PROGRAMAÇÃO ORIENTADA A OBJETOS

RICARDO DE ALMEIDA ROCHA

SOBRE OS AUTORES

Ricardo de Almeida Rocha

Especialização em Desenvolvimento Orientado a Objetos em Java - Cesumar - PR - 2009.
Graduação em Ciência da Computação - UEM - PR - 2008.

Graduado pela Universidade Estadual de Maringá em Ciência da Computação e Pós-Graduado em Desenvolvimento Orientado a Objetos em Java. Trabalhou na empresa Benner Sistemas, em Maringá-PR, atuando com as linguagens de programação Delphi e C#, e na empresa CISS, em Dois Vizinhos-PR, com a linguagem PowerBuilder. Atualmente, é sócio-administrador na empresa Lojacor Tintas, em Pato Branco-PR

Introdução

A linguagem de programação Java se tornou popular ao longo dos anos, sendo utilizada em diversas áreas de atuação, como dispositivos móveis, aplicações Web e, também, aplicações desktop, porém com menos intensidade.

O Java é uma linguagem de programação que utiliza o paradigma da orientação a objetos, diferenciando-se das linguagens de programação estruturais, que tinham bastante aceitação, como Delphi e C. Java não foi a linguagem precursora no paradigma orientado a objeto, mas foi a que teve a maior visibilidade.

Este livro é dividido em 4 unidades, as quais têm como objetivo introduzir os conceitos da orientação a objeto aplicados na linguagem de programação Java, para produzir projetos com conectividade a um banco de dados.

A Unidade 1 apresenta alguns tipos de linguagens de programação e seus conceitos, desde as linguagens de baixo nível até as linguagens orientadas a objeto. Aborda, também, a instalação e a configuração do ambiente necessário para o correto funcionamento do Java, como a utilização do ambiente de desenvolvimento NetBeans IDE. A sintaxe de programação do Java, como declaração de classes, variáveis e atributos, as estruturas condicionais e as estruturas de repetições são exploradas na última seção desta unidade.

A segunda unidade aborda as características específicas da orientação a objetos, que são utilizadas em todas as linguagens que usam esse paradigma. A primeira seção conceitua e explica como é realizada a divisão das classes e dos pacotes em Java, seus modificadores de acesso e o encapsulamento de seus atributos. A segunda seção apresenta e exemplifica o conceito de herança, que é abrangente e muito utilizado em Java, principalmente para a implementação de padrões de projetos. Continuando os conceitos de orientação a objetos, a terceira seção abrange o

polimorfismo, mostrando como um objeto pode assumir diversas formas, desde que essas estejam relacionadas por meio da herança. Nessa seção, ainda, é introduzida a questão de interfaces e sobreposição de métodos. A unidade termina explicando o essencial tratamento de exceções no Java, para evitar que o sistema entre em um estado de falha, não possibilitando a continuação da sua execução.

A Unidade 3 deste livro tem como objetivo realizar uma introdução ao vasto mundo das interfaces gráficas implementadas em modo desktop. Conceituaremos a diferença entre AWT x Swing; a segunda é utilizada para as implementações dessa unidade, que aborda o comportamento dos componentes principais e o tratamento de evento, para que o componente realize a interação entre o usuário e a lógica do sistema. A interface gráfica do Java é construída em cima de gerenciadores de layout, que organizam e distribuem os componentes nessa interface. Os gerenciadores de layout mais utilizados são introduzidos na última seção, mostrando como eles se comportam à medida que os componentes são adicionados.

Na última unidade, são abordados os conceitos de banco de dados, a linguagem específica para realizar operações nos bancos de dados, a SQL e a instalação e a configuração de um sistema gerenciador de banco de dados. A terceira e a quarta seções dessa unidade realizam a conectividade de um SGBD dentro de uma aplicação Java, estruturando-a de uma forma desacoplada, por meio do conceito de MVC (Model-View-Controller), e a introdução de padrões de projetos com a utilização do padrão DAO (Data Access Object).

UNIDADE I

Introdução à Orientação a Objetos

Ricardo de Almeida Rocha

Esta unidade trará os princípios básicos de programação orientada a objetos, abordando os conceitos de objetos e classes, para que servem os atributos e o porquê de os métodos existirem. As linguagens de programação não foram sempre orientadas a objetos e não utilizavam compiladores para transformarem código de alto nível em um programa executável. Os primeiros computadores eram enormes, demandavam bastante tempo para executar operações e tinham programação árdua.

Serão discutidos o histórico das linguagens de programação, desde as linguagens de máquina até as orientadas a objetos, assim como os conceitos básicos para a programação orientada a objetos, que são essenciais para o decorrer do nosso estudo. Além disso, são apresentadas a instalação e a configuração do ambiente correto para o desenvolvimento na linguagem Java, a utilização da IDE NetBeans, que dispõe de ferramentas para facilitar a programação em Java, e a sintaxe dos comandos e das estruturas de decisão e repetição fundamentais para o desenvolvimento em Java.

Linguagens de Programação

Histórico acerca das linguagens de programação

Um dos primeiros computadores a realizarem cálculos era eletromecânico, constituído de circuitos elétricos (relés), construído por Konrad Zuse em 1936. A forma como esses relés eram ligados definia como a máquina realizaria seu processamento, e seus resultados eram exibidos em fitas perfuradas.

