríamos à conclusão de que o ente “pessoa” é capaz de agrupar alguns atributos comuns aos dois subtipos de pessoa e, por isso, pode servir de base para a geração de pessoa física e pessoa jurídica.

Assim, na condição de classe que servirá apenas como plataforma para a criação de outras, a classe Pessoa pode ser declarada como abstrata. Neste caso, sua declaração seria como segue:

```
public abstract class Pessoa {  
    // corpo da classe.  
}
```



Reflita

O que aconteceria se o desenvolvedor tentasse instanciar a classe Pessoa? Por exemplo, a linha Pessoa p = new Pessoa(); geraria erro de compilação?

Já sabemos que o uso de classe abstrata auxilia quando atributos (preferencialmente vários) podem ser definidos para outros objetos de um dado tipo. Há, contudo, a possibilidade de que sejam criados também métodos abstratos, que compartilham comportamentos como outros objetos. Assim, cada método não implementado na classe abstrata também é indicado como abstract, embora esse expediente possa ser efetivado por meio das interfaces. Estas, aliás, serão o tema central de nosso próximo encontro.

Um exemplo de declaração de método abstrato é:

```
public abstract void imprime();
```

Merece sua atenção a seguinte observação: uma classe que contém métodos abstratos deve ser declarada como uma classe abstrata, mesmo se essa classe contiver métodos concretos, ou seja, não abstratos. Cada subclasse concreta de uma superclasse abstrata também deve fornecer implementações concretas de cada um dos métodos abstratos da superclasse (DEITEL; DEITEL, 2010, p. 309).

Na sequência, durante a abordagem das características das classes abstratas, um exemplo da sua utilização será desenvolvido.

Características de classes abstratas

O exemplo que segue foi adaptado de Russel e Roberts (2009) e ilustra o uso de classes e métodos abstratos. Os comentários contidos no código são importantes para a compreensão da aplicação.



Exemplificando

```
abstract class SuperClasse1 {  
    public abstract void f();  
    public void h() {  
        System.out.println("SuperClasse abstrata, método h  
concreto. Pode ou não ser sobreposto");  
    }  
    //como a classe é abstrata, métodos estáticos não podem ser  
    declarados. Então,  
    // public abstract static void x(); provocaria erro.  
}  
  
class SubClasse1A extends SuperClasse1 {  
    //SuperClasse1A deve implementar o método abstrato  
    herdado f().  
    public void f() {
```

```

        System.out.println("Método f implementado na
SuperClasse1A e sobreposto.");
    }
}

abstract class Subclasse1B extends SubClasse1A {

    //Simples declaração de um novo método abstrato, que
deverá ser implementado em outra classe.

    public abstract void g();
}

class SubClasse1C extends SubClasse1A {

    //Esta classe é concreta e deve implementar os dois métodos
abstratos herdados.

    //Objetos desta classe podem ser criados.

    public void f() {

        System.out.println("Método f implementado na
subclasse1C");
    }

    public void g() {

        System.out.println("Método g implementado na
subclasse1C");
    }

    //O método h não precisaria ser sobreposto, mas isso é
permitido.

    public void h() {

        System.out.println("Método h implementado na
subclasse1B. Sobreposto.");
    }
}

```

A classe Abstrato2 deve ser escrita em seu próprio arquivo.

```
public class Abstrato2 {  
  
    public static void main (final String args[]) {  
  
        /*Atenção:  
  
        * objetos da classe abstrata SuperClasse1 não  
podem ser criados.  
  
        * objetos da classe abstrata Subclasse1B não podem  
ser criados.  
  
        */  
  
        final SubClasse1A s1 = new SubClasse1A();  
        s1.f();  
        s1.h();  
  
        final SubClasse1C s2 = new SubClasse1C();  
        s2.f();  
        s2.g();  
        s2.h();  
  
    }  
  
}
```

Fica claro que a característica principal das classes abstratas é não poderem passar por instanciação, conforme tratamos anteriormente neste texto.

Outra característica interessante das classes abstratas é a possibilidade de definirmos um comportamento padrão – mas com certo grau de especificidade - para um grupo de outras classes. Faremos a análise deste recurso por meio de exemplo adaptado de Furgeri (2013).

Considere a superclasse Veiculo e as subclasses Automovel e

Aviao. A superclasse Veiculo, que é abstrata, implementa (ou seja, efetivamente codifica) os métodos ligar, desligar e mostrarStatus, que representam comportamentos comuns tanto para um automóvel quanto para um avião.

Contudo, o método acelerar não é implementado na classe Veiculo. Em vez disso, ele é apenas definido. Lembre-se: situação análoga nos foi apresentada no Exemplificando, especificamente por meio do método f().

Voltando ao nosso exemplo atual, temos então que a classe abstrata Veiculo define um método abstrato chamado acelerar que deve, obrigatoriamente, ser implementado em todas as subclasses de Veiculo. Isso ocorre porque o comportamento de acelerar certamente se efetiva de modos diferentes para um avião e para um veículo.



Refleta

A orientação a objetos oferece algum outro meio de implementar um mesmo método de *formas* diferentes?

Observe o esboço do código que implementa a classe Veiculo e seus métodos.

```
public abstract class Veiculo {  
    public int velocidade;  
    public boolean status;  
    public void ligar() {  
        status = true;  
    }  
    public void desligar() {  
        status = false;  
    }  
    public void mostrarStatus() {  
        System.out.println(status);  
    }  
}
```

```
public abstract void acelerar();  
}
```

Deste código podemos extrair que:

- Não há método `main()` definido. Portanto, do jeito que se encontra, o código não poderá ser executado.
- Os métodos `ligar`, `desligar` e `mostrarStatus` são concretos, ou seja, foram devidamente implementados, embora de maneira bem simplista em nosso exemplo.
- O método `ligar` apenas simula o acionamento do motor. O método `desligar` simula o desligamento do motor. O método `mostrarStatus` exhibe em tela o estado do motor, por meio do atributo `status`.

E o método `acelerar`? Bem, um possível trecho da implementação na classe `Automovel` seria `velocidade = velocidade + 1`. Já na classe `Aviao` seria `velocidade = velocidade + 10`. Faz sentido, não acha?

Embora esta particularidade já tenha sido apresentada de alguma forma, é prudente destacar que a classe abstrata se assemelha a um modelo incompleto, que será obrigatoriamente complementado pelas classes especialistas que implementarão seus métodos abstratos. Assim, para evitar seu uso de forma concreta, ela é declarada com o modificador `abstract`.

Vale ainda a menção de que, embora as classes abstratas devam ter seus métodos construtores, eles não devem ser declarados como `abstract`. Métodos declarados com os modificadores `static` ou `final` também não podem ser abstratos.

Relação entre classes abstratas e herança

Neste ponto já nos situamos sobre a utilidade das classes abstratas. Sabemos que elas servem como base para criação de outras classes que conterão dados e comportamentos comuns. No entanto, este efeito não poderia ser conseguido simplesmente pelo uso da herança, na forma como a conhecemos?

Caelum (s.d.) argumenta que a única diferença é que não se pode instanciar um objeto do tipo da classe abstrata, o que já é bastante significativo, pois confere mais consistência ao sistema.



Pesquise mais

O vídeo disponível em <<https://www.youtube.com/watch?v=Oibb-17nD14>> (acesso em: 23 nov. 2017) oferece, em pouco mais de 10 minutos, ótima visão de classes abstratas e suas relações com herança. Nossa sugestão é que você também assista à aula anterior a esta.

Ainda que nos itens anteriores já tenhamos abordado em boa profundidade a relação entre classes abstratas e herança, vale ainda uma observação: para fins de correta utilização do recurso de classes abstratas é necessário reforçar que elas são usadas para que a modelagem de classes seja feita com base no mecanismo da herança.

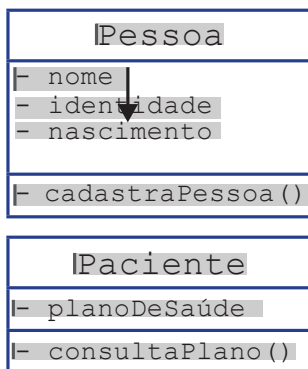
A utilização das classes abstratas deve se basear no critério de necessidade ou não de instanciação da classe. Como essa escolha depende, via de regra, da situação que se pretende implementar, a decisão ficará também por conta da sua boa avaliação da situação.

Terminamos nosso conteúdo teórico, agora veja a seguir a breve abordagem de herança múltipla.

Definição de herança múltipla

Apenas para resgatarmos os fundamentos de herança, Santos (2003) nos ensina que o mecanismo de herança funciona em apenas um sentido: da classe pai para a classe filha. Considere a relação expressa na Figura 3.1:

Figura 3.1 | Relação entre as classes Pessoa e Paciente



Fonte: adaptado de Santos (2003).

Da observação desta relação extraímos que a classe Paciente tem acesso aos campos da classe Pessoa, mas a classe Pessoa não tem acesso aos campos e métodos únicos de Paciente, como consultaPlano(). Da mesma forma, não se pode estabelecer relações entre duas classes estendidas que herdam de uma única classe pai, exceto campos em comum que o recurso da herança oferece. Enfim, o mecanismo da herança é bastante útil, porém restrito.

Esta explicação pavimenta o caminho para o que queremos definir. Herança múltipla é o mecanismo que permite a uma classe herdar métodos e campos de duas classes simultaneamente. Interessante, não acha? Pois é, mas o Java não implementa herança múltipla.

Por meio do uso de interfaces poderemos simular, de modo simples, a herança múltipla em Java.

Curioso para saber como? Em nosso próximo encontro trataremos de interfaces em detalhes e você não perde por esperar. Por ora, mantenha-se focado neste conteúdo e capriche na resolução dos exercícios propostos.

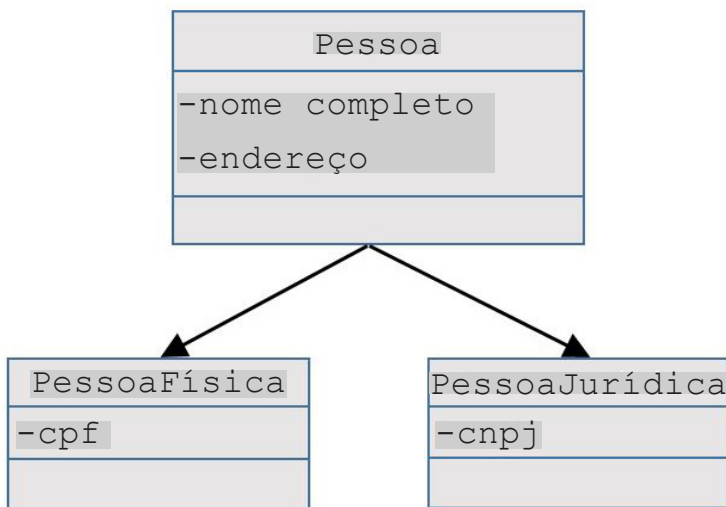
Bom estudo!

Sem medo de errar

A apresentação da situação-problema desta seção relatava a necessidade de se criar uma (ou mais, dependendo da solução adotada) classe abstrata que organizasse hierarquicamente as classes que representam pessoa física e pessoa jurídica.

Se você compreendeu o mecanismo de hierarquia (abordado detalhadamente em seção anterior) e a justificativa da existência de uma classe abstrata, então a solução desta situação será bastante simples.

Deixamos para você a missão de estruturar o código-fonte, mas sugerimos o diagrama que segue como base para a sua solução.



Como se pode observar, o núcleo da solução consiste em criar a classe abstrata **Pessoa**, que agrupa dados e comportamentos comuns entre **PessoaFísica** e **PessoaJurídica**. Devem ser acrescentados muitos outros dados e tantos outros comportamentos (métodos) nas três classes. Para que a correta distribuição de dados e comportamentos seja feita, cabe a você identificar quais dados podem ser comuns e quais são específicos.

Um sistema para todos os carros

Descrição da situação-problema

A TotaMotors é uma montadora de automóveis que acabou de se instalar no Brasil e te contratou como um dos desenvolvedores locais em Java. Logo em sua chegada a empresa pretende fabricar todos os seus modelos no país e, para que tudo seja nacionalizado, quer construir um sistema próprio para o Brasil.

A primeira missão que você recebeu foi a de estruturar uma solução para controlar dados de cor e tipo de motor que os produtos da TotaMotors terão. Apesar do pouco tempo na empresa, você sabe que os modelos produzidos são muitos e as opções de motorização e de cor variam bastante, mesmo entre modelos iguais.

Para superar este desafio você deve usar recurso da orientação a objetos. A tarefa consiste em criar classes e métodos genéricos o suficiente para acomodar todas as características dos produtos da empresa.

Mãos à obra!

Resolução da situação-problema

Caso a descrição da situação-problema tivesse informado que a quantidade de modelos é pequena (e que não tem potencial de crescimento), uma solução simples e convencional poderia ser adotada. No entanto, a contextualização é em sentido contrário: muitos modelos e muitas variações entre eles. Neste caso, a utilização de classes abstratas é solução viável e indicada.

A solução passa pela declaração de uma classe abstrata (classe Modelo, por exemplo) e de métodos também abstratos que simulem as ações relacionadas a motor e cor. Observe este esboço:

```
abstract class Modelo {  
    abstract void cor();  
    abstract void motor();  
}
```

Embora faltem os dados genéricos dos métodos e da classe, é

a partir desta estrutura que derivarão métodos e classes específicas de cada modelo. O esboço do modelo Tota1 é descrito abaixo.

```
class Tota1 extends Modelo {  
    void cor() {  
        //código das ações referentes a cor aqui.  
        //modo de pintura, modo de preparação da tinta  
etc.  
    }  
    void motor() {  
        //código das ações referentes ao motor aqui.  
    }  
}
```

Desta forma, as implementações concretas de todos os modelos poderão ser feitas da forma que foi feita para o modelo Tota1, respeitada grandes variações entre eles.

Faça valer a pena

1. Se uma classe contém um método declarado como abstrato, as classes que herdarem desta classe deverão obrigatoriamente implementar o método abstrato com o nome, modificador, tipo de retorno e argumentos declarados na classe ancestral, ou classe pai. (SANTOS, 2003).