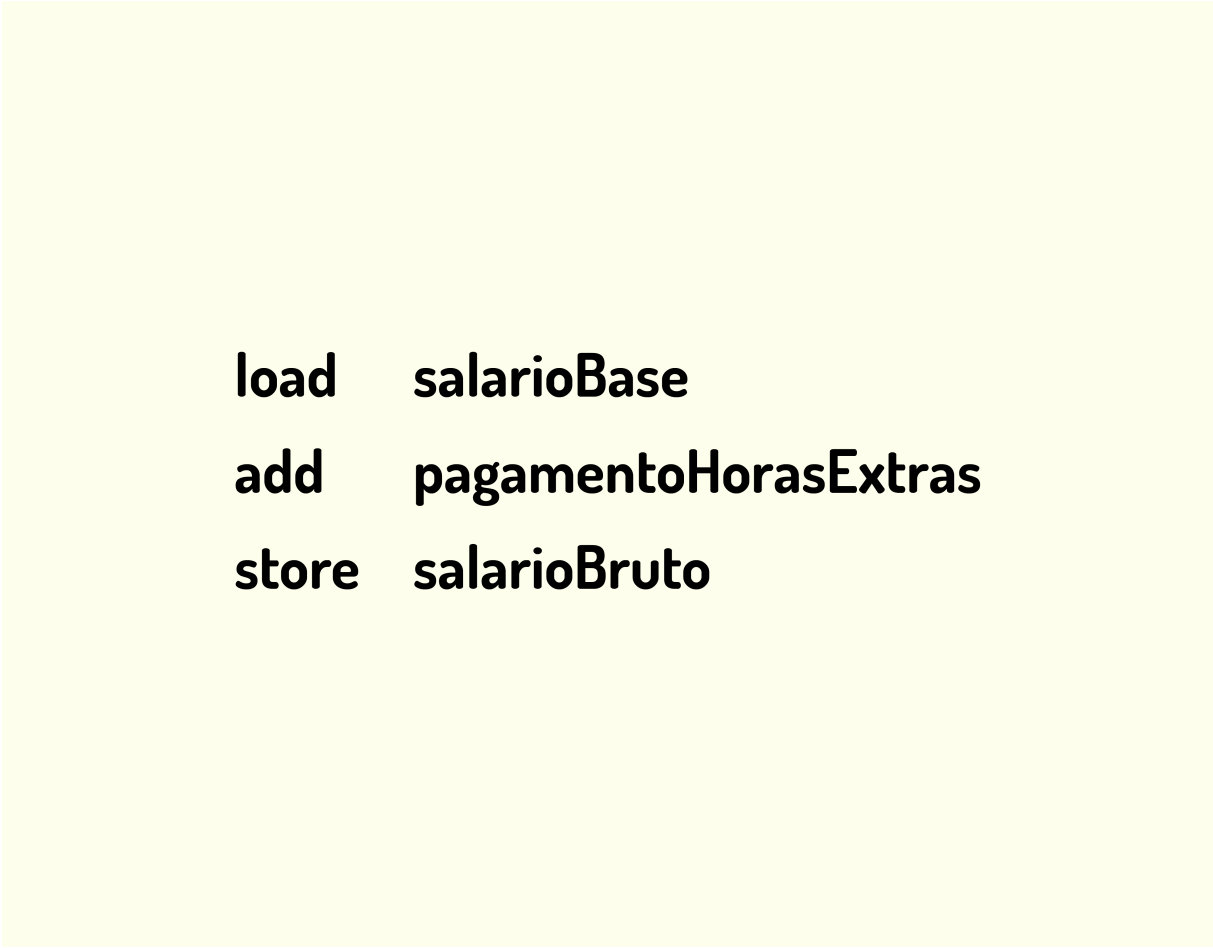
O ENIAC (*Electronic Numerical Integrator And Computer* - Computador e Integrador Numérico Eletrônico) foi projetado e concluído na Universidade da Pennsylvania em 1946; com projeto de J. Presper Eckert e John Mauchly, foi considerado o primeiro computador eletrônico utilizável, necessitando de válvulas a vácuo para funcionar, aproximadamente 18000 válvulas, no lugar dos transistores, que ainda não tinham sido inventados. A programação no ENIAC era feita por cabos que eram conectados nos painéis, e a combinação de como esses cabos eram conectados representava os problemas que deveriam ser solucionados. Para cada problema diferente a ser resolvido, os cabos deveriam ser desconectados e conectados novamente, em outra combinação (HORSTMANN, 2004).

Os programadores utilizam linguagens de programação para escreverem instruções para o computador, as quais definem como o computador operará para executar determinados programas. Essas linguagens podem ser entendíveis diretamente pelo computador ou necessitar de um "tradutor" para convertê-las em uma linguagem compreensível pela máquina (DEITEL, 2010).

As linguagens podem ser definidas como linguagens de máquina, linguagens *assembly* e linguagens de alto nível. A linguagem de máquina é a nativa do computador, ou seja, ele entende diretamente esse tipo de linguagem, não sendo necessária a utilização de um "tradutor". Essas linguagens são formadas, geralmente, por *strings* hexadecimais, que são convertidas para a forma binária, composta de 0s e 1s, instruindo os computadores a realizarem as operações solicitadas. As linguagens de máquinas são específicas de determinadas plataformas, portanto, um programa escrito em uma linguagem de máquina para determinado computador não funcionará em outro com arquitetura diferente (DEITEL, 2010).

Linguagem de baixo nível

Para facilitar a programação, os números foram trocados por abreviações em inglês para as operações elementares dos computadores, chamadas de linguagem *assembly*. Para realizar a tradução das palavras escritas em *assembly* para a linguagem de máquina, foram criados os *assemblers* (DEITEL, 2010). A Figura 1.1 mostra um trecho de um programa escrito em *assembly* que soma os ganhos de horas extras ao salário base e armazena no salário bruto.