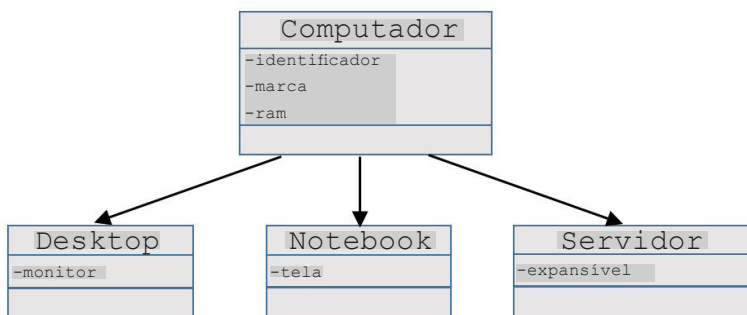
As classes abstratas caracterizam-se por:

- a) Obrigarem a implementação e instanciação da própria classe filha em seu corpo.
- b) Obrigarem a instanciação de métodos abstratos que forem nelas declarados.
- c) Permitirem a instanciação de outras classes abstratas que forem nelas declaradas.
- d) Permitirem que um método não seja implementado na própria classe, mas na classe filha.
- e) Permitirem a efetivação do mecanismo da herança na orientação a objetos.

2. O mecanismo de criação de superclasses com declarações, mas sem definições de métodos, permite a criação de métodos declarados como

abstratos. Métodos abstratos são somente declarados (com seu nome, modificadores, tipo de retorno e lista de argumentos), não tendo um corpo que contenha os comandos da linguagem que este método deva executar (SANTOS, 2003, p. 148).

Considere as relações expressas no diagrama que segue:



- I. As classes Desktop, Notebook e Servidor relacionam-se entre si diretamente, por serem todas herdeiras de Computador.
- II. O diagrama expressa o recurso da herança múltipla, já que apresenta relacionamento entre várias classes.
- III. A classe Computador não poderia ser abstrata, já que é nela que os atributos comuns são declarados.
- IV. A classe Servidor terá acesso direto ou indireto aos campos da classe Computador.

Assinale a alternativa que contém as indicações corretas das afirmações verdadeiras.

- a) Apenas a afirmação IV é verdadeira.
- b) Apenas as afirmações III e IV são verdadeiras.
- c) Apenas as afirmações I e IV são verdadeiras.
- d) As afirmações I, II, III e IV são verdadeiras.
- e) Apenas as afirmações II e III são verdadeiras.

3. Um método pode ser declarado como abstrato para que tenha que ser sobreposto em subclasses.

Um método abstrato tem que ser sobreposto e ter um corpo em uma subclasse concreta, o que não é o caso se um método comum com um corpo de método vazio for declarado no lugar do método abstrato (RUSSEL; ROBERTS, 2009, p. 614.)

Com base no conteúdo relacionado a classes e métodos abstratos, assinale a alternativa que contém os termos que completam corretamente as lacunas do texto que segue.

Uma classe declarada como _____ pode incluir qualquer declaração de variável e método, mas não pode ser usada em uma palavra reservada new. Métodos abstratos não incluem sua _____, apenas sua _____.

- a) herdada, execução, declaração.
- b) abstrata, implementação, declaração.
- c) abstrata, declaração, implementação.
- d) concreta, identificação, declaração.
- e) abstrata, execução, implementação.

Seção 3.3

Definição e uso de interfaces

Diálogo aberto

O contexto de aprendizagem que baseou a criação de situações-problemas nesta unidade remete à necessidade de se aplicar melhoramentos em funções específicas de sistemas confiados à uma empresa. Você, na condição de executor destes melhoramentos, deve propor a aplicação de recursos próprios da orientação a objetos para maximizar tais melhoramentos.

Especificamente nesta seção você deve implementar uma estrutura de interface(s), tomando como base alguma função do sistema original, escrito em linguagem procedural. A resolução desta situação visa meramente à construção de uma classe que defina uma coleção de métodos, sem, no entanto, implementá-los. Como de costume, a resolução da situação se dará pela entrega do código-fonte da aplicação, o que será mais bem detalhado em sua resolução.

Para que a superação deste desafio seja possível o conteúdo de interfaces será abordado ao longo da seção, dividido em:

- Introdução a interfaces para programação orientada a objetos.
- Declaração de interfaces para programação orientada a objetos.
- Extensão de interfaces para programação orientada a objetos.
- Implementação de interfaces para programação orientada a objetos.

Além de habilitá-lo para a situação-problema, este conteúdo também deverá torná-lo capaz de identificar situações em que a resolução do problema demanda a aplicação de interfaces.

Mãos à obra!

Não pode faltar

Seja bem-vindo à última seção da terceira unidade!

Parece que a orientação a objetos não para de colocar ótimos recursos à nossa disposição, não é mesmo? A flexibilidade da herança e do polimorfismo e a praticidade dos métodos construtores e das classes abstratas são alguns exemplos do poder deste paradigma. No entanto, há mais um mecanismo interessante por aí a ser explorado: que tal se pudéssemos criar estruturas de classes que permitissem a implementação de herança múltipla?

Em nosso último encontro discutimos que o Java não permite herança múltipla de forma direta, mas que com o uso das **interfaces** isso seria indiretamente possível. Pois bem, aqui estamos prestes a conhecer as interfaces. Sigamos adiante!

Introdução a interfaces para programação orientada a objetos

Antes de abordarmos especificamente as interfaces, vale a pena lembrarmos um pouco das classes abstratas. As classes abstratas, segundo Santos (2003), podem conter métodos não abstratos que serão herdados e poderão ser usados por instâncias das classes herdeiras. Por padrão, não se pode criar instâncias delas e são obrigatoriamente declaradas com o modificador de acesso `abstract`.

Se a classe abstrata não possuir nenhum método concreto (não abstrato), então podemos declará-la como uma **interface**. Uma interface é como uma classe, mas contém apenas declarações vazias de seus métodos. O projetista de uma interface declara os métodos que devem ser oferecidos pelas classes que implementam a interface e declara o que esses métodos devem fazer (ARNOLD; GOSLING; HOLMES, 2007, p. 52).



Assimile

Costuma-se dizer que uma interface permite estabelecer um "contrato" entre as classes; funciona de maneira bastante similar as classes abstratas, porém não permite implementação de nenhum método, contendo apenas a especificação deste (FURGERI, 2013).

Bem, até o momento não nos foi apresentada, de forma direta e inequívoca, a diferença entre **interfaces** e **classes abstratas** que justificassem a existência da primeira. Tudo ainda parece muito igual, não é mesmo? Pois bem, a diferença essencial entre classes abstratas e interfaces é que uma classe herdeira somente pode herdar de uma única classe (independentemente de ser abstrata ou não), enquanto qualquer classe pode implementar várias interfaces simultaneamente. Ou seja, temos com a interface a possibilidade de implementarmos herança múltipla em Java (SANTOS, 2003).

Declaração de interfaces para programação orientada a objetos

A declaração da interface, assim como da classe abstrata, também é muito simples. Observe o exemplo.

```
public interface Lookup {  
    Object procura (String nome);  
}
```

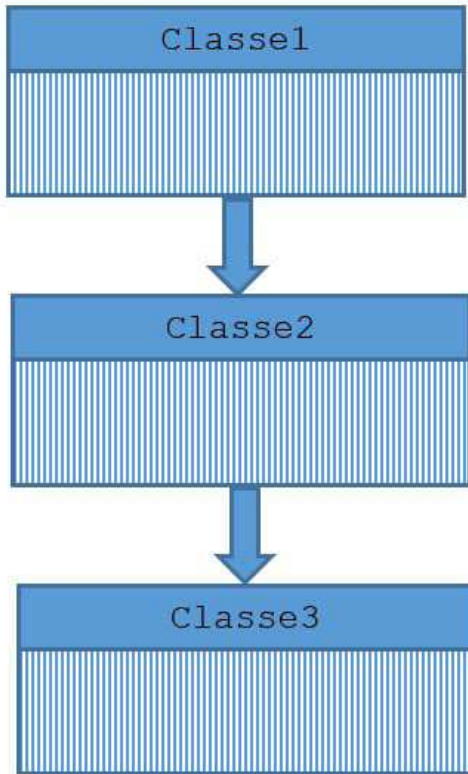
Observe a existência da palavra reservada `interface` onde estamos acostumados a encontrar `class`. No entanto, tanto a gravação do arquivo que contém o código-fonte como a compilação acontecem como se estivéssemos lidando com uma classe.

Além do que se refere à própria declaração de uma interface, este exemplo será aproveitado para mais duas considerações:

- Nenhuma implementação foi atribuída ao método `procura`, ou seja, apenas a declaração do método foi incluída na interface. Apenas a classe que de fato implementar a interface deverá providenciar codificação específica para o método.
- As classes que não estendem explicitamente nenhuma outra classe implicitamente estendem a classe `Object`. Para que este conceito fique mais claro vamos recorrer ao que ensina Deitel e Deitel, 2010: a **superclasse direta** é aquela a partir da qual a subclasse herda explicitamente. Ou seja, a superclasse direta efetiva o mecanismo da herança do jeito que o desenvolvemos até o momento. A superclasse indireta é qualquer superclasse acima da classe direta na hierarquia de classes.



Observe, neste diagrama simples, o conceito de superclasse direta e superclasse indireta.



A Classe1 é superclasse direta da Classe2.

A Classe1 é superclasse indireta da Classe3.

A Classe2 é superclasse direta da Classe3.

A Classe2 é subclasse direta da Classe1.

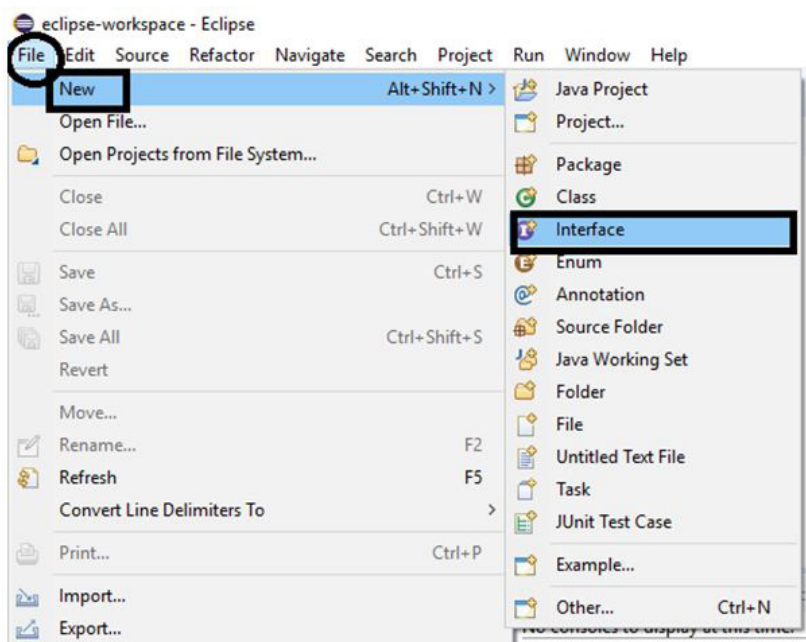
A Classe3 é subclasse direta da Classe2.

A Classe3 é subclasse indireta da Classe1.

No Java, a hierarquia de classe se inicia com a classe Object. É esta classe Object que toda classe em Java, direta ou indiretamente, estende ou “herda de”.

Outra informação importante neste contexto é a criação de uma interface num IDE (*Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado). Como temos estimulado o uso do IDE Eclipse para criação das aplicações, cabe aqui a demonstração de como se cria uma interface nesta ferramenta. Observe a Figura 3.2:

Figura 3.2 | Criação de nova interface no IDE Eclipse



Fonte: elaborada pelo autor.

Mas, afinal, se uma interface se assemelha a uma classe em sua estrutura, quais membros ela pode conter? Uma interface pode declarar constantes (campos), métodos e, por fim, classes e interfaces aninhadas. Como todos os membros de uma interface são públicos por padrão, o modificador de acesso public é omitido.

Feita esta contextualização, nosso caminho nos leva agora até a extensão e implementação das interfaces.

Extensão de interfaces para programação orientada a objetos

As interfaces podem ser estendidas por meio do uso da palavra reservada `extends`. Antes de desenvolvermos nosso exemplo, vale destacar que a extensão de várias entidades só é possível no caso de interfaces. O mecanismo primário de herança só permite que se estenda uma única classe.

Para entendermos este mecanismo, tomemos como exemplo duas interfaces: `Quadrado` e `Retangulo`. Prestemos atenção nesta relação:

```
public interface QuadradoRetangulo extends Quadrado,
Retangulo {
}
```

A interface `QuadradoRetangulo` estende `Quadrado` e `Retangulo` ao mesmo tempo. Como efeito imediato, todos os métodos e constantes definidos nas interfaces `Quadrado` e `Retangulo` passam a fazer parte do contrato de `QuadradoRetangulo`. Arnold, Gosling e Holmes (2007) nos ensinam que as interfaces que são estendidas são chamadas *superinterfaces* da nova interface. A nova interface torna-se uma *subinterface* de suas *superinterfaces*.



Pesquise mais

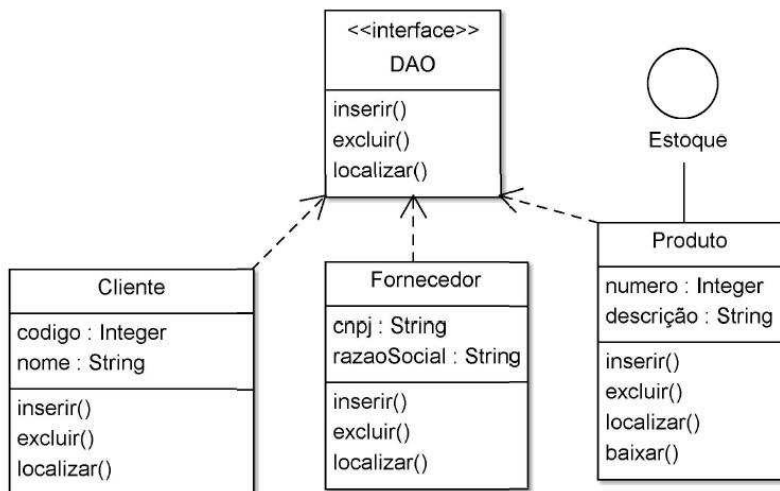
Os pacotes padrões da linguagem Java implementam várias interfaces, dentre elas `Runnable` e `Serializable`. A descoberta de suas utilidades certamente enriquecerá seu conhecimento em Java. Comece pelo artigo disponível em <https://imasters.com.br/artigo/17576/java/entendendo-serializacao-em-java?trace=1519021197&source=single>. Acesso em: 6 dez. 2017.

Com base naquilo que já conhecemos sobre herança, é possível afirmar que temos aí configurada a herança múltipla?