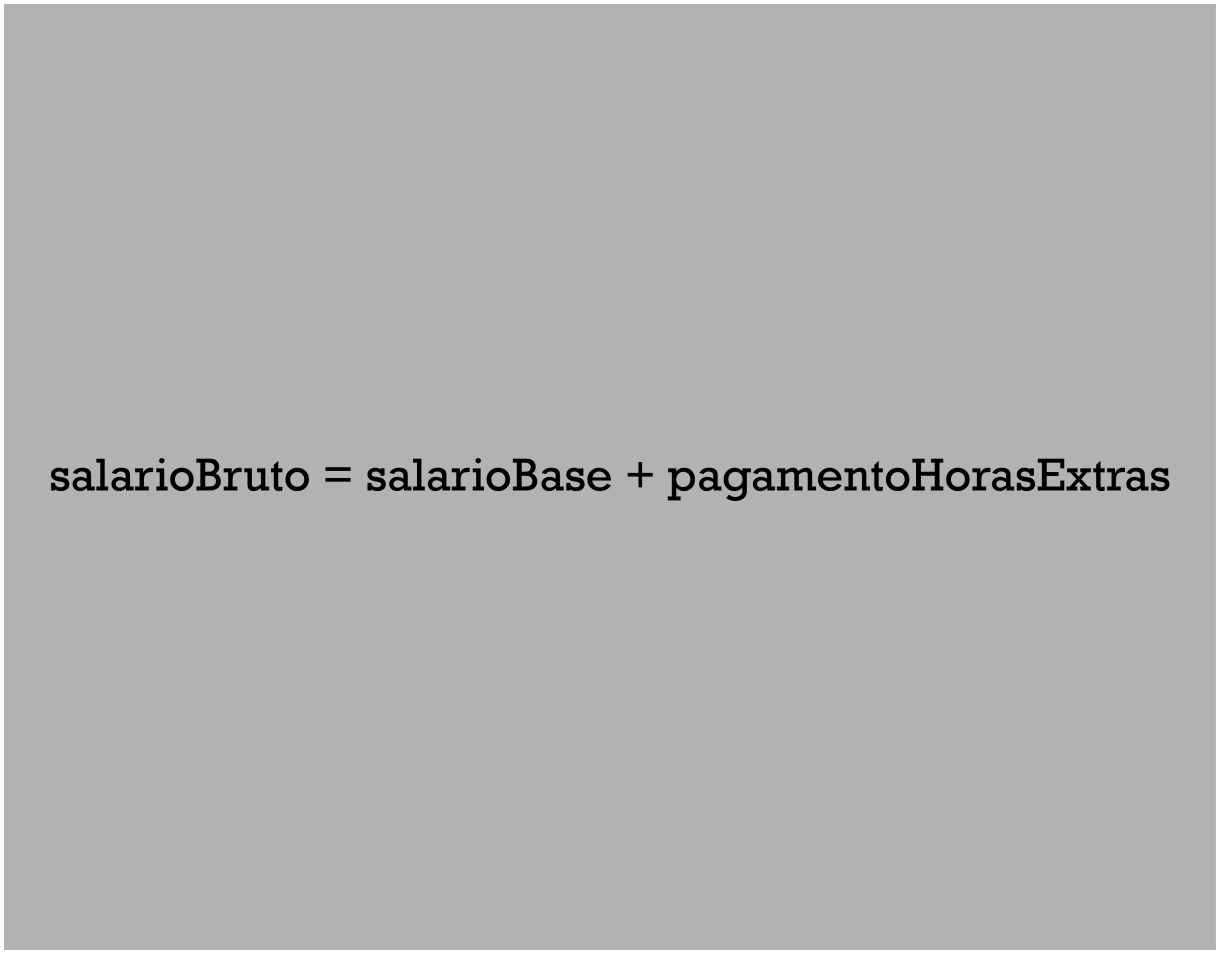


1FIGURA 1.14 - Linguagem Assembly FONTE: Deitel (2010, p. 5).

Contudo, para cada processador, poderia existir pelo menos uma linguagem *assembly* correspondente, pois essa linguagem era diretamente dependente da arquitetura do processador utilizado (GUDWIN, 1997). A diferenciação entre as linguagens *assembly* era um complemento ao seu nome, referenciando para qual processador cada linguagem foi criada, por exemplo, Assembly 8085, para microprocessadores de 8 bits, e, para microprocessadores de 16 bits, as linguagens Assembly 8086 e Assembly 8088.

As linguagens *assembly* disseminaram a utilização dos computadores, porém ainda era trabalhoso escrever diversas instruções para realizar tarefas simples. Assim, foram desenvolvidas linguagens de alto nível, que continham instruções únicas para realizar diversas tarefas básicas. A linguagem era convertida em

linguagem de máquina por meio de um programa chamado compilador. Os programas, agora, eram escritos em inglês cotidiano, combinados com operações matemáticas. As instruções em *assembly* representadas anteriormente poderiam ser escritas em uma linguagem de alto nível, como exibida na Figura 1.2.



1FIGURA 2.14 - Linguagem de alto nível FONTE: Deitel (2010, p. 6).

Além dos compiladores, foram criados os interpretadores, que executam um programa diretamente, sem ser necessário consumir um tempo considerável com o processo de compilação; a execução do programa diretamente do interpretador,

porém, pode ser mais lenta que a execução normal após a compilação. O Java é uma linguagem que utiliza uma combinação entre compilação e interpretação (DEITEL, 2010).

As linguagens de programação de alto nível podem ser divididas em (GUDWIN, 1997):

- Linguagens não estruturadas.
- Linguagens procedurais.
- Linguagens orientadas a objeto.

Existem outras linguagens com aplicações específicas, como SQL, que realiza operações em Banco de Dados, LaTeX, para formatação de texto, MatLab, que realiza simulações matemáticas, dentre outras que não serão citadas neste texto.

Linguagens não estruturadas

As linguagens não estruturadas têm comandos que não estão tão vinculados ao processador utilizado quanto as linguagens de baixo nível, podendo ser utilizadas em diversas plataformas. Dentre as linguagens não estruturadas, destacam-se: a COBOL (*COmmon Business Oriented Language*), desenvolvida em 1959 especificamente para aplicações comerciais, que utilizam grande volume de dados, é, ainda, bastante utilizada em aplicações mais pesadas e que demandam uma alta confiabilidade (como bancos); o Basic (*Beginners All-purpose Symbolic Instruction Code*), desenvolvido em 1963 com o intuito de ser uma linguagem simples e interativa, a qual poderia ser executada ao mesmo tempo em que é programada. O Basic tem uma variante orientada a objetos conhecida por Visual Basic (GUDWIN, 1997).

As linguagens não estruturadas permitem a utilização de desvios incondicionais, que são instruções que mudam o fluxo da execução do programa sem necessitar satisfazer uma determinada condição, "pulando" trechos do código por meio de

comandos, como a instrução GOTO numeroDaLinha, que desvia o fluxo de execução para a linha especificada. A utilização desses comandos poderia acarretar problemas, pois instruções essenciais para o programa poderiam nunca ser executadas, e, também, dificultaria o entendimento da lógica do programa, acarretando problemas em possíveis manutenções.

Linguagens procedurais

Ao contrário das linguagens não estruturadas, as linguagens procedurais têm estruturas de controle que promovem o teste de condição (*if-then-else*), controlam repetições de laços (*for*, *while*, *do*) e seleção de alternativas (*switch*, *case*) e dividem o programa em blocos chamados funções ou procedimentos (GUDWIN, 1997). Ou seja, as linguagens procedurais são um subtipo das linguagens estruturadas. Dentre as linguagens procedurais, destacam-se o C e o Pascal.

O C evoluiu a partir do BCPL e do B. Em 1967, Martin Richards criou o BCPL como uma linguagem de programação para escrever softwares de sistemas operacionais e compiladores, enquanto, em 1970, Ken Thompson criou o B, com algumas características herdadas do BCPL, para criar versões do sistema operacional UNIX na Bell Laboratories (DEITEL, 2010). O C, implementado em 1972, foi criado por Dennis Ritchie, também na Bell Laboratories, tornando-se mundialmente conhecido como a linguagem de programação do sistema operacional UNIX. Os sistemas operacionais atuais de uso geral são escritos em C ou C++ (DEITEL, 2010).

O Pascal foi desenvolvido entre 1967 e 1968 por Niklaus Wirth com propósitos educacionais, mas, posteriormente, tornou-se uma linguagem de propósitos gerais, com uma sintaxe bem definida, não sendo tão flexível e perdendo lugares de utilização para o C; nas suas versões mais recentes, contudo, há suporte a algumas funcionalidades da orientação a objetos (GUDWIN, 1997).