Implementação de interfaces para programação orientada a objetos

A efetiva utilização de uma interface será demonstrada a partir do exemplo desenvolvido por Furgeri (2013). A Figura 3.3 apresenta diagrama da UML para o relacionamento da interface `DAO` com as classes `Cliente`, `Fornecedor` e `Produto`.

Figura 3.3| Representação de interface e de classes que se relacionam com ela



Fonte: adaptado de Furgeri (2013, p. 133).

As classes Cliente e Fornecedor implementam os métodos que foram apenas declarados na interface DAO. É necessário registrar que nada impede que essas mesmas classes implementem outros métodos que não foram declarados na interface.

A classe Produto também implementa a interface DAO por meio dos métodos incluir, excluir e localizar. A interface Estoque implementa o método baixar. Não estranhe o círculo Estoque no diagrama. Ele também representa uma interface.

De modo resumido, a interface DAO pode ser assim codificada:

```

public interface DAO {
    public abstract void inserir();
    public abstract void excluir();
    public abstract void localizar();
}

```

A interface Estoque pode assim ser representada em código:

```

public interface Estoque {
    public abstract void baixar();
}

```

Atenção especial deve ser dada à classe Cliente. Primeiro vejamos o código:

```
public class Cliente implements DAO {  
    private int codigo;  
    private String nome;  
    public void inserir() {  
        // Aqui vai o código do método inserir().  
    }  
    public void excluir() {  
        // Aqui vai o código do método excluir().  
    }  
    public void localizar() {  
        // Aqui vai o código do método localizar().  
    }  
}
```

Observe a utilização da palavra reservada `implements` colocada na primeira linha do código. É a forma usada pelo Java para vincular uma classe a uma interface. Uma classe pode implementar uma ou mais interfaces através do uso do `implements`. Essa vinculação assemelha-se a um contrato. Furgeri (2013) faz uma associação interessante: eu, classe Cliente, aceito implementar todos os métodos definidos na interface DAO.



Reflita

O que aconteceria se a classe Cliente deixasse de cumprir esse contrato? Em outras palavras, o que aconteceria se a classe não implementasse todos os métodos da interface? Ela se tornaria inválida e não seria compilada.

Neste ponto há uma questão a ser feita: o que se ganha ao se estruturar as classes dessa forma? A expressão-chave da resposta é padronização do código. O estabelecimento de contrato por meio de interfaces obriga que tanto as classes Cliente e Fornecedor

(esta segue o mesmo princípio da primeira, por isso seu código foi omitido) implementem os mesmos métodos, inclusive com os mesmos nomes. Acha isso pouco? Então imagine se uma das classes definisse o método `excluir`, outra classe definisse `exclui` e uma terceira chamasse o mesmo método de `excluiReg`. Adeus facilidade de manutenção e adeus padronização...

Furgeri (2013) finaliza sua explicação ressaltando que uma interface ajuda a garantir que uma classe vai disponibilizar determinados serviços para outras classes. O autor ainda propõe o seguinte exemplo:

Considere A e B duas classes e X uma interface. A classe A implementa serviços definidos na interface X e B utiliza esses serviços de A. Caso uma nova classe C, que também implementa serviços definidos na interface X, melhore os serviços de A e seja colocada no lugar desta, a classe B não deve notar a diferença. Isso reforça ainda mais a ideia de que um sistema pode ficar mais flexível quando padrões são seguidos. (FURGERI, 2013, p. 135)



Difícil imaginar como foi possível fazer bons programas sem os recursos da orientação a objetos, não é mesmo? Esperamos que a apresentação do conteúdo teórico desta seção o tenha estimulado a praticar bastante a construção de programas com interfaces.

Continue firme, leia bastante e busque conhecimento em várias fontes. Certamente você logo se destacará em Java.

Sem medo de errar

A situação que se apresenta nesta seção demanda a aplicação de melhoramentos em uma função qualquer de um sistema, que devem ser implementados com a utilização de recursos da orientação a objeto, principalmente as interfaces.

Da maneira que nos foi apresentada, a situação não especifica qual função deve ser modificada. Também não sabemos quais dados e comportamentos têm essa função. Se por um lado essa forma de apresentação do problema não nos permite planejar de imediato uma solução (lembre-se, não sabemos em qual função

aplicar as melhorias), por outro nos obriga a imaginar uma situação em que a aplicação da interface se faz necessária. É esse, aliás, o principal objetivo desta seção: torná-lo apto a identificar situações em que as interfaces são a melhor e mais viável solução.

Uma utilização comum para a interface se dá em projetos que, por serem normalmente extensos e com muitas variações de implementações, devem seguir um padrão estabelecido na construção de seus métodos. Esta característica de aplicação permite que nossa sugestão para a resolução do problema remeta a uma interface que declare métodos comuns de operações de acesso a dados, tais como salvar, deletar e atualizar. Como ainda não sabemos utilizar os recursos de acesso a dados no Java, a resolução não estará completa, caso essa sugestão seja codificada.

```
public interface OperacoesBasicas {  
    public void salvar();  
    public void deletar();  
    public void atualizar();  
}
```

As operações que envolvem, por exemplo, um produto, devem ser implementadas segundo o padrão da interface OperacoesBasicas, como segue:

```
public class Produto implements OperacoesBasicas {  
    @Override  
    public void salvar() {  
        // TODO implementação do método salvar,  
        conforme dados específicos do produto.  
    }  
  
    @Override  
    public void deletar() {
```



```

        // TODO implementação do método deletar,
        conforme dados específicos do produto.
    }

    @Override
    public void atualizar() {
        // TODO implementação do método atualizar,
        conforme dados específicos do produto.
    }
}

```

Crie outras situações em que métodos particulares devem sobrepor outros métodos já criados e entregue o código gerado ao professor. Bom trabalho!

Avançando na prática

Conflitos de herança

Descrição da situação-problema

Você foi incumbido de desenvolver determinada aplicação para um cliente. Durante o desenvolvimento você avaliou que a aplicação do conceito de herança múltipla seria a melhor solução para o problema que se apresentava.

No entanto, durante as revisões feitas no programa você percebeu um problema com herança múltipla: uma determinada classe implementava duas interfaces e essas interfaces declaravam campos. A classe que implementa os métodos não passava pelo processo de compilação. Como superar esse problema?

Resolução da situação-problema

A declaração de campos nas interfaces é permitida e constitui recurso bastante utilizado. No entanto, o problema se manifesta se as interfaces declararem campos com os mesmos nomes. Assim, para que a classe que implementa os métodos possa ser compilada, basta reescrever o código diferenciando os nomes dos campos.

Faça valer a pena

1.



Interfaces também podem ser úteis para implementar bibliotecas de constantes: já que todos os campos de uma interface devem ser declarados como static e final, podemos escrever interfaces que somente contêm campos, e qualquer classe que implementar essa interface terá acesso a estes campos (SANTOS, 2003, p. 155).

Assinale a alternativa que contém a correta conceituação de interface.

- a) Estrutura própria da linguagem Java que permite a aplicação do conceito de herança.
- b) Estrutura que permite que uma – e somente uma - interface determine os métodos que uma classe herdeira deve implementar.
- c) Estrutura que agrupa métodos abstratos e não abstratos, mas nenhuma interface aninhada.
- d) Estrutura que agrupa apenas métodos não abstratos, ou seja, com declarações sem implementações.
- e) Estrutura que viabiliza que classes abstratas sejam declaradas sem implementação alguma de métodos.

2.



Interfaces dão suporte direto ao conceito de compatibilidade de tipos. Um tipo tem um nome e especifica um conjunto de métodos, cada um com um nome, um conjunto de tipos, um para cada parâmetro, e um tipo de retorno. Um tipo, B, é compatível com outro, A, quando todos os métodos especificados em A estão presentes em B. B pode ter mais métodos, mas contanto que tenha os métodos especificados em A, será compatível com A (RUSSEL e ROBERTS, 2009, p. 619).

Em relação às interfaces e suas relações com classes abstratas, analise as afirmações que seguem.

- I. Assim como as classes abstratas, as interfaces não podem ser instanciadas.

- II. Ao contrário das classes abstratas, as interfaces podem estender mais de uma interface.
- III. Assim como as classes abstratas, as interfaces podem estender mais de uma classe concreta.

Assinale a alternativa que contém apenas indicações de afirmações verdadeiras.

- a) Apenas as afirmações I e III são verdadeiras.
- b) Apenas as afirmações II e III são verdadeiras.
- c) Apenas as afirmações I e II são verdadeiras.
- d) As afirmações I, II e III são verdadeiras.
- e) Apenas as afirmações I e III são verdadeiras.

3.

As bibliotecas de classes Java fazem vasto uso de interfaces: elas são usadas para especificar tipos “conceituais”, que determinam o conjunto de métodos que uma classe deve fornecer para seus objetos serem usados. Um exemplo é o da interface `Iterator` que especifica um tipo e um conjunto de métodos para iteração em uma estrutura de dados. (RUSSEL e ROBERTS, 2009, p. 620).



Considerando o contexto da implementação das interfaces, avalie as seguintes asserções e as relações propostas entre elas.

I. Ao implementar métodos da interface, a classe que herda esses métodos não pode conter outros métodos além daqueles que devem ser implementados.

PORQUE

II. Se a classe herdeira não implementar todos os métodos da interface ela se torna inválida e não é compilada.

Assinale a alternativa verdadeira.

- a) As asserções I e II são proposições falsas.
- b) A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.
- c) As proposições I e II são verdadeiras, e a II justifica a I.
- d) As proposições I e II são verdadeiras, mas a II não justifica a I.
- e) A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.

Referências

_____. **Apostila Java e orientação a objetos**. Classes abstratas. Disponível em: <<https://www.caelum.com.br/apostila-java-orientacao-objetos/classes-abstratas/>>. Acesso em: 22 nov. 2017.

ARNOLD, K., GOSLING, J., HOLMES, D. **A Linguagem de Programação Java**. 4. ed. Porto Alegre: Bookman, 2007.

CAELUM. **Interfaces**. Disponível em <<https://www.caelum.com.br/apostila-java-orientacao-objetos/interfaces/>>. Acesso em: 6 dez. 2017.

DEITEL, H. M., DEITEL, P. J. **Java – Como Programar**. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

FERNANDO, R. **Tratando exceções em Java**. Disponível em: <<https://www.devmedia.com.br/tratando-excecoes-em-java/25514>>. Acesso em: 16 nov. 2017.

FERREIRA, Aurélio Buarque de Holanda. **Dicionário Eletrônico Aurélio Século XXI**. Rio de Janeiro: Editora Nova Fronteira, 1999.

FURGERI, S. **Java 7 Ensino Didático**. 2. ed. São Paulo: Érika, 2013.

GRONER, L. **Curso de Java 44: orientação a objetos: Interfaces**. YouTube, 17 fev. 2016. Disponível em: <<https://www.youtube.com/watch?v=6uLLfRNnRA4>>. Acesso em: 6 dez. 2017.

RUSSEL, W; ROBERTS, G. **Desenvolvendo software em Java**. 3. ed. Rio de Janeiro: LTC, 2009.

SABINO, V. **Asserções**. Disponível em: <<http://www.linhadecodigo.com.br/artigo/86/assercoes.aspx>>. Acesso em: 20 nov. 2017.

SANTOS, R. **Introdução à programação orientada a objetos usando Java**. Rio de Janeiro: Campus, 2003.

SEVERO, C. E. P. **Programação orientada a objetos (Parte 7): herança múltipla, o conceito de interface**. YouTube, 11 jul. 2014. Disponível em: <https://www.youtube.com/watch?v=Cc_9eqZH9dY>. Acesso em: 6 dez. 2017.

Aplicações orientadas a objetos

Convite ao estudo

Bem-vindo à quarta unidade do curso!

Muitos dos recursos interessantes e úteis que só a orientação a objetos oferece foram abordados nas três primeiras unidades deste material. Foi lá que apresentamos a você classes, métodos, atributos, herança, polimorfismo e outras coisas legais que, em dado momento, passaram a coexistir em perfeita harmonia, em seu acervo de habilidades, com as estruturas de seleção e repetição, essas provavelmente já velhas conhecidas.

É provável que você queira usar os mecanismos avançados da orientação a objetos que já conhece mesmo em aplicações que não os demandem obrigatoriamente. Mas é bem provável também que os recursos de seleção e repetição estarão em todas as suas aplicações. É exatamente de outros destes “recursos universais” da programação que trataremos nesta derradeira unidade.

Ao final deste ciclo, teremos como produto uma aplicação que armazena dados da música e a quantidade de vezes que ela foi executada. Esta aplicação, aliás, será construída em três etapas e com base nas situações que serão expostas nas três seções desta unidade. Como de costume, os problemas são baseados em um contexto de aprendizagem, assim apresentado:

A empresa cresceu e, na mesma proporção, também evoluiu sua habilidade para criar aplicações em Java. Contudo, “com grandes poderes vêm grandes responsabilidades”.

Uma grande desenvolvedora de aplicações de fornecimento de música *on-line* resolveu confiar à sua equipe determinadas atividades próprias do seu ciclo de criação. Uma dessas atividades consiste na criação de funcionalidade que armazena dados da música (incluindo código, nome, gênero e autor) e a quantidade de execuções realizadas pelos usuários em dado período. Este armazenamento, que num primeiro momento será feito em um conjunto de *arrays*, deverá ser gravado em arquivos na versão final do programa.

Para que essa funcionalidade seja bem construída – e seu desafio superado – você contará, nesta unidade, com conteúdo apropriado. A Seção 1 define os *arrays* como agrupamentos de dados relacionados e armazenados em uma única estrutura. Veremos que até mesmo instâncias de classes podem compor um *array*.

A Seção 2 aborda sequências de caracteres, viabilizadas por meio da classe *String*. Com os métodos desta classe será possível comparar duas sequências de caracteres, obter a informação de se determinada sequência está vazia ou não e retornar um trecho específico de uma determinada *string*.

Por último, a Seção 3 nos apresenta coleções de objetos e arquivos. À propósito, a habilidade que você irá adquirir na manipulação de arquivos será útil na versão final do aplicativo de música que criaremos.

Pronto, agora é só iniciar a leitura do nosso conteúdo, resolver os exercícios e se preparar para ser reconhecido como um ótimo desenvolvedor Java.

Bom estudo!

Seção 4.1

Arrays em Java

Diálogo aberto

Olá! Preparado para mais uma seção de Programação orientada a objetos?

Depois de termos estudado as ferramentas que tornam o paradigma tão poderoso, voltamos nosso foco para alguns elementos comuns a outros tipos de programação. Em específico, esta seção trata dos *arrays*, estruturas capazes de armazenar dados do mesmo tipo. Veremos que se trata de um recurso bastante interessante quando se pretende agrupar dados relacionados.