Linguagens orientadas a objetos

As linguagens orientadas a objetos são por si só uma técnica. Uma linguagem será chamada de orientada a objetos se ela suportar o estilo de programação orientado a objetos. A primeira linguagem orientada a objetos foi o Simula, desenvolvido em 1967, antecedendo o SmallTalk, de 1970 (GUDWIN, 1997). Hoje em dia, há uma grande diversidade de linguagens orientadas a objetos, desde algumas linguagens mais antigas, como C++, até mais recentes, como Java e C#.

C++ é uma extensão do C, linguagem desenvolvida no início da década de 1980, fornecendo vários recursos que aprimoram a linguagem C, capacitando-a para a orientação a objetos. O C++ é uma linguagem que pode ser programada no estilo C ou no estilo orientado a objetos, podendo ser classificada como uma linguagem híbrida (DEITEL, 2010).

O Java foi criado em 1991 por James Gosling, na Sun Microsystems. Seu primeiro nome foi Oak, homenageando uma árvore de carvalho que ficava próxima ao seu escritório, porém já existia uma linguagem com esse nome. Oak virou Java em uma cafeteria, baseando-se no nome de origem de um café importado, a ilha de Java (DEITEL, 2010).

Java foi formalmente anunciado em 1995, com utilização diretamente para aplicações Web, possibilitando a utilização de conteúdo dinâmico a páginas da Internet. Atualmente, Java é a linguagem de programação orientada a objetos mais utilizada no mundo, desenvolvendo aplicativos corporativos de grande porte, é utilizada em servidores Web, dispositivos móveis etc. (DEITEL, 2010).

IDEs de Desenvolvimento

Um IDE (*Integrated Development Environment* - Ambiente Integrado de Desenvolvimento) é uma ferramenta que contém diversos recursos para facilitar o desenvolvimento de um software (DEVMEDIA, on-line). Com o objetivo principal de

aumentar a produtividade do desenvolvedor, o IDE adiciona diversos atalhos e facilidades, desde a criação de um projeto mais complexo, como a inserção de métodos, até a autogeração de códigos para interfaces gráficas.

Os IDEs mais conhecidos para desenvolvimento orientado a objetos em Java são o NetBeans e o Eclipse. O NetBeans tem diversas ferramentas, modelos e APIs, como *GUI* (*Graphic User Interface* - Interface Gráfica do Usuário), *Builder* Matisse, além de sua comunidade, que, por meio de fóruns, blogs e *ebooks*, disponibiliza uma grande quantidade de materiais (WEXBRIDGE, 2014).

¶ Fique por dentro

Aplicações criadas no NetBeans

Para visualizar quais aplicações foram criadas utilizando a plataforma do NetBeans, basta acessar o site: **netbeans.org <https://netbeans.org/features/platform/showcase.html>** .

São disponíveis para visualização aplicações utilizadas para vigilância, processamento de dados biológicos, saúde, dentre outras.

Instalações necessárias (Java e NetBeans)

Tanto o Java quanto o NetBeans podem ser instalados em Windows, Linux ou OS X (Mac). Na página da Oracle, são disponibilizadas as instalações referentes ao ambiente que está sendo utilizado.

O Java JDK (Java Development Kit), que é o *kit* com o ambiente necessário para desenvolver aplicações em Java, inclui a plataforma do Java SE (Standard Edition), a JRE (Java Runtime Environment) e as ferramentas para desenvolvimento, depuração e monitoramento de aplicações Java.

A JRE é o ambiente de execução para aplicativos desenvolvido com a tecnologia Java. Sem a JRE, nenhum aplicativo Java é possível ser executado (JAVA).

O Java SE é o *kit* de desenvolvimento básico do Java, sendo utilizado como base para diversas aplicações e, também, para os outros *kits* de desenvolvimento do Java, como o Java EE (Enterprise Edition), que é voltado para aplicações *on-line*, fornecendo diversos protocolos e APIs para desenvolvimento em multicamadas com base na Web (JAVA).

¶ Fique por dentro

Como são lançadas novas versões e/ou atualizações do Java, é interessante acessar o site a seguir, que contém as versões para *download* e sua documentação, informando o que foi alterado em um novo lançamento.

Disponível em: **www.oracle.com <http://www.oracle.com/technetwork/java/javase/downloads/index.html>** .

Outra opção é fazer o *download* do NetBeans juntamente com o instalador do JDK, também disponível no site apresentado.

O NetBeans tem várias opções de ambiente de desenvolvimento, variando de acordo com o tipo de aplicação com a qual o programador trabalhará. A opção Java SE oferece as funcionalidades para o desenvolvimento padrão do Java e para JavaFX (linguagem do Java utilizada para aplicações web ricas - RIA). A opção com Java EE oferece as funcionalidades da opção Java SE e mais funcionalidades do Java EE e dos servidores de aplicação Web Glassfish e Apache Tomcat. A opção C/C++ oferece suporte às linguagens C e C++. A opção PHP oferece suporte à linguagem PHP. A opção Tudo contempla todas as opções citadas anteriormente (NETBEANS, on-line). Para o conteúdo abordado, utilizaremos a versão do Java SE.

¶ Fique por dentro

Site para *download* do instalador do NetBeans:

netbeans.org <https://netbeans.org/downloads/> .

É interessante acessar o site anterior para conferir se foram lançadas novas versões do NetBeans, que podem abranger correções de alguns problemas identificados nas versões anteriores.

Aspectos de Utilização do NetBeans

O NetBeans é um IDE de desenvolvimento que poderá ser utilizado com diversas linguagens e com diversos tipos de aplicações (*desktop*, *web*, *webservices*, dentre outros); abordaremos, porém, somente a linguagem Java com foco em aplicações Desktop.

O funcionamento do NetBeans ocorre mediante projetos, ou seja, para uma nova aplicação, deverá ser criado um novo projeto. Caso essa aplicação utilize uma funcionalidade criada por você e será reutilizada em outras aplicações, essa funcionalidade, normalmente, será criada em um projeto independente, para poder ser reutilizada sem ser necessário usar a aplicação inteira.

Os projetos podem ser criados normalmente ou utilizar alguma ferramenta de suporte, nesse caso, o Apache Maven. O Apache Maven, ou simplesmente Maven, é uma ferramenta de automação de *builds* e gerenciamento de projetos disponibilizada pela Apache. O Maven é disponibilizado juntamente com a instalação padrão do NetBeans (DANTAS, 2011).