A situação na qual se baseia este encontro nasce na incumbência recebida pela sua equipe de criar e armazenar uma estrutura apropriada para o registro de execuções das músicas oferecidas por aplicação on-line. O problema deverá ser resolvido em três etapas e a primeira – que nos interessa por ora – consiste em armazenar em uma estrutura de *array* os dados da música e a quantidade de execuções que ela teve.

De forma mais detalhada, a resolução deste primeiro desafio inclui a criação de aplicação que:

- Declare o vetor de **código da música** e o vetor de **quantidade de execuções**, com seus tipos apropriados para conter um código e um valor que expressa quantidade.
- Receba via teclado o código da música e a quantidade de execuções realizadas em um dia. Vale registrar que, quando as classes aqui criadas estiverem integradas à aplicação final, estes dados serão fornecidos por outro meio diferente da digitação.
- Imprima em tela o código da música e sua respectiva quantidade de execuções.

Você vai notar que, por causa da sua natureza introdutória, este desafio é de resolução simples. Ele demandará os conhecimentos

de assuntos que serão tratados nesta primeira seção, quais sejam a conceituação de *arrays*, sua utilização e *arrays* com mais de uma dimensão.

Boa leitura!

Não pode faltar

É provável que, em seu primeiro contato com uma variável, a tenham apresentado como uma posição de memória capaz de armazenar apenas um dado em uma mesma unidade de tempo. Ou, quem sabe, como um objeto escalar, apropriado para reter um valor, com um nome e um tipo associados.

Sim, havia lógica naquelas explicações e uma coisa em comum entre elas: o fato de que uma variável consegue armazenar (e oferecer de volta) um – e apenas um – dado. Se você precisar de um meio que armazene mais do que um dado, sem precisar declarar mais do que uma variável, então deverá recorrer a outro recurso. É exatamente este outro recurso – chamado *array* – que será esmiuçado nas próximas linhas.

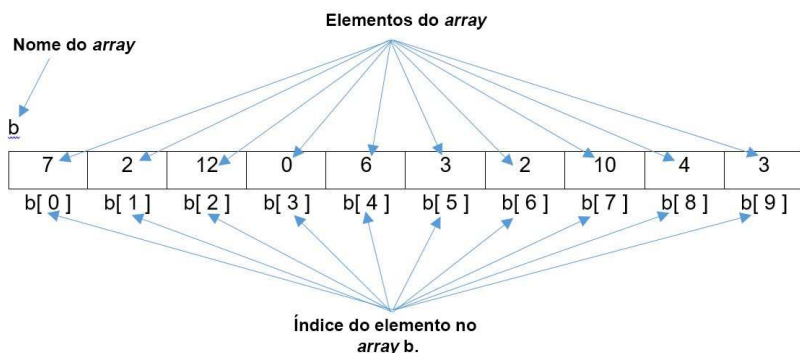
Introdução a *Arrays* para programação orientada a objetos

Antes de conceituarmos o termo, convém esclarecer que um *array* não é um comando ou método, embora ambos estejam inseridos em seu contexto de utilização. Em outras palavras, será por meio de comandos que manipularemos os *arrays* e será no corpo de algum método que ele desempenhará sua função.

Um *array* é um grupo de variáveis (chamados elementos ou componentes) que contém valores, todos do mesmo tipo. Os elementos de um *array* podem ser tipos primitivos ou tipos por referências (DEITEL; DEITEL, 2010, p.190).

Com a representação gráfica exibida na Figura 4.1, você poderá compreender o conceito e outros aspectos da estrutura.

Figura 4.1 | Array de 10 elementos



Fonte: elaborada pelo autor.

Antes de prosseguirmos, vale destacar que a denominação “vetor” é usada como sinônimo de *array*, inclusive neste texto. Sigamos.

Para efeito de ilustração do exemplo, vamos imaginar que estamos diante de um *array* que armazena a quantidade de alunos que não obtiveram nota de aprovação em cada uma das 10 salas da instituição.

Cada elemento do *array* apontado individualmente por uma das setas superiores em nossa figura representa a quantidade de reprovações em cada turma. Relacionadas a estes elementos, cabem duas observações:

- Para que um elemento faça parte do vetor, ele deve ser inserido durante a execução do programa, via digitação pelo teclado ou por comando de atribuição. Trataremos desta característica logo adiante, em um novo exemplo.
- Os elementos devem ser de um único tipo, definido na declaração do *array*.



Assimile

Não há array que contenha elementos de tipos diferentes entre si. Este fato os caracteriza como estruturas de dados homogêneas.

Se é verdade que um *array* é capaz de agrupar vários elementos em sua estrutura, então é necessário que se tenha um meio de acessar individualmente cada um desses elementos. O vetor de 10 elementos que representamos na Figura 4.1 foi chamado de *b*. No entanto, apenas dar um nome a estrutura não garante a individualização de seus elementos.

Um índice deve ser criado para que cada elemento possa ser identificado em particular. Assim, para referenciar um elemento particular na estrutura, utilizamos o nome do *array* e índice, que é exatamente o número de posição do elemento no *array*.

Ensinam Deitel e Deitel (2010, p. 190) que o primeiro elemento do *array* tem índice zero e às vezes é chamado zero-ésimo elemento.

Observe os exemplos extraídos da Figura 4.1:

- A posição indicada por *b*[2] contém o elemento 12, ou seja, o elemento 12 está no índice 2 do *array b*.
- A posição indicada por *b*[9] contém o elemento 3, ou seja, o elemento 3 está no índice 9 do *array b*.

Um índice deve ser uma variável do tipo inteiro, não negativo. Em relação aos tipos de dados suportados pelos elementos do *array*, cabe uma observação: os *arrays* são objetos e, por isso, considerados **tipos por referência**. Esses tipos são usados para armazenar as localizações de objetos na memória e os criamos assim que criamos um objeto de determinada classe e a referência àquele objeto por meio de uma variável. Por exemplo, na linha `Automovel a1 = new Automovel();`, por exemplo, *a1* é a variável de instância de tipo por referência, pois contém uma referência ao objeto `Automovel`.

Os tipos primitivos são nossos conhecidos e lembraremos aqui apenas seus nomes: *byte*, *short*, *int*, *long*, *float*, *double*, *boolean* e *char*.

A declaração dessa estrutura tem dois componentes: o tipo do *array* e o nome do *array*. Até aí, nada diferente do que já utilizamos nas declarações de variáveis de tipos primitivos. Por exemplo, a linha `int [] b` indica que foi criado um *array* que conterá elementos inteiros e que recebeu o nome *b*. Os colchetes colocados após o tipo indicam que a variável contém um *array*.



Já que os tipos declarados para os elementos dos *arrays* podem ser variados, seria possível declarar um *array* do tipo *array*?

Para que a criação da estrutura esteja completa, é necessário, ainda, criar uma instância de *array* e atribuí-la a variável que lhe dá o nome. Assim teríamos `b = new int[10]`. A variável de *array* `b` recebe a referência para um novo vetor de inteiros de 10 elementos.

No entanto, é comum declararmos e incluirmos a expressão de criação do *array* numa mesma linha, como segue:

```
int b[ ] = new int[10]
```

Quando um *array* é criado, cada elemento seu recebe o valor zero para tipos numéricos, `false` para o tipo `boolean` e `null` para referências (DEITEL; DEITEL, 2010, p. 191). No ato da criação, no entanto, o desenvolvedor pode inicializar o *array* com os valores que desejar, como no exemplo `int[] valoresIniciais = {7, 12, 58, 4, 15, 0}`. A lista de elementos separados por vírgulas é chamada lista inicializadora e o tamanho do *array* é determinado pela quantidade de elementos nesta lista.

A primeira – e mais elementar – forma de se incluir elementos em um *array* é por meio de comandos individuais de atribuição ou de leitura pelo teclado. Em relação à recuperação (impressão em tela, por exemplo) desses elementos, se a considerarmos de forma individual, também precisaremos fazer referência a cada índice do vetor, um a um. Observe a aplicação que segue.

```
public class Array1 {
    public static void main (String[] args) {
        int[] b = new int[4]; //declaração do vetor
        /* Criação dos elementos do vetor */
        /* por meio de comandos de atribuição */
        /* estas atribuições poderiam ser
substituídas */
        /* por comandos de leitura pelo teclado.
*/
    }
}
```

```

        b[0] = 7;
        b[1] = 2;
        b[2] = 12;
        b[3] = 0;

        // Impressão individual de cada elemento
do vetor

        System.out.println (b[0]);
        System.out.println (b[1]);
        System.out.println (b[2]);
        System.out.println (b[3]);
    }
}

```

Este recurso, no entanto, apresenta um problema: imagine como seria o tratamento dos elementos em um vetor de, digamos, 50 ou mais posições? Imagine, por exemplo, imprimir ou atribuir valores em muitas posições, como fizemos na sequência `b[0] = 7; b[1] = 2; b[2] = 12; b[3] = 0`. O desenvolvedor teria muito trabalho e a implementação estaria propensa a erros, para dizer o mínimo.

A segunda forma, muito mais útil e racional, se efetiva pela utilização do comando `for`. Seu mecanismo é apropriado para a leitura e recuperação dos elementos do vetor. Com essa adaptação do exemplo anterior, as coisas ficarão mais claras:

```

import java.util.Scanner;

public class Array2 {
    public static void main (String[] args) {
        int i;
        int[] b = new int[10];
        Scanner entrada = new Scanner(System.in);
        //Entrada dos elementos do vetor.
        for (i=0;i<10;i++) {
            System.out.printf("Informe o %d

```

```

elemento do array: ", i+1);
        b[i] = entrada.nextInt();
    }
    // Impressão de cada elemento do vetor.
    for (i=0;i<10;i++) {
        System.out.printf("\nElemento   %d:
%d ", i+1,b[i]);
    }
}
}

```

A execução dessa aplicação será:

```

Informe o 1 elemento do array: 5
Informe o 2 elemento do array: 6
Informe o 3 elemento do array: 7
Informe o 4 elemento do array: 2
Informe o 5 elemento do array: 3
Informe o 6 elemento do array: 4
Informe o 7 elemento do array: 9
Informe o 8 elemento do array: 5
Informe o 9 elemento do array: 9
Informe o 10 elemento do array: 1

```

```

Elemento 1: 5
Elemento 2: 6
Elemento 3: 7
Elemento 4: 2
Elemento 5: 3
Elemento 6: 4
Elemento 7: 9
Elemento 8: 5
Elemento 9: 9
Elemento 10: 1

```

O índice do *array* é implementado pela variável *i*. Por meio do comando `for`, o índice varia de 0 a 9, o que permite percorrer as dez posições da estrutura. A cada iteração, um valor de elemento é obtido pelo teclado por meio do comando de leitura. No segundo comando `for`, a impressão dos elementos é feita.

Pronto! Já temos a forma apropriada para tratarmos os elementos do *array* de modo racional. Antes de avançarmos para as primeiras utilizações dessas estruturas, cumpre tratarmos da classe `Arrays`. Para que possamos utilizar seus métodos, é necessário importar o pacote que a contém com `import java.util.Arrays`. Depois de tomada esta providência, já teremos à disposição métodos que permitem, por exemplo, busca e organização nos arrays. Vale ressaltar que, ao criarmos um array, já podemos usar os métodos da classe `Arrays` nesse vetor, sem necessidade de declararmos objetos.

Para praticar a utilização de um dos métodos da classe `Arrays`, insira o comando `Arrays.sort(b)` antes do laço `for` que imprime os elementos do vetor. Você vai observar que os valores foram exibidos em ordem crescente. Prático, não acha?



Pesquise mais

A classe `Arrays` oferece ótima variedade de métodos para manipulação de vetores, o que pode facilitar bastante o trabalho do desenvolvedor. Saiba mais sobre esses métodos e sobre a estrutura da classe consultando a documentação oficial da Oracle, disponível em: [__https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html](https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html). Acesso em 18 dez. 2017. Caso prefira a documentação escrita na Língua Portuguesa, acesse: <http://www.javaprogressivo.net/2012/09/como-usar-classe-arrays-java.html>. Acesso em 18 dez. 2017.

Na sequência dessa abordagem introdutória, trataremos de exemplos de utilização dos *arrays*.

Arrays unidimensionais para programação orientada a objetos

O título desse item remete a *arrays* unidimensionais – ou seja, com uma só dimensão – para que a devida diferenciação das estruturas com mais de uma dimensão seja feita. Um *array*

unidimensional tem representação igual à feita na Figura 4.1 e possui apenas um índice para que seja possível o acesso a seus elementos. Da forma com que foi desenhado, sua única dimensão é a largura. Por sua vez, os bidimensionais são representados por quadros com altura e largura e serão abordados adiante neste texto.

Um bom exemplo de utilização de array unidimensional é dado por Deitel e Deitel (2010) e aqui adaptado para nosso contexto. Trata-se de aplicação que faz a leitura dos oito elementos de um vetor de inteiros, soma e calcula a média dos valores lidos, exibindo soma e média antes do final da execução.

```
import java.util.Scanner;
import java.util.Arrays;
public class SomaArray {
    public static void main (String[] args) {
        Scanner entrada = new Scanner(System.in);
        //uma constante é criada para conter a
        dimensão do vetor.
        final int tamanhoVetor = 8;
        int soma = 0; //variável que conterá a soma
        dos elementos.
        float media = 0; //variável que conterá a
        média dos elementos.
        int i;
        int a[] = new int[tamanhoVetor]; //
        declaração e criação do vetor.
        for (i=0; i < a.length; i++) {
            System.out.printf("Informe o %d
            elemento do array: ", i+1);
            a[i] = entrada.nextInt();
            soma = soma + a[i];
        }
        media = soma/8;
        System.out.printf("\nA soma dos elementos
        é %d ", soma);
```

```
        System.out.printf("\nO valor médio dos  
elementos é %.2f ", media);  
    }  
}
```

A classe `java.util.Arrays` é importada para que seus atributos e métodos possam ser utilizados. No corpo do comando `for`, observe o acesso ao valor do atributo `length`. Ele retorna o tamanho (ou dimensão) do `array` que está em uso.

À propósito, o tamanho do `array`, é fixado a partir da sua criação e permanece imutável durante o programa. Um novo `array` deverá ser criado, caso a dimensão do primeiro não seja suficiente.

Arrays de instâncias de classes para programação orientada a objetos

Este item do nosso conteúdo, por sucinto que seja, servirá para demonstrar a flexibilidade de armazenamento de uma estrutura de `array`. Sabemos que um `array` suporta tipos primitivos e, no item dos arrays com mais de uma dimensão, trataremos do `array` que armazena outro `array`.