O Maven utiliza um arquivo XML chamado *Project Object Model* (Modelo de Objeto de Projeto - POM), que descreve tudo o que o projeto necessita para ser executado corretamente, como quais dependências ele utiliza, por exemplo, compilar, e, até mesmo, informações sobre *deploy*.

Os projetos são criados por meio do menu **Arquivo** (File) -> **Novo Projeto** (New Project) -> **Pasta Maven** -> **Aplicação Java** (Java Application). Na janela do Assistente de Novos Projetos, serão mostrados os diversos tipos de projetos que o NetBeans suporta. Dependendo do objetivo da sua aplicação, será escolhido o tipo de projeto, com isso, o esqueleto da aplicação vem por padrão, por exemplo, uma aplicação Web já vem com as pastas necessárias para que a aplicação seja executada corretamente.

Dentro do projeto, há a divisão por pacotes, que, normalmente, são divididos pela funcionalidade. Além de um código mais bem organizado, é uma má prática de programação deixar todas as classes no mesmo pacote ou no pacote raiz (DANTAS, 2011).

A nomenclatura utilizada para pacotes é relativa à empresa que desenvolveu a classe, por exemplo:

br.com.nomeempresa.nomedoprojeto.pacote;

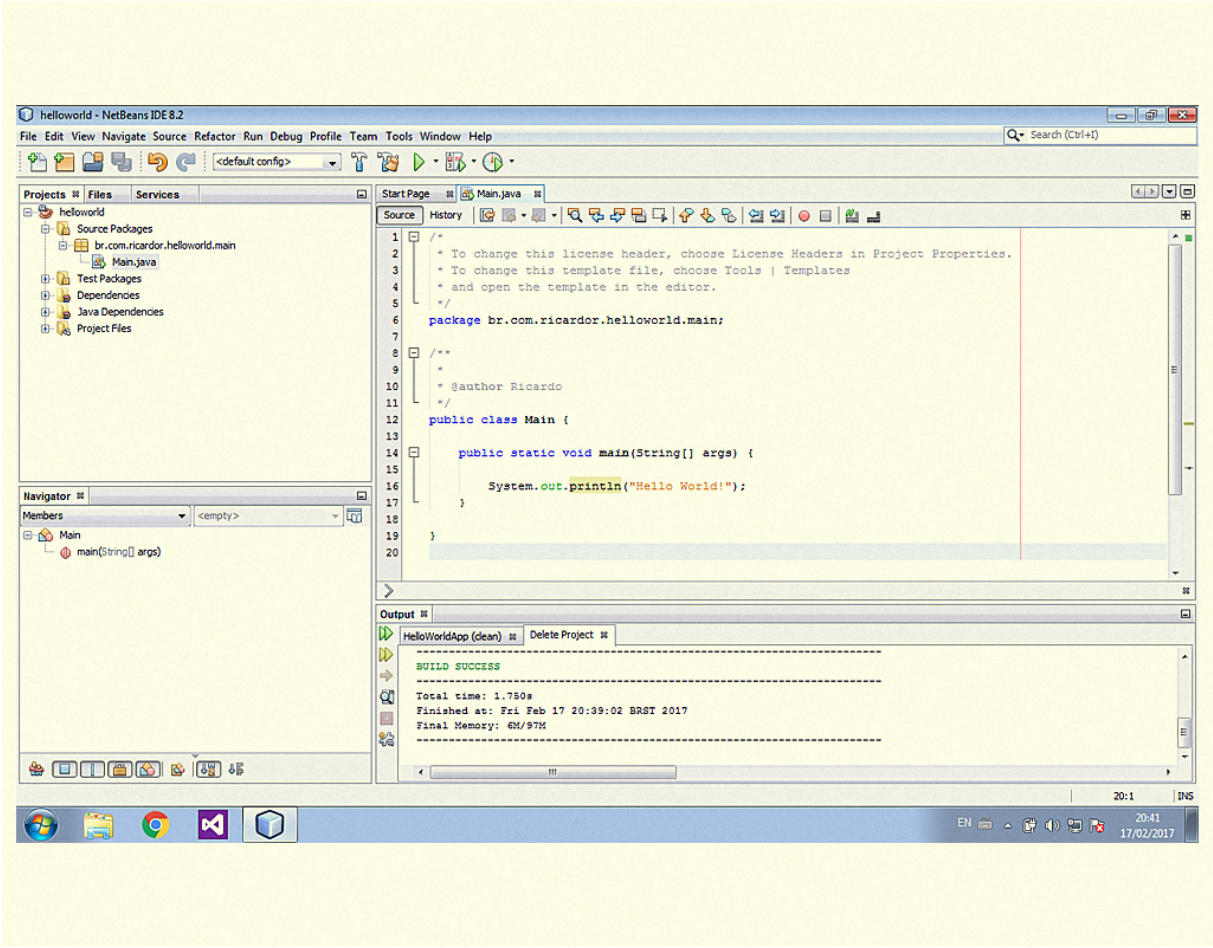
br.com.ricardor.helloworld.util;

Essa divisão da nomenclatura dos pacotes é refletida no local em que o projeto foi salvo, cada nó do nome do pacote é referente a uma pasta. O pacote Util anterior ficará na pasta C:\...\HelloWorld\src\main\java\br\com\ricardor\helloworld\util.

A Figura 1.3 mostra a tela principal do NetBeans para a escrita dos códigos da sua aplicação. À esquerda, temos a lista do(s) projeto(s) aberto(s), com suas classes, bibliotecas e todos os outros arquivos utilizados nesse projeto. No centro, está a janela principal do NetBeans, nela, aparecem todos os arquivos abertos para codificação, que ficam separados em abas na parte superior dessa janela.

A visualização dos projetos no NetBeans ocorre por meio do formato árvore, no qual, no topo, temos o projeto e, abaixo, os arquivos correspondentes a esse projeto, como a pasta com os pacotes do código fonte (Source Packages), que são divididos por módulos da aplicação, por exemplo, um pacote para a interface gráfica, outro pacote para funcionalidades que serão utilizadas na aplicação inteira (Utils), outro pacote para acesso ao dados do banco de dados (DAO), dentre outros, que dependem da divisão do seu projeto.

A criação de um novo pacote é realizada com um clique com o botão direito no pacote que está dentro do nó **Source Packages**, em seguida, seleciona-se **Novo** (New) -> **Pacote Java** (Java Package). Abrirá uma janela para inserir o nome do pacote e se ele estará em algum subnível, por exemplo, br.com.ricardor.helloworld.util.dateutils, que será um pacote que abordará a manipulação de datas.



1FIGURA 3.14 - Tela principal do NetBeans FONTE: NetBeans IDE.