Embora estes recursos já nos deem condições para criar boa variedade de aplicações, é possível ainda contarmos com a possibilidade de armazenarmos instâncias de classes nos vetores. Para fins de simplificação do exemplo, consideraremos apenas o trecho que segue:

```
Automovel[] auto = new Automovel[4];  
auto[0] = new Automovel();  
auto[1] = new Automovel();  
auto[2] = new Automovel();  
auto[3] = new Automovel();
```

Observe que um `array` de 4 elementos foi declarado normalmente. No entanto, os elementos que compõem este `array` são instâncias da classe `Automovel`. O mesmo efeito seria

obtido com a linha que segue:

```
Automovel[] auto = new Automovel[]{new Automovel(),new  
Automovel(),new Automovel(), new Automovel()};
```

Arrays multidimensionais para programação orientada a objetos

Embora o nome deste item granular sugira a abordagem de *arrays* multidimensionais, é dos *arrays* bidimensionais que trataremos aqui. Em outras palavras, nossos vetores terão apenas duas dimensões.

Furgeri (2013) ensina que um *array* bidimensional possui dois índices e possibilita que os valores sejam armazenados na forma de matriz. Embora o Java não suporte *arrays* bidimensionais, é possível obter a mesma funcionalidade criando um *array* de *arrays*.

O exemplo que segue, extraído e adaptado de Furgeri (2013), demonstra esse mecanismo para armazenamento e soma de seis valores inteiros.



Exemplificando

Esta aplicação demonstra o modo como *arrays* são declarados e utilizados. O primeiro vetor representa a quantidade de linhas da matriz e o segundo representa as colunas. Excepcionalmente, os elementos foram inseridos por meio de atribuição, conforme comentado no código.

A aplicação contém dois laços de repetição com a instrução *for*. O primeiro serve para que o algoritmo percorra as linhas e o outro serve para que as colunas da matriz sejam percorridas.

```
public class ArrayBi {  
  
    public static void main (String[] args) {  
  
        int matriz [][] = new int [2][3];  
  
        int soma = 0;  
  
        int linha, coluna;
```

```
// inicialização da matriz por atribuição.

matriz [0][0] = 1; matriz [0][1] = 2; matriz [0][2] = 3;
matriz [1][0] = 4; matriz [1][1] = 5; matriz [1][2] = 6;

for (linha = 0; linha < 2; linha++)

    for (coluna = 0; coluna < 3; coluna++) {

        soma = soma + matriz [linha][coluna];

    }

System.out.printf("\nA soma dos elementos é %d ", soma);

}
```

A saída de execução do programa é composta pela mensagem "A soma dos elementos é 21". Trata-se de aplicação bastante simples, mas capaz de demonstrar o uso de um *array* bidimensional.

Agora que você já tem boa base teórica sobre arrays, siga firme nas leituras e na resolução dos exercícios. Bom trabalho!

Sem medo de errar

Uma grande fornecedora de música on-line contratou a sua equipe para implementar a funcionalidade de armazenamento de dados das músicas executadas por seus assinantes. O problema será dividido em três etapas e a implementação se dará paulatinamente.

Estamos diante de uma excelente oportunidade para a utilização de *arrays*, como a própria descrição da situação sugere. A resolução, por mais simples que seja, ainda assim requer habilidade em dimensionar e declarar vetores, bem como fazer a leitura e impressão de seus elementos.

Inicie a solução planejando como serão os dois vetores que serão usados para armazenar o código da música e a quantidade de execuções. A escolha do nome e do tamanho dos vetores fica por sua conta, mas uma sugestão pode ser bem-vinda:

```
final int TamArrayExecucoes = 20;
final int TamArrayCodigo = 20;
int ArrayExecucoes[] = new int[TamArrayExecucoes];
int ArrayCodigo[] = new int[TamArrayCodigo];
```

Passada essa fase, você escreverá os comandos da aplicação, cuidando para que os elementos dos vetores sejam lidos em laço(s) de repetição. As mensagens dirigidas ao usuário para entrada de dados devem ser claras e objetivas.

Ato contínuo à digitação, a aplicação deve exibir os dados digitados. A seu critério, você poderá organizar a saída em ordem crescente ou decrescente de execuções, por exemplo.

Com a aplicação executada e testada, você deve entregar o código-fonte para completar a tarefa.

Lembre-se que esta é a primeira fase da resolução e, nas seguintes, outros recursos e habilidades serão necessárias.

Avançando na prática

Otimizando o armazenamento dos dados

Descrição da situação-problema

Em seu local de trabalho, há uma considerável variedade de desenvolvedores. Alguns novatos, outros mais antigos; uns estudiosos, outros nem tanto. Enfim, há uma variedade de perfis que podem até mesmo ser identificados pela utilização (ou não) de certos recursos do Java.

Chamado a ajudar um colega na otimização de uma certa aplicação, você observou o uso excessivo de variáveis de um mesmo tipo para armazenar valores associados. No caso particular, as variáveis em grande número foram criadas para armazenar temporariamente as idades dos 20 clientes mais representativos da empresa contratante.

Sua missão é oferecer uma solução mais racional para o armazenamento temporário destes dados. Bom trabalho!

Resolução da situação-problema

Considerando que os dados são do mesmo tipo e que a quantidade deles (20) excede o razoável para armazenamento em itens escalares, a solução está na criação de um *array* de inteiros, de dimensão 20.

Ao invés da leitura ou da atribuição de valores às idades ser feita individualmente para cada variável, uma estrutura única e apta a receber vários elementos será usada. Essa estrutura, naturalmente, é um *array*.

Uma solução simples e viável para o caso é a que segue:

```
import java.util.Scanner;

public class AvancandoNaPratica41 {
    public static void main (String[] args) {
        Scanner entrada = new Scanner(System.in);
        final int tamanhoVetor = 20;
        int idade[] = new int[tamanhoVetor];
        int i;
        for (i=0; i < idade.length; i++) {
            System.out.printf("Informe a %d idade:
", i+1);

            idade[i] = entrada.nextInt();
        }
        System.out.println("\nAs      idades      foram
armazenadas com sucesso.");
    }
}
```

Faça valer a pena

1. Suponha que seja necessário armazenar e manipular dezenas de nomes de pessoas num programa de computador, seriam necessárias dezenas de variáveis, cada uma armazenando um nome diferente. Em vez disso, é possível a declaração de apenas uma variável indexada, chamada *array* (FURGERI, 2013).

Em relação ao meio utilizado para incluir valores em cada uma das posições de um *array*, analise as afirmações que seguem e assinale a alternativa que contém apenas indicações de afirmações verdadeiras.

- I. A forma mais eficiente de inclusão de valores nas posições de um *array* é por meio da leitura dos elementos a cada iteração, principalmente em arrays de grande dimensão.
- II. Um *array* aumentará automaticamente sua dimensão em tempo de execução a cada novo elemento inserido além da sua capacidade.
- III. Embora os *arrays* sejam estruturas homogêneas, o desenvolvedor pode optar por incluir tipos distintos nos elementos, desde que uma exceção seja prevista.

- a) Apenas a afirmação II é verdadeira.
- b) Apenas as afirmações I e II são verdadeiras.
- c) Apenas a afirmação I é verdadeira.
- d) As afirmações I, II e III são verdadeiras.
- e) Apenas as afirmações II e III são verdadeiras.

2. Para referenciar um elemento particular em um *array*, especificamos o nome da referência para o *array* e o número da posição do elemento no *array*. O número da posição do elemento é chamado índice ou subscrito do elemento (DEITEL; DEITEL, 2010).

Considere a aplicação que segue:

```
import java.util.Arrays;
import java.util.Scanner;
public class q2 {
    public static void main (String[] args) {
        Scanner e = new Scanner (System.in);
        int a [][] = new int [3][4];
        int m, n;
```

```

        //System.out.printf("Tamanho: %d", a.length);
        for (m=0; m<a.length; m++)
            for (n=0; n<a.length+1; n++) {
                a[m][n] = e.nextInt();
            }
        for (m=0; m<a.length; m++)
            for (n=0; n<a.length+1; n++) {
                System.out.println(a[m][n]);
            }
    }
}

```

Assinale a alternativa que expressa sua funcionalidade.

- a) Leitura e impressão de um vetor genérico com 3 colunas e 4 linhas.
- b) Leitura e impressão de dois vetores de inteiros, ambos com 3 colunas e 4 linhas.
- c) Leitura e impressão de dois vetores de inteiros, ambos com 3 linhas e 4 colunas.
- d) Leitura e impressão de um vetor genérico com 3 linhas e 4 colunas.
- e) Leitura e impressão de dois vetores de inteiros, um de tamanho 3 e outro de tamanho 4.

3. Os objetos de *array* ocupam espaço na memória. Como os outros objetos, os *arrays* são criados com a palavra-chave *new*. Para criar um objeto de *array*, especifique o tipo dos elementos do *array* e o número de elementos como parte de uma expressão de criação do *array* que utiliza a palavra-chave *new* (DEITEL; DEITEL, 2010, p. 191).

Em relação à classe *Arrays* e a sua utilização, assinale a alternativa correta.

- a) Trata-se da classe que viabiliza a utilização de *arrays* em uma aplicação.
- b) Ela contém métodos e atributos que oferecem funcionalidades para manipulação dos *arrays*.
- c) Dispensa importação na aplicação, já que os objetos já foram pré-instanciados.
- d) Por meio desta classe é possível declarar os *arrays* da aplicação.
- e) Esta classe permite alocação de memória dos objetos do *array* durante sua declaração.

Seção 4.2

Strings em Java

Diálogo aberto

Quando tratamos de *arrays* em nosso último encontro, ficou claro que estávamos diante da estrutura apropriada para resolvermos a primeira etapa da situação que se apresentava. Apenas para nossa lembrança ficar mais clara, era sua missão usar dois vetores para guardar valores numéricos de código da música e da quantidade de execuções, ambos recebidos pelo teclado.

Até agora, tudo que conseguimos manipular em nossas aplicações foram valores numéricos, o que, sem dúvida, impôs alguma restrição durante o desenvolvimento.

Chegou a hora, no entanto, de darmos um passo à frente. Nesta seção serão abordadas as cadeias de caracteres, particularmente chamadas de *strings*. Por meio dos métodos da classe *String*, conseguiremos manipular caracteres e resolver, com eficiência, a situação que nos é colocada.

Depois de construirmos parte da funcionalidade que armazena dados das músicas tocadas em um certo aplicativo, avançamos agora para a inclusão do **nome**, do **gênero** e do **autor** da música. Se na primeira etapa os *arrays* nos serviram bem, agora precisaremos de estrutura que comporte o armazenamento e manipulação de cadeias de caracteres. E, claro, a classe *String* nos fornecerá esse suporte.

Para resolver essa situação, você deverá acrescentar à aplicação construída na Seção 4.1 uma funcionalidade que faça a leitura e o armazenamento do nome, o gênero e o autor da música. Entretanto, antes de armazenar esses dados, eles deverão passar pelos seguintes processamentos:

- O primeiro caractere de cada nome do autor deve ser alterado para maiúsculo, caso tenha sido digitado em minúsculo.
- O nome da música deve ser guardado com todas as letras maiúsculas.
- O gênero da música deve ser guardado apenas com seu

- primeiro nome. Os caracteres que subsequentes a eventual espaço em branco devem ser ignorados.

A entrega ao professor deve ser feita em código-fonte.

Para que você seja municiado com conteúdo apropriado para o caso, esta seção abordará tópicos introdutórios de cadeias de caracteres, a classe `String` e alguns dos seus métodos mais utilizados.

Sigamos adiante!

Não pode faltar

Seja bem-vindo a mais um encontro!

Seria legítimo afirmar que, também no mundo da programação de computadores, o conhecimento que possuímos hoje é fruto da evolução dos conhecimentos que adquirimos no passado? Em termos mais específicos, será que conhecer variáveis torna mais suave o caminho até os *arrays*? Ou, se sabemos como usar as funções, então saberemos usar um método com mais facilidade?

As respostas podem variar bastante e serem orientadas, inclusive, por fatores subjetivos. No entanto, é improvável que consigamos afastar o vínculo lógico que existe entre alguns recursos que as linguagens de programação oferecem. E não precisamos ir longe para sustentar este raciocínio: o estudo das *strings* nesta seção remeterá a outros conceitos que você certamente já domina e que se relacionarão com os mecanismos utilizados em uma cadeia de caracteres.

Introdução a sequências de caracteres para programação orientada a objetos

Ao longo do nosso curso temos praticado os mecanismos do Java por meio da manipulação de números e expressões numéricas. No entanto, a linguagem oferece também meios eficientes de tratarmos unidades de texto, aqui mais comumente chamados de cadeias de caracteres ou *strings*.

Segundo Deitel e Deitel (2010, p. 516) uma *string* é uma sequência de caracteres tratada com uma única unidade, que pode incluir letras, dígitos e vários caracteres especiais, como `+`, `-`, `*`, `/` e `$`.

Especificamente em Java, as *strings* são tratadas como objetos da classe `java.lang.String` e, por isso, devem ser declarados e

depois instanciados. Lembre-se sempre de que os tipos primitivos `byte`, `short`, `int`, `long`, `float`, `double`, `char` e `boolean` não requerem instânciação. No próximo item do nosso texto, a classe `String` será mais bem apresentada.



Assimile

String é um tipo texto que corresponde à união de um conjunto de caracteres. Em Java, uma variável do tipo *string* é uma instância da classe `String`, isto é, gera objetos que possuem propriedades e métodos, diferente dos tipos primitivos como `int`, `float`, `double` etc (FURGERI, 2013, p. 73).

Uma sequência de caracteres é sempre apresentada entre aspas. A representação de um nome ou de um endereço, por exemplo, seriam respectivamente "Carlos Silva" e "Avenida Paulista, 1".

Para efeito de declaração de uma *string*, a sequência de caracteres deve ser atribuída a uma referência da classe `String`. Por exemplo, a linha `String nomeDisciplina="Java"` inicializa `nomeDisciplina`, que servirá para referenciar um objeto da classe `String` que contém a sequência de caracteres "Java".

Da mesma forma que os *arrays*, cada elemento de uma *string* também é referenciado por um índice que indica a posição de determinado caráter na sequência de caracteres. Este índice pode valer entre zero e o valor do comprimento da *string* menos um. Tomemos como exemplo a string "Java", de tamanho 4: ela é composta dos caracteres 'J', 'a', 'v', 'a', sendo que 'J' é o primeiro caractere da sequência e 'a' é o último (SANTOS, 2003).