As classes são criadas dentro de determinado pacote, seguindo a divisão por funcionalidade. A criação de uma nova classe segue o princípio dos pacotes, clique com o botão direito no pacote desejado, **Novo** (New) -> **Classe Java** (Java Class) e, na janela do *wizard* de criação de classes, será solicitado o nome da classe e mostrará em qual pacote ela será criada.

Para uma aplicação Java ser executada corretamente, ela deve ter uma classe chamada Main.java e um método main(String[] args), que será invocado quando a aplicação for executada. Os métodos serão explicados mais detalhadamente adiante.

O IDE do NetBeans contempla a funcionalidade de compilar o código no momento em que ele é salvo, ou seja, não é preciso executar um comando (ou botão) específico para compilar um código escrito no NetBeans. Quando o arquivo é salvo, ele é automaticamente compilado no IDE. Contudo, existe a possibilidade de desabilitar o Compilar ao Salvar nas propriedades do projeto.

Para utilizar bibliotecas de terceiro no seu projeto, na janela de visualização em árvore dos arquivos do projeto, há uma pasta chamada ***Libraries*** (Bibliotecas), basta clicar com o botão direito e escolher a opção ***Add JAR/Folder***, quando a biblioteca que será utilizada estiver em formato JAR (que é a forma mais comum de se utilizar bibliotecas de terceiro); caso seja utilizada uma biblioteca criada por você, porém em outro projeto, deve-se escolher a opção ***Add Project***. Com isso, a biblioteca necessária será adicionada ao *classpath* do seu projeto e, assim, conseguirá usufruir de suas funcionalidades sem causar erros de compilação e permitir que o usuário utilize a funcionalidade de autocompletar o código no projeto quando essa biblioteca é referida no meio da aplicação.

Para executar a aplicação, clica-se com o botão direito no nó do projeto e escolhe-se a opção *Clean and Build* (Limpar e Construir), que deleta arquivos compilados anteriormente, recompila o projeto e gera um JAR referente ao projeto. Após esse passo, no menu *Run* (Executar), escolhe-se a opção *Run Project* (Executar Projeto). Caso algum erro ocorra, na janela de Saída, será exibido qual erro foi encontrado durante a execução.

O NetBeans tem a funcionalidade de depuração em tempo de execução para a verificação de erros que podem ocorrer durante a execução da aplicação, e não somente erros de compilação.

Para habilitar a depuração, é necessário criar um *breakpoint*, que é um ponto no qual o IDE executará o sistema até que determinada linha seja executada. O ponto de interrupção é criado clicando no número da linha na janela de edição de código e, assim, a linha ficará sinalizada, mostrando que está marcada para ser interrompida durante a execução.

Quando essa linha for executada, o IDE pausará a execução e trará o código com o ponto de interrupção grifado. Nesse momento, o programador tem a sua disposição todos os dados que estão sendo utilizados no programa durante a execução, como valores de variáveis, endereços de memória, exceções etc. Para executar o projeto em modo de depuração, basta acessar o menu *Debug* (Depuração) -> *Debug Project* (Depurar Projeto).

¶ Fique por dentro

A própria página do NetBeans tem diversas documentações com orientações de como construir aplicações, integrar o projeto com ferramentas externas, criar webservices, dentre outras funcionalidades que o IDE abrange. Essa documentação pode ser acessada no endereço: **netbeans.org <https://netbeans.org/kb/index.html>** .

Introdução à orientação a objetos

Paradigma e Motivação

As atividades do dia a dia podem ter diversos problemas que precisam ser resolvidos em um ambiente computacional. Essa solução pode ser construída de diversas formas, que são chamadas de paradigmas. Jungthon (2010, p. 1) conceitua

paradigma como "o que determina o ponto de vista da realidade e como se atua sobre ela. (...) Cada paradigma determina uma forma particular de abordar os problemas e de formular respectivas soluções".

Os paradigmas mais conhecidos são o **estruturado** e o **orientado a objetos**. O paradigma estruturado é composto de um modelo de entrada, processamento e um resultado de saída, no qual os dados estão separados das funções (SOUZA, 2013). Segundo Gudwin (1997), a Engenharia de Software baseava-se nesse paradigma, que era o mais utilizado até a alta aceitação da orientação a objetos, e tinha como objetivo particionar o projeto de um software em vários módulos, com o intuito de baixar o custo do projeto. Esses módulos tinham como entrada dados que eram processados gerando novos dados como saída.

O paradigma orientado a objetos propõe uma nova visão, na qual o mundo é composto por objetos, e o funcionamento do software acontece pela troca de mensagens e pelo relacionamento entre esses objetos (JUNGTHON, 2010). Os objetos têm métodos, que são os comportamentos que esse objeto terá, e características, que são os atributos.

A motivação de utilizar o paradigma orientado a objetos em relação a outros deve-se ao fato de ele tentar se aproximar o máximo possível do mundo real, com objetos individuais, porém que se comunicam para construir um sistema complexo. A orientação a objetos tem um grande apoio à reutilização e à extensibilidade, principalmente de objetos que compartilham dados em comum, com características de herança; por exemplo, um processo que é utilizado mais de uma vez no sistema implementado em um objeto pode ser reutilizado com outros objetos, diminuindo o retrabalho e a manutenção, pois não será necessário alterar mais de um local com códigos repetidos. A orientação a objetos, contudo, não deve ser utilizada isoladamente, ao contrário, deve ser usada em conjunto com técnicas de Engenharia de Software, para diminuir custos de produção (SOUZA, 2013).

Classes e Objetos

Ao contrário dos paradigmas tradicionais de desenvolvimento, como citado anteriormente, o paradigma orientado a objetos trabalha com um mundo representado por objetos. Horstmann (2004, p. 48) define objeto como "***uma entidade que você pode manipular em seu programa. [...] Você pode pensar no objeto como uma caixa preta que possui uma interface pública e uma implementação oculta***". Por meio dessa definição, podemos concluir que um objeto é algo que pode ser criado para ser utilizado posteriormente no programa, ou, até mesmo, utilizar algum método já existente, com uma implementação oculta, ou seja, não sabemos como o código é escrito dentro desse objeto, mas sabemos o que é necessário para utilizá-lo, que seria a interface.