Conforme se depreende do exemplo, um elemento da *string* tomado individualmente é um valor do tipo `char`.



Reflita

O que ocorre se a aplicação tentar acessar um caractere individual ou uma sequência de caracteres que estejam além da dimensão da *string*?

Feita esta introdução, agora nos falta saber como usar os recursos que o Java oferece para manipulação das sequências de caracteres.

Apresentação da classe `String` para programação orientada a objetos

Embora a criação de instâncias da classe `String` dispense a chamada do construtor, os objetos ainda assim podem ser criados pela forma usual, por meio do uso da palavra reservada `new` em conjunto com um construtor que recebe como argumento outra instância previamente criada da classe `String`. Santos (2003) nos ensina que uma instância de `String` também pode ser construída usando a passagem de um *array* de caracteres ao construtor. Observe o exemplo adaptado de Deitel e Deitel (2010).

```
public class ConstrutoresString {
    public static void main (String[] args) {
        char[] ArrayDeChar = {'a', 'n', 'i', 'v', 'e', 'r',
's', 'á', 'r', 'i', 'o'};

        String s = new String("Olá! Feliz");
        String s1 = new String();
        String s2 = new String(s);
        String s3 = new String(ArrayDeChar);
        System.out.printf(" s1 = %s\n s2 = %s\n s3 =
%s\n", s1, s2, s3);
    }
}
```

É possível inicializar os objetos do tipo `String` de várias maneiras, por meio dos construtores da classe. No trecho `String s1 = new String()` do nosso exemplo, o novo objeto é criado sem argumento e sua referência é atribuída a `s1`. Este novo objeto não contém caracteres.

Já no trecho `String s2 = new String(s)`, é instanciado um novo objeto utilizando um construtor que aceita um objeto da classe `String` como argumento e sua referência é atribuída a `s2`. Por fim, no trecho `String s3 = new String(ArrayDeChar)`, atribui-se

à s3 a referência do objeto criado com um construtor que aceita um array de caracteres como argumento. O conteúdo desse array é definido por `char[] ArrayDeChar = {'a', 'n', 'i', 'v', 'e', 'r', 's', 'i', 'd', 'a', 'd', 'e', 'e', 'n', 'g', 'e', 'n', 'h', 'e', 'r', 'a', 'r', 'i', 'a', 's'}`.

Antes de finalizarmos a apresentação da classe `String`, vale destacar:

- Um objeto da classe `String` não poderá ser alterado depois de criado, já que o Java não oferece nenhuma funcionalidade que mude uma sequência de caracteres;
- Não há necessidade de especificar o tamanho (ou dimensão) de um objeto `String` no ato da sua criação.



Pesquise mais

Arnold et. al., 2007 afirmam que, se *strings* imutáveis fossem a única espécie disponível, você teria que criar um novo objeto `String` para cada resultado intermediário de uma sequência de manipulação de *string*. No entanto, o Java usa um objeto `StringBuilder` para construir *strings* a partir de expressões, criando a sequência final apenas quando necessário. Busque informações sobre a classe `StringBuilder` e certifique-se da sua eficiência em criar variáveis de *string* modificáveis. Seguem duas sugestões de fonte de informação:

ARNOLD, K., GOSLING, J., HOLMES, D. **A Linguagem de Programação Java**. 4. ed. Porto Alegre: Bookman, 2007.

PALMEIRA, T. V. **A Classe `StringBuilder` em Java**. Disponível em: <<https://www.devmedia.com.br/a-classe-stringbuilder-em-java/25609>>.

Métodos da classe `String` para programação orientada a objetos

Os métodos da classe `String` oferecem ao desenvolvedor meios de processar os caracteres que compõem a *string*. O interessante é que o Java não economiza nas funcionalidades e disponibiliza uma lista com mais de 60 métodos para serem usados na comparação entre sequências, na concatenação de duas *strings* e em testes dos mais variados.

Nossa aula não abordará todos os métodos, mas não podemos deixar de lançar um olhar geral sobre os mais frequentemente

utilizados. São eles:

- `length()`: retorna o tamanho da *string* expresso em um número inteiro. A formação do tamanho de uma sequência inclui também os caracteres de controle, como espaços e tabulações, por exemplo. Observe o código que segue. Embora de forma simples e sem o laço de repetição devido, ele mostra a utilização do `length` para validação da quantidade de caracteres informada para o CPF do usuário.

```
import java.lang.String;
import java.util.Scanner;
public class ValidaEntrada {
    public static void main (String[] args) {
        Scanner entrada = new Scanner(System.in);
        String cpf = new String();
        System.out.print("Informe o número
do seu CPF: ");

        cpf = entrada.nextLine();
        if (cpf.length() != 11) System.out.
println("CPF inválido");
    }
}
```

- `charAt(int indice)`: retorna o caractere situado na posição dada por índice. O valor do índice varia de 0 até `length - 1`, ou seja, até a dimensão da *string* subtraída de 1.

Tomemos como exemplo a aplicação que segue:

```
import java.lang.String;
public class TestaMetodosString {
    public static void main (String[] args) {
        String s = new String ("Caracteres");
        int c = s.charAt(1);
        System.out.printf("%c",c);
    }
}
```

```
}  
}
```

A saída do programa será simplesmente a apresentação do caractere 'a'. Embora ele esteja na segunda posição da sequência, o índice da letra 'a' é 1, já que a primeira posição da *string* é 0.



Assimile

O método `charAt` retorna o caractere localizado no índice da sequência, que não corresponde ao seu posicionamento físico.

Não se esqueça de formatar a saída com o especificador de formato `%c`. Desta forma o valor `char` será visualizado.

- `equals (Object umObjeto)`: este método está inserido na categoria dos que realizam comparações entre sequências de caracteres. Ele promove a comparação entre uma *string* e o argumento `umObjeto`, que pode ser uma sequência de caracteres ou um objeto da classe `String`. O resultado será `true` se – e somente se – o argumento for um objeto `String` diferente de `null` que represente a mesma sequência de caracteres.

Imagine que em alguma parte da sua aplicação você precise comparar a senha que o usuário deseja criar com a sua confirmação. Uma possibilidade de resolver o problema é usando este método, como segue:

```
import java.lang.String;  
import java.util.Scanner;  
public class ConfirmaSenha {  
    public static void main (String[] args) {  
        Scanner entrada = new Scanner(System.in);  
        String senhaCriada = new String();  
        String senhaConfirmada = new String();  
        System.out.print("Informe a senha  
que você deseja criar: ");
```

```

        senhaCriada = entrada.nextLine();
        System.out.print("Confirme a senha: ");
        senhaConfirmada = entrada.nextLine();
        if (!senhaCriada.equals(senhaConfirmada))
            System.out.println("As  senhas
não são iguais.");
    }
}

```

Um caso mais particular de comparação entre duas sequências é feito com o método `equalsIgnoreCase(String)`. Diferentemente do método `equals`, ele não considera como diferentes as letras maiúsculas e minúsculas. Duas *strings* são consideradas iguais, ignorando-se maiúsculas e minúsculas, se forem do mesmo tamanho e os caracteres correspondentes nas duas sequências forem iguais.

Um segundo método de comparação relacionado ao `equals` é o método `compareTo(String)`, que compara lexicograficamente duas *strings*. O termo pode parecer bem incomum, mas a compreensão da funcionalidade é imediata.

Para que duas strings sejam diferentes, elas devem ter:

- Tamanhos diferentes;
- Caracteres diferentes na mesma posição; ou
- Ambas as características.

A comparação baseia-se no valor Unicode de cada caractere nas duas *strings*. Quando o método encontra o primeiro caractere diferente entre as duas sequências, ele retorna o valor inteiro que representa a diferença entre os caracteres que diferem entre si.



Exemplificando

Observe este código:

```
import java.lang.String;
```

```

public class TestaCompareTo {

    public static void main (String[] args) {

        String s1 = new String ("Astronauta");

        String s2 = new String ("Cosmonauta");

        int resultado = s1.compareTo(s2);

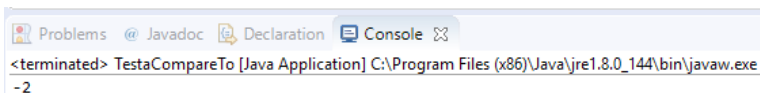
        System.out.printf("%d", resultado);

    }

}

```

Saída:



The screenshot shows a Java IDE with a console window open. The console title is "TestaCompareTo [Java Application] C:\Program Files (x86)\Java\jre1.8.0_144\bin\javaw.exe". The output in the console is "-2".

Neste exemplo de uso do `compareTo`, a comparação é feita entre a sequência "Astronauta" e "Cosmonauta". O primeiro caractere que difere entre as duas é exatamente o primeiro, ou seja, os caracteres "A" e "C". O método então realiza a operação `s1.charAt(1) - s2.charAt(1)`, ou seja, subtrai o primeiro caractere de `s2` do primeiro caractere de `s1` e retorna o valor `-2`, já que a diferença entre C e A é de duas unidades.

O valor do retorno seria 0 se as duas *strings* fossem iguais. Um valor de retorno negativo é obtido quando `s1` é lexicograficamente menor do que `s2`, como é o caso. Por fim, um valor maior que zero seria retornado se `s1` fosse lexicograficamente maior que `s2`. Não deixe de mudar as *strings*, ao executar a aplicação, para averiguar as saídas.

De fato, há muitos outros métodos que permitem a manipulação de strings. Para obter a lista completa deles, acesse a documentação da classe `String`. Você vai observar, entre outras coisas, que esta classe oferece apenas funcionalidades que não têm a capacidade de alterar suas instâncias. Mas, será que devemos carregar essa limitação em todas as nossas aplicações? Prossigamos.

Apresentação da classe StringBuffer

Um buffer de *string* é como uma *string*, mas que pode ser modificada tanto em tamanho quanto em conteúdo.

O Java disponibiliza a classe `StringBuffer`, que permite a criação e manipulação de uma *string* modificável. Uma de suas características mais úteis é a possibilidade de se criar instâncias com um número inicial fixo de caracteres. Essa quantidade inicial pode ser modificada pelo programador ou automaticamente, dependendo da necessidade de crescimento da *string* (SANTOS, 2003).

As operações principais dessa classe estão disponíveis nos métodos que realizam inserção e concatenação entre sequências de caracteres. Os métodos de inserção adicionam os caracteres em um ponto específico da *string* e a concatenação se dá no final da sequência.

Observe o exemplo que segue:

```
import java.lang.String;

public class StringBufferExemplo {
    public static void main (String[] args) {
        StringBuffer nome = new StringBuffer ("José
- ");

        nome.append("Gerente ");
        nome.append("Sênior");

        System.out.printf("O nome e o cargo do
funcionário são: %s", nome);
    }
}
```

A aplicação simplesmente cria um objeto de `StringBuffer` contendo a sequência "José – ". Depois, por meio do uso do método `append`, adiciona as sequências Gerente e Sênior ao nome do funcionário.

Feita a apresentação, chega a hora de tratarmos da situação-problema e da resolução dos exercícios propostos.

Sem medo de errar

A criação completa do código relativo à solução ficará sob sua responsabilidade. Contudo, algumas dicas serão úteis em seu processo de criação.

A aplicação deve solicitar:

- A digitação do nome do autor.
- A digitação do nome da música.
- A digitação do gênero da música.

A aplicação deve armazenar tais dados em objetos da classe `String`. O problema da transformação do primeiro caractere do nome do autor pode ser resolvido com o uso encadeado dos métodos `substring` e `toUpperCase`, conforme exemplo: `nomeAutor.substring(0,1).toUpperCase()`.

A transformação do nome da música em letras maiúsculas deve ser feita com o método `toUpperCase()`.

Por fim, a separação apenas do primeiro nome do gênero da música pode ser feita por meio do uso da expressão `genero.substring(0,genero.indexOf(" "))`.

Assim, a sequência de caracteres deverá ser exibida ao usuário.

Pesquise os métodos sugeridos e faça o melhor uso deles para a resolução do problema.

Bom trabalho!

Avançando na prática

Chamando pelo nome

Descrição da situação-problema

Você acaba de assumir o cargo de desenvolvedor Java da empresa em que trabalha e logo de cara se vê diante da incumbência de aprimorar uma aplicação que vem sendo utilizada há bastante tempo. A comunicação da empresa com os clientes se dá normalmente via mala direta, implementada por uma aplicação Java.

O desenvolvedor a quem você sucedeu não dominava verdadeiramente os métodos da classe `String` e, por falta de recursos de programação, acabou deixando o vocativo do texto com o nome completo do cliente. É fácil de entender: em vez de chamar o leitor pelo primeiro nome, o texto da mala direta o chamava pelo nome completo. Estranho, para dizer o mínimo.

Sua missão é criar um método que receba o nome do cliente (aquele que vai ler o texto) e imprima na mala direta apenas seu primeiro nome. Por exemplo, em vez da mensagem começar por “Oi, Paulo Ricardo Ribeiro da Silva Amaral Peixoto, tudo bem?”, ela deve começar por “Oi, Paulo, tudo bem?”

A aplicação correta de dois métodos da classe `String` resolvem o caso. Mãos à obra!

Resolução da situação-problema

Para que a resolução possa ser executada, ela será apresentada em forma de aplicação, com a sequência que representa nome e sobrenome já pré-fixada. Numa solução real, o nome do cliente deveria ser recebido como argumento.

Eis uma solução possível para o caso:

```
import java.lang.String;

public class AvancandoNaPratica {

    public static void main (String[] args) {

        String nomeCompleto = new String ("Nome
Sobrenome");

        System.out.println(nomeCompleto.
substring(0,nomeCompleto.indexOf(" ")));

    }

}
```

O método `indexOf(String str)` retorna o índice em que se encontra a primeira ocorrência de uma determinada *substring*. O que procuramos é a primeira ocorrência de um caractere em branco, ou seja, um espaço. É ele que separa o primeiro nome dos demais nomes/sobrenomes do cliente.

Por sua vez, o método `substring(int beginIndex, int endIndex)` retorna uma nova *string*, que começa em determinado índice e termina em outro determinado índice. Como estamos tratando do primeiro nome, a *substring* que nos interessa começa no índice zero e termina no índice que contém a primeira ocorrência de um caractere em branco.

É possível chamar os dois métodos no mesmo comando e o retorno será o primeiro nome do cliente.