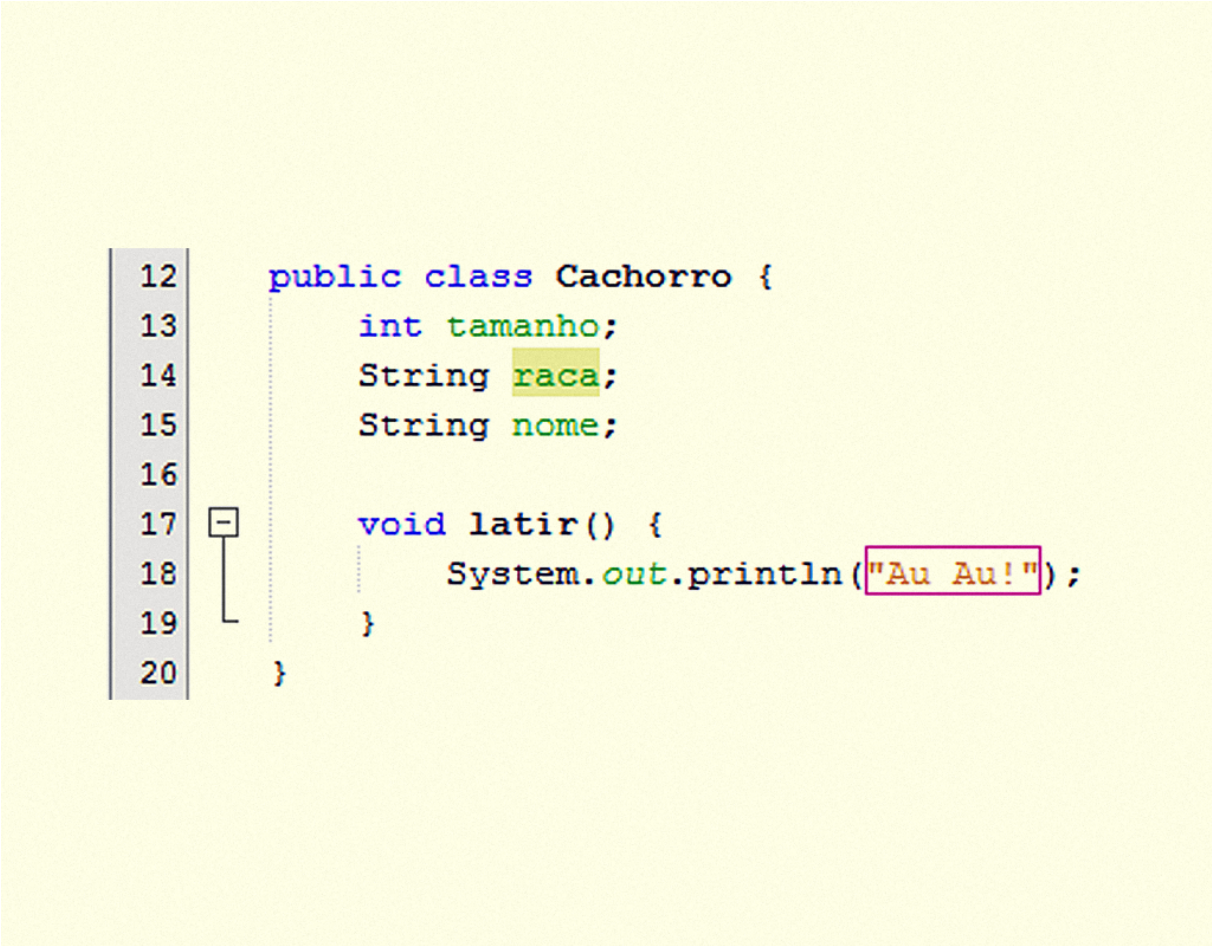
Partindo para um exemplo do mundo real, um **objeto** é algo, qualquer coisa, um Cachorro, por exemplo. Esse objeto, o Cachorro, conhece coisas a respeito de si mesmo; essas coisas que ele conhece são as **variáveis de instância, ou atributos**. Além das coisas que conhece, o Cachorro também faz coisas, essas coisas são chamadas de **métodos** (BATES, 2005).

Quando um objeto for projetado, deverão ser analisadas quais informações ele terá sobre si próprio, o que e como ele deverá trabalhar com esses dados. O objeto Cachorro pode ter os **atributos tamanho,raçae nome**, e o método *latir( )*, que fará o cachorro latir; dependendo da raça e do tamanho, ele irá latir de forma diferente.

Esses objetos são definidos como uma classe, ou seja, a classe é o projeto de um objeto (BATES, 2005). A Figura 1.4 mostra como o objeto Cachorro é definido no formato de uma classe, contendo os atributos tamanho, raça e nome e o método latir. Um objeto não precisa, necessariamente, ter métodos e/ou atributos, o que dependerá de como o sistema será projetado.

1FIGURA 4.14 - Representação gráfica da classe Cachorro FONTE: Bates (2005, p. 26).

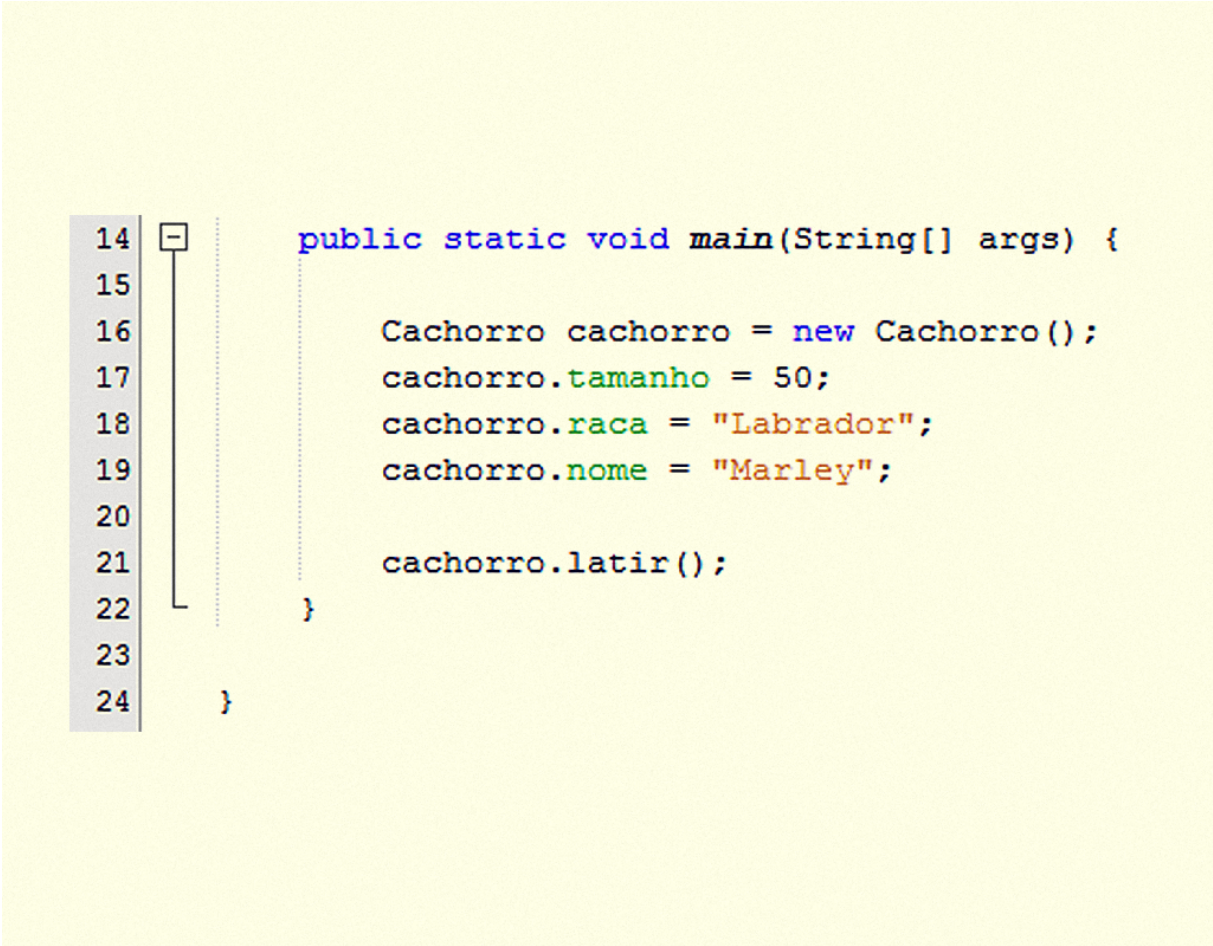
Cada objeto criado a partir dessa classe será um objeto do tipo Cachorro, porém cada um com seus próprios valores, ou características, para cada um dos atributos definidos na classe (BATES, 2005). A Figura 1.5 mostra a definição de uma classe Cachorro.



```
12    public class Cachorro {
13        int tamanho;
14        String raca;
15        String nome;
16
17        void latir() {
18            System.out.println("Au Au!");
19        }
20    }
```