Faça valer a pena

1. As cadeias de caracteres são sequências alfanuméricas delimitadas como uma sequência de caracteres entre aspas inglesas. A referência em idioma inglês a essa sequência de caracteres se faz com o termo *string*. Uma cadeia de caracteres, ou seja, um *string* é considerado em linguagem Java como um objeto instanciado a partir da classe **String** pertencente ao pacote **java.lang** (MANZANO e COSTA JÚNIOR, 2011, p. 123).

Em relação a alguns dos métodos da classe **String**, analise as afirmações que seguem:

- I. O método **charAt()** retorna o caractere da posição indicada como sendo um dado do tipo inteiro.
- II. O método **equals()** realiza a comparação entre os tamanhos de duas strings e retorna verdadeiro, se forem iguais.
- III. O método **length()** retorna o tamanho da string expresso em um número inteiro, considerando todos os caracteres da sequência.

Assinale a alternativa que contém a afirmação verdadeira.

- a) Apenas as afirmações II e III são verdadeiras.
- b) As afirmações I, II e III são verdadeiras
- c) Apenas a afirmação III é verdadeira.
- d) Apenas a afirmação I é verdadeira.
- e) Apenas as afirmações I e III são verdadeiras.

2.

O tratamento de dados do tipo *string* na linguagem Java é conseguido com a classe **String** que possui internamente um conjunto de métodos para a manipulação e o



tratamento desse tipo de dado. Neste sentido, está presente um grande conjunto de métodos para a manipulação de strings (MANZANO e COSTA JÚNIOR, 2011, p. 123).

Assinale a alternativa que contém os termos que preenchem corretamente as lacunas do texto que segue:

Uma *string* em Java é um arranjo de caracteres dispostos sequencialmente e individualizados por um _____, que começa com o valor _____. Um objeto da classe _____ não poderá ser alterado depois de criado, já que o Java não oferece nenhuma funcionalidade que _____ uma sequência de caracteres.

- a) *Array*, zero, **StringBuffer**, classifique.
- b) Índice, zero, **String**, altere.
- c) Índice, um, **String**, altere.
- d) *Buffer*, um, **StringBuffer**, altere.
- e) Índice, zero, **String**, concatene.

3.



O método `substring()` da classe `String` possibilita desmembrar uma string em determinados trechos. Os parâmetros `x` e `y` são valores do tipo `int`. O parâmetro `x` determina o início do trecho de uma string e o parâmetro `y` determina o final do trecho a ser desmembrado. (MANZANO e COSTA JÚNIOR, 2011, p. 127).

Analise o código que segue e assinale a alternativa que corresponde à saída da aplicação.

```
import java.lang.String;
public class StringQ3 {
    public static void main (String[] args) {
        int r;
        String str1 = new String ("Disciplina Ondas
Eletromagnéticas");
        String str2 = new String ("Disciplina Programação
Orientada a Objetos");
        r = str1.compareTo(str2);
        if (r > 0)
```

```
        System.out.printf("%c", str2.charAt(12));  
    else  
        System.out.printf("%c", str1.  
charAt(12));  
    }  
}
```

- a) r
- b) P
- c) <espaço em branco>
- d) O
- e) n

Seção 4.3

Coleções e arquivos

Diálogo aberto

Depois de estudarmos juntos algumas das estruturas mais interessantes de armazenamento temporário de dados, chegou a hora de aprendermos a torná-los persistentes por meio dos recursos dos arquivos. Mas, antes de tratarmos deles nesta seção, veremos as coleções em Java e suas modalidades. Dentre essas modalidades, escolhemos as listas para um olhar mais aprofundado.

Usamos nossos dois primeiros encontros desta unidade para tratarmos da resolução das duas primeiras partes do problema que a empresa nos delegou:

1) Na primeira seção, desenvolvemos a aplicação que recebia do teclado os dados de código da música e de quantidade de execuções, e os guardava em seus respectivos arrays.

2) Na segunda seção, incluímos nessa aplicação a funcionalidade que faz a leitura e o armazenamento do nome, do gênero e do autor da música. Antes do armazenamento na estrutura apropriada, esses dados passavam por alguns processamentos.

Nossa missão agora consiste em criar uma aplicação que, aproveitando o que foi desenvolvido nas duas primeiras partes da resolução, use recursos de arquivos em Java para realizar a gravação dos dados gerados.

Para que esse desafio possa ser superado com êxito, o conteúdo desta aula abordará tópicos básicos de manipulação de arquivos, com a apresentação da classe `File` e com a oferta de um exemplo prático.

Assim que resolvermos a situação, a aplicação gerada deverá ser entregue ao professor em meio eletrônico ou impresso, a critério dele.

Bom estudo!

Não pode faltar

Olá! Há alguns meses, quando começamos nossa jornada por Orientação a Objetos, você provavelmente achava o assunto pouco atraente e até mesmo o real motivo para ter de estudá-lo não era tão claro. Afinal, com qualquer linguagem de programação e com qualquer paradigma se cria programas flexíveis e elegantes, certo? Não é bem assim.

Os recursos que a orientação a objetos oferece são únicos e sem eles a representação e manipulação de entidades do mundo real seriam feitas com um grau maior de dificuldade. Sem contar a facilidade de se desenvolver em poucas linhas funções, que, em linguagens mais antigas, demandava muito tempo e espaço.

Para encerrar nossa caminhada, ainda precisamos tratar de coleções de objetos, listas e de manipulação de arquivos em Java. Com o conhecimento sobre arquivos, conseguiremos perpetuar as informações geradas durante a execução de uma aplicação, recurso que não dominávamos até agora.

Boa leitura!

Introdução a coleções de objetos

Vivemos às voltas com coleções. As cartas de um baralho, um time de futebol e as músicas da sua *playlist* são exemplos bastante comuns de elementos que fazem sentido quando estão juntos. Como o paradigma da orientação a objetos é muito bom em fornecer meios de facilitar a transição das coisas do mundo real para um programa de computador, é certo que os desenvolvedores seriam brindados com recursos que implementam com muita facilidade as coleções.

Santos (2003) observa que o Java contém classes e interfaces que poupam trabalho e oferecem mecanismos para agrupar e processar objetos em conjuntos. Mas, por qual motivo não podemos agrupar elementos afins em uma estrutura de *array*? Embora a resposta inclua vários itens, é possível resumi-los, conforme Santos (2003): (i) Via de regra, o tamanho de um *array* não pode ser modificado após sua criação. Alterações no tamanho devem ser feitas com a criação de outro *array*, de tamanho diferente, para que os elementos do *array* original possam ser copiados nele.

(ii) Um *array* pode apenas conter elementos de um único tipo, exceto se o desenvolvedor lançar mão de recursos complexos para contornar esta limitação.

(iii) inserções e deleções em um *array* não são operações simples.

O Java disponibiliza, como membro do pacote `java.util`, uma estrutura de coleções. Essa estrutura é formada por interfaces que declaram as operações possíveis de serem realizadas nas várias modalidades de agrupamentos que o Java oferece. O Quadro 4.1 mostra algumas interfaces da estrutura de coleções.

Quadro 4.1 | algumas interfaces da estrutura de coleções.

Interface	Descrição
Collection	A interface raiz na hierarquia de coleções, a partir da qual as interfaces <code>Set</code> , <code>Queue</code> e <code>List</code> são derivadas.
Set	Uma coleção que não contém duplicatas
List	Uma coleção ordenada que pode conter elementos duplicados.
Map	Associa chaves a valores e não pode conter chaves duplicadas.
Queue	Em geral, uma coleção FIFO (primeiro a entrar, primeiro a sair) que modela uma fila de espera. Outras ordens podem ser especificadas.

Fonte: Deitel e Deitel (2010).

Conforme o primeiro item do quadro sugere, a interface `Collection` é a primeira na escala das coleções e dela derivam `Set`, `Queue` e `List`. Esta última, inclusive, será discutida com mais detalhes adiante.

Deitel e Deitel (2010) definem a interface `Collection` como aquela que contém operações de volume, ou seja, operações realizadas na coleção inteira. Esse conjunto de operações inclui ações como adicionar, limpar e comparar objetos em uma coleção.

A documentação oficial da Oracle (ORACLE, s.d.) estabelece que todas as classes que implementam a interface `Collection` devem fornecer dois construtores "padrão": um construtor vazio (sem argumentos), que cria uma coleção vazia, e um construtor com um único argumento de tipo `Collection`, que cria uma nova

coleção com os mesmos elementos do seu argumento. Esse último construtor permite ao usuário copiar qualquer coleção, produzindo uma coleção equivalente ao tipo de implementação desejado.

Algumas implementações de `Collection` têm restrições sobre os elementos que elas podem conter. Por exemplo, algumas implementações proíbem elementos nulos, e algumas têm restrições sobre os tipos de seus elementos. A tentativa de adicionar um elemento inelegível lança uma exceção não verificada (`unchecked`), normalmente `NullPointerException` ou `ClassCastException`.

A tentativa de consultar a presença de um elemento inelegível pode lançar uma exceção, ou pode simplesmente retornar falso. Algumas implementações exibirão o comportamento anterior e algumas exibirão o último.

A classe `Collections`, diferentemente da interface `Collection`, consiste exclusivamente em métodos estáticos que operam ou retornam coleções. Os métodos desta classe lançam uma `NullPointerException` se as coleções ou os objetos de classe fornecidos forem nulos.

Na sequência, trataremos de um item específico das coleções, que tem amplo campo de utilização.



Pesquise mais

Para pesquisar mais sobre coleções, acesse o material abaixo: CAELUM. Collections framework. In: **Apostila Java e Orientação a Objetos**. [s.d, s.l]. Disponível em: <<https://www.caelum.com.br/apostila-java-orientacao-objetos/collections-framework/>>. Acesso em: 11 abr. 2018.

Listas de objetos para programação orientada a objetos

Uma lista (também chamada de sequência) é uma `Collection` ordenada, que pode conter elementos duplicados. Da mesma forma que é implementado nos *arrays*, as listas têm seus elementos identificados por índices e o primeiro elemento é o de índice zero. Todos os métodos de `Collection` estão disponíveis para `List` pelo mecanismo de herança. No entanto, `List` ainda conta com métodos próprios para manipular seus elementos por meio do índice, tratar um intervalo específico de elementos e procurar determinado elemento.



Listas de objetos são coleções onde elementos repetidos podem ocorrer e onde os elementos têm posições definidas (SANTOS, 2003).

De acordo com Santos (2003), a interface `List` declara quais métodos podem ser usados para manipulação de listas e duas classes que implementam essa interface, cada uma com um modo distinto de representação dos objetos nas listas.

Essas classes mencionadas são `ArrayList` e `LinkedList`, e ambas apresentam recursos que as tornam mais vantajosas para determinadas operações em relação a outra. `ArrayList` implementa a lista como um array e tem desempenho superior, a não ser em operações de inserção e deleção de itens na lista. Por sua vez, `LinkedList` oferece melhor desempenho em ações de inserção e deleção, sendo mais lenta em acessos sequenciais.

Vamos analisar o código a seguir. Ele foi construído sem algumas validações importantes e da forma mais sucinta possível, mas é capaz de demonstrar como se insere e se retira elementos de um `ArrayList`.

```
import java.util.ArrayList;
import java.util.Scanner;

public class ArrayListExemplo {

    public static void main (String[] args) {

        Scanner entrada = new Scanner(System.in);

        ArrayList<Integer> lista = new
ArrayList<Integer>(); //Cria um arraylist vazio.

        int elemento;

        while (true) { //insere mais um elemento na
lista até que CTRL+z seja digitado.

            System.out.print("Insira um número
inteiro positivo para ser incluído ou -1 para terminar: ");

            elemento = entrada.nextInt();

            if (elemento==-1) {

                break;
            }
        }
    }
}
```

```

    }
    lista.add(elemento);
}
System.out.println("\nA lista que você criou
contêm os seguintes elementos: ");
for (int i: lista) System.out.println(i);

System.out.print("\nInforme o índice do
elemento que você deseja retirar da lista: ");
entrada.reset();
elemento = entrada.nextInt();
lista.remove(elemento);
System.out.println("Agora sua lista ficou
assim: ");
for (int i: lista) System.out.println(i);
}
}

```

Basicamente, a aplicação solicita que o usuário inclua elementos no `ArrayList` até que -1 seja digitado. Na sequência, a lista é exibida e o usuário deve informar em qual índice se encontra o valor que deseja excluir. A lista resultante é exibida ao final do processo.

Em linhas gerais, assim tratamos da forma com que o Java implementa estruturas de coleções. As facilidades de uso das modalidades de coleções vão além do que foi abordado nesta lição, mas sua experiência em lidar com interfaces e métodos de classes pré-definidas certamente completarão as lacunas que permaneceram neste assunto.

Introdução a arquivos em Java

A motivação para estudarmos arquivos em Java é bem conhecida: os dados que armazenamos em variáveis e vetores se perdem quando os valores são substituídos nessas estruturas ou quando o programa encerra sua execução. Mesmo com sua utilidade resguardada, variáveis e vetores não foram feitos para preservar valores além da existência da aplicação na RAM.

Essa limitação é superada pelos arquivos. De modo consciente ou não você usa arquivos todos os dias. Quer um exemplo? A leitura deste texto só está sendo possível por causa deles.

Arnold et al (2007) afirmam que o Java oferece diversos pacotes que permitem a movimentação de dados para e a partir de programas. Tais pacotes diferem entre si nos tipos de abstrações que eles providenciam para trabalhar com operação de Entrada e Saída (E/S).

O pacote `java.io` define E/S em termos de fluxos, ou *streams*. O termo fluxo se refere a dados ordenados que são lidos ou gravados em um arquivo. Eles possuem uma origem (fluxos de entrada) ou um destino (fluxo de saída).



Reflita

Em nossas aplicações, sempre que precisamos permitir ao usuário a inserção de dados usando o teclado, usamos um objeto de `System.in`. Será que esse objeto está relacionado a fluxo?

O processamento de arquivos é um subconjunto das operações de fluxo do Java, que incluem, por exemplo, leitura e escrita de dados contidos na memória. A parte mais interessante de se lidar com arquivos é a que as classes de E/S disponibilizadas pelo Java isolam os desenvolvedores de detalhes de tratamento de fluxos específicos do Sistema Operacional. A linguagem “enxerga” cada arquivo como um fluxo sequencial de bits, conforme mostra a Figura 4.1.

Figura 4.2 | Visualização do Java de um arquivo de n bits.

0	1	2	3	4	5	6	7	8	9	...	n-1

Fonte: Deitel e Deitel (2010).

Um dos aspectos da facilidade para o desenvolvedor se manifesta, por exemplo, na questão da verificação do atingimento do fim de arquivo. A aplicação que está processando arquivos recebe uma indicação do Sistema Operacional quando o fim do fluxo é atingido, sem que a aplicação necessite ter ciência de como a plataforma representa fluxos ou arquivos.

Todo o processamento que nos interessa é realizado pelas classes do pacote `java.io`, já mencionado anteriormente.



Além do `java.io`, o Java oferece outros pacotes úteis no tratamento de arquivos. Dois desses pacotes são `java.nio` e `java.net`. O primeiro define as operações de entrada e saída em termos de buffers e canais. O pacote `java.net` dá suporte específico para operações em redes, com base no uso de soquetes. Para saber mais sobre esses pacotes, consulte os links a seguir:

BREGAGNOLI, E. E. **Trabalhando com NIO.2: A nova API de I/O do Java 7 – Revista Java Magazine 111**. [s.d., s.l.] Disponível em: <<https://www.devmedia.com.br/trabalhando-com-nio-2-a-nova-api-de-i-o-do-java-7-revista-java-magazine-111/26809>>. Acesso em: 11 abr. 2018.

ROCHA, H. **Fundamentos de Sockets**. [s.d., s.l.]. Disponível em: <http://www.argonavis.com.br/cursos/java/j100/java_19.pdf>. Acesso em: 11 abr. 2018.

Uma das principais classes deste pacote é `File`. Ela oferece recursos para recuperar informações sobre arquivos e diretórios em disco, embora não seja capaz de abrir arquivos ou processá-los.

A documentação oficial da Oracle (ORACLE, [s.d.]) identifica a classe `File` como uma representação abstrata de nomes de arquivos e diretórios.

As interfaces de usuário e os sistemas operacionais usam *strings* dependentes do sistema para nomear arquivos e diretórios. Essa classe apresenta uma visão de estrutura de diretórios não dependente do sistema.

Um sumário dos principais métodos da classe `File` é fornecido no Quadro 4.2.

Quadro 4.2 | Alguns dos principais métodos da classe `File`.

Modificador e Tipo	O método e sua descrição
<code>boolean</code>	<code>canExecute()</code> Retorna <code>true</code> se a aplicação for capaz de executar o arquivo indicado no caminho de diretório.

boolean	<code>canRead()</code> Retorna true se a aplicação for capaz de ler o arquivo indicado no caminho de diretório.
boolean	<code>canWrite()</code> Retorna true se a aplicação for capaz de modificar o arquivo indicado no caminho de diretório.
boolean	<code>delete()</code> Apaga o arquivo ou o diretório indicado no caminho de diretório.
void	<code>deleteOnExit()</code> Solicita que o arquivo ou o diretório indicado no caminho de diretório seja deletado quando a máquina virtual encerrar.
boolean	<code>exists()</code> Retorna true caso o arquivo ou o diretório indicado no caminho de diretório exista.
String	<code>getAbsolutePath()</code> Retorna uma sequência de caracteres contendo o caminho absoluto do arquivo ou diretório.
long	<code>getFreeSpace()</code> Retorna a quantidade de bytes não alocados na partição identificada pelo caminho de diretório.
String	<code>getName()</code> Retorna a cadeia de caracteres contendo o nome do arquivo ou do diretório.
String	<code>getParent()</code> Retorna cadeia de caracteres contendo o diretório pai do arquivo ou null, se o nome do diretório não for o nome de um diretório pai.
boolean	<code>isFile()</code> Retorna true se o arquivo indicado no nome de diretório for um arquivo normal.

Fonte: Oracle ([s.d.])

O pacote `java.io` oferece diversas outras classes para entrada e saída de fluxos de dados. Antes de avançarmos para um exemplo usando a classe `File`, vale o registro: salvo indicação em contrário, passar um argumento nulo para um construtor ou método, em qualquer classe ou interface neste pacote, fará com que uma `NullPointerException` seja lançada.

Depois de abordados esses temas, resta-nos tratar da aplicação de alguns métodos da classe `File`.

Manipulação de arquivos para programação orientada a objetos

O exemplo que preparamos para esta última parte do nosso texto tem relação com a classe `File` e se trata, na verdade, de descobertas dos dados sobre o arquivo e/ou seu caminho por meio da aplicação de alguns métodos da classe. Observe o exemplo que segue.



Exemplificando

```
import java.io.File;

import java.util.Scanner;

import java.io.FileWriter;

import java.io.PrintWriter;

import java.io.IOException;

public class DemonstraClasseFile {

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

        Scanner entrada = new Scanner(System.in);

        FileWriter arquivo = new FileWriter("C:\\arqttexto.
            txt");

        PrintWriter gravaArquivo = new
            PrintWriter(arquivo);

        gravaArquivo.print("Este arquivo texto foi
            gravado com sucesso%n");

        arquivo.close();
```

```

        System.out.print( "Informe o nome do arquivo ou
do diretório: ");

        analisaOCaminho (entrada.nextLine());

    }

    public static void analisaOCaminho(String caminho) {

        File nome = new File (caminho);

        if (nome.exists()) {

            System.out.printf("%s%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n",nome.getName(), " existe",

                (nome.isFile() ? "É um arquivo" : "não é
um arquivo"),

                (nome.isDirectory()? "É um diretório" :
"Não é um diretório"),

                (nome.isAbsolute() ? "É um caminho
absoluto" : "Não é um caminho absoluto"),

                "Modificado em: ", nome.lastModified(),
                "Tamanho: ",nome.length(),

                "Caminho: ", nome.getPath(), "Caminho
absoluto: ", nome.getAbsolutePath(),

                "Pai: ", nome.getParent());

            if (nome.isDirectory()) {

                String[] directory = nome.list();

                System.out.println (" \n \nO diretório contém:
\n");

                for (String directoryName: directory )
                    System.out.println(directoryName);

            }

        }

        else {

```



```

        System.out.printf("%s%s",    caminho,    "    não
        existe.");

    }

}

}

```

A aplicação começa declarando um objeto da classe `FileWriter`, que permitirá a criação de um arquivo texto, aqui chamado `arqtexto.txt`.

A primeira ação visível do programa é a solicitação ao usuário para que informe o caminho do arquivo ou seu nome. A sequência de caracteres digitada é passada ao método `analisaOcaminho`, que cria um novo objeto do tipo `File` e atribui sua referência a `nome`. O método `exists` retorna se o nome informado existe no meio de armazenamento (disco). Se o nome não existir, uma mensagem correspondente será exibida. Caso exista, o nome do arquivo ou diretório é enviado à tela.

Na sequência, a aplicação realiza os testes que retornam se o objeto em questão é um arquivo ou um diretório e se o caminho é absoluto. O programa também informa a última modificação, o tamanho do arquivo e o conteúdo do diretório.

Sem medo de errar

É chegado o momento de tornarmos persistentes os dados gerados em nossos programas. Em específico, a situação que nos foi colocada pede que os dados a serem gravados sejam aqueles relacionados às músicas oferecidas por aplicação on-line.

Assim, você deve reunir em sua aplicação classes e métodos oferecidos pelo Java, apropriados para gravar os dados de nome da música, nome do autor, gênero e quantidade de execuções.

Para que você tenha parâmetro para iniciar a resolução, observe o código que segue:

```

    public static void main(String[] args) throws
IOException {
        File arquivoMusica = new File("/musica.txt");
        arquivoMusica.createNewFile(); //Cria o novo
arquivo
        FileWriter fw = new FileWriter(arquivoMusica);
//cria objeto da classe FileWriter
        BufferedWriter bw = new BufferedWriter(fw); //cria
objeto da classe BufferedWriter
        bw.write("Dados da música"); //Escreve "Dados da
música" no arquivo texto.
        bw.newLine(); //Adiciona uma linha no arquivo texto
        bw.close(); //fecha o BufferedWriter
        fw.close(); //fecha o FileWriter
    }

```

Ele explicita os meios para abertura, gravação e fechamento de arquivo.

Avançando na prática

Mudando a estrutura de armazenamento

Descrição da situação-problema

Assim que seu colega de trabalho deixou o emprego, Merkel ficou responsável pela manutenção de alguns dos seus programas. Em um deles, a estrutura usada para armazenamento de determinados valores (não vem ao caso quais valores e para quais propósitos, mas sabemos que se tratam de números inteiros) era um *array*.

Ocorre que, frequentemente, certos números do *array* precisavam ser removidos e outros incluídos, conseqüentemente, a necessidade de se alterar o tamanho da estrutura era muito comum, o que custava muito tempo e sacrifício de desenvolvimento.

Sua missão é ajudar Merkel a encontrar outro meio de armazenar os números, de tal modo que inclusões, exclusões e alterações no tamanho se tornem operações simples na estrutura. Na prática, você deverá criar aplicação que simplesmente mova

os elementos do vetor para essa nova estrutura.

Bom trabalho!

Resolução da situação-problema

Conforme especificado na descrição do caso, a tarefa que se coloca é a criação de programa que simplesmente passe os elementos do vetor para uma nova estrutura de armazenamento de dados, mas apta a passar por inclusões, exclusões e alterações de tamanho.

O primeiro passo, então, é escolher tal estrutura e, pela circunstância apresentada, a lista é o meio ideal. Oficialmente, seu conceito é o de uma coleção ordenada que pode conter elementos duplicados. Vejamos como fica a aplicação:

```
import java.util.List;
import java.util.ArrayList;
public class AvancandoPratica4_3 {
    public static void main (String[] args) {
        int original[] = {7, 25, 42, 8, 22, 13, 8,
            1, 9, 10};
        List<Integer> list = new ArrayList< Integer
        >();
        for (int e: original)
            list.add(e);
        System.out.println("Nova estrutura: ");
        for (int count=0; count<list.size(); count++)
            System.out.printf("%s      ", list.
            get(count));
    }
}
```

Sabemos que o *array* original já possui valores inseridos. Para facilitar o exercício, reproduzimos esses valores em sua inicialização. Na sequência, foi criada um *ArrayList* de inteiros, que receberá o conteúdo do *array* original. O comando *for* utilizado é o aprimorado para percorrer elementos de uma sequência e, em seu escopo, o método *add* é chamado para

adicionar o elemento na lista.

Por fim, a aplicação apenas exibe em tela os elementos inseridos na lista, que são os mesmos do array original. Pronto! Agora será possível manipular facilmente o conteúdo da lista.

Faça valer a pena

1.



O pacote `java.io` define vários tipos de *streams* [...] e a maioria tem ambas as variantes de *streams* de bytes e *streams* de caracteres. Alguns desses *streams* definem propriedades comportamentais gerais. Por exemplo, *streams filters* são classes abstratas que definem *streams* com alguma operação de filtragem aplicada à medida que dados são lidos ou escritos por outro stream (ARNOLD et al, 2007, p. 462).

No contexto do estudo dos arquivos em Java, fluxo significa:

- a) Tipo de classe que determina a ordem em que os dados são gravados em arquivo.
- b) Dados ordenados que são lidos ou gravados em um arquivo.
- c) Dados que são lidos ou gravados apenas em arquivos ordenados.
- d) Dados ordenados que são descarregados na tela durante a execução da aplicação.
- e) Método que ordena dados antes de serem gravados em uma stream do Java

2. Coleções (algumas vezes chamadas contêineres) são receptáculos que permitem a você armazenar e organizar objetos de maneiras úteis para acessos eficientes. O que será eficiente depende de como você precisa usar a coleção, e assim existem coleções de muitos sabores (ARNOLD et al, 2007).

Considerando as interfaces próprias para manipulação de coleções de dados, analise as afirmações que seguem e assinale a alternativa que contenha as afirmações verdadeiras.

I. Set representa associações de valores a chaves e não pode conter elementos duplicados.

II. **List** representa uma coleção, ordenada ou não, que permite elementos duplicados.

III. **Queue** representa uma coleção que modela uma fila, em que o primeiro elemento a entrar é o primeiro a sair.

- a) Apenas as afirmações I e II são verdadeiras.
- b) Apenas as afirmações I e III são verdadeiras.
- c) Apenas as afirmações II e III são verdadeiras.
- d) Apenas a afirmação III é verdadeira.
- e) As afirmações I, II e III são verdadeiras.

3. Os computadores armazenam arquivos em dispositivos de armazenamento secundários. Os dados mantidos nos arquivos são chamados dados persistentes, por que existem além da duração da execução do programa. Uma linguagem de programação deve ser capaz de processar arquivos (DEITEL; DEITEL, 2010).

Assinale a alternativa que contém os termos que preenchem corretamente as lacunas do texto que segue:

Essencialmente, arquivos podem ser considerados como _____, nos quais os valores são armazenados externamente a um _____. No nível mais baixo de abstração, arquivos são conjuntos de _____, semelhantes a arranjos, armazenados fora da memória principal, comumente em um _____.

- a) contêineres, disco, fluxos, contêiner.
- b) coleções, programa, bytes, contêiner.
- c) contêineres, programa, bytes, disco.
- d) coleções, código-fonte, programas, disco.
- e) contêineres, disco, fluxos, programa.

Referências

ARNOLD, K., GOSLING, J., HOLMES, D. A **Linguagem de Programação Java**. 4. ed. Porto Alegre: Bookman, 2007.

CAELUM. Um pouco de arrays. In: **Apostila Java e Orientação a Objetos**. Disponível em: <https://www.caelum.com.br/apostila-java-orientacao-objetos/um-pouco-de-arrays/#arrays-de-referencias/>. Acesso em: 18 dez. 2017.

DEITEL, H. M., DEITEL, P. J., **Java – Como Programar**. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

FURGERI, S. **Java 7 Ensino Didático**. 2. ed. São Paulo: Érika, 2013.

JAVA PROGRESSIVO. **Classe Arrays (Arrays Class): aprenda a manusear (copiar, ordenar, buscar e manipular) Arrays**. [s.p, s.d, s.l]. Disponível em: <<http://www.javaprogressivo.net/2012/09/como-usar-classe-arrays-java.html>>. Acesso em: 11 abr. 2018.

MANZANO, J. A. N. G., COSTA JÚNIOR, R. A. **Java SE 7 Programação de Computadores - Guia Prático de Introdução, Orientação e Desenvolvimento**. 1. ed. São Paulo: Érica, 2011.

ORACLE. **Interface Collection<E>**. [s.d., s.l.]. Disponível em: <<https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html>>. Acesso em: 11 abr. 2010.

SANTOS, R. **Introdução à Programação Orientada a Objetos usando Java**. Rio de Janeiro: Campus, 2003.

Anotações

[illegible]

Anotações

[illegible]

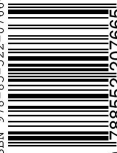
Anotações

[illegible]

Anotações

[illegible]

ISBN 978-85-522-0766-5



9 788552 207665 >