1FIGURA 5.14 - Implementação da classe cachorro FONTE: adaptada de Bates (2005, p. 26).

A utilização dessa classe se dará mediante a criação de objetos do tipo *Cachorro*. O objeto do tipo *cachorro* será instanciado na variável de nome *Cachorro*, terá nome, raça, tamanho e irá latir. A Figura 1.6 apresenta a criação de uma nova instância da classe *Cachorro*, acrescentando os atributos nas linhas 17-19 e invocando o método *latir()* na linha 21.



```
14      public static void main(String[] args) {
15
16          Cachorro cachorro = new Cachorro();
17          cachorro.tamanho = 50;
18          cachorro.raca = "Labrador";
19          cachorro.nome = "Marley";
20
21          cachorro.latir();
22      }
23
24  }
```

1FIGURA 6.14 - Implementação da classe Cachorro FONTE: NetBeans IDE.

Caso quiséssemos um novo Cachorro, criaríamos um novo objeto, **cao** (ou qualquer outro nome diferente do nome que já foi utilizado), e instanciaríamos um novo cachorro por meio da palavra reservada *new* (que instancia um novo objeto, em outro local de memória). Por mais que, nesse novo objeto Cachorro (**cao**), setássemos os mesmos valores para seus atributos, esse seria um outro Cachorro, ou, no caso, um outro objeto do tipo Cachorro, porém com mesmos valores de dados. Se alterássemos os valores do objeto **Cachorro**, os valores setados para o objeto **cao** seriam independentes e não sofreriam alteração.

Um aplicativo Java diz respeito, simplesmente, a objetos se comunicando com outros objetos, utilizando os valores de seus atributos para processar e devolver resultados nas mais diversas formas (BATES, 2005).

A comunicação entre objetos é feita mediante os **métodos**. Os métodos são responsáveis por realizar tarefas pertinentes ao objeto a que eles estão vinculados. Os métodos podem não receber nenhum dado por parâmetro, como o método latir(),ou podem receber algum dado por meio de seus parâmetros. Esses nossos métodos não retornam nenhum resultado, que é definido pela palavra reservada *void* na implementação da classe (na próxima seção, veremos a sintaxe e os comandos básicos do Java); o método, no entanto, pode retornar um resultado, o que dependerá de como o objeto será utilizado no sistema.

Variáveis

As variáveis que foram utilizadas até o momento descrevem uma característica do objeto, que são os atributos (variáveis de instância). Variáveis podem ser, também, declaradas dentro de um método (variáveis locais), podem ser utilizadas como parâmetros de um método ou como tipos de retorno de um método. Elas são utilizadas a partir de sua declaração, que consiste em informar o tipo e o nome da variável. A Figura 1.7 exibe a declaração de uma variável do tipo *int*, que será utilizada pelo nome contador.

1FIGURA 7.14 - Declaração de variável FONTE: NetBeans IDE.

O nome das variáveis devem ser únicos, não sendo possível haver repetição do mesmo nome no escopo em que essa variável está declarada; por exemplo, se uma variável foi declarada como atributo de uma classe, nessa classe, não pode ter outra variável de instância com mesmo nome, porém é permitido utilizar outra variável com mesmo nome dentro de algum método dessa classe.

As variáveis podem ser de tipos primitivos ou referências de objetos. As primitivas são as que contêm valores básicos, como inteiros, booleanos, decimais etc.; as de referência de objetos fazem uma referência a algum objeto que foi instanciado na memória (BATES, 2005).

Os tipos primitivos são: *boolean*, *byte*, *char*, *short*, *int*, *long*, *float* e *double*. As variáveis de tipos primitivos armazenam o valor que foi atribuído a essa variável; caso um outro valor seja atribuído a essa variável, o valor anterior será substituído. As variáveis de instância dos tipos primitivos *byte*, *char*, *short*, *int*, *long*, *double* e *float* são inicializadas automaticamente com o valor 0; já as variáveis de instância do tipo *boolean* são inicializadas como *false*.

As variáveis de referência de objetos armazenam a localização de um objeto na memória, ou seja, elas guardarão a referência para determinado objeto, e não o objeto em si. Esses objetos são classes que foram instanciadas, e essas classes podem ter diversas variáveis de instâncias e métodos. Uma variável de instância é automaticamente inicializada como **null**, que é uma referência a um espaço nulo, ou seja, uma referência a nada (DEITEL, 2010).

Métodos

Um objeto se relaciona com outros e com o sistema por meio de seus métodos. Ou seja, um método é o que o objeto faz. Ao mesmo tempo em que esse objeto tem variáveis de instância que especificam suas características, os métodos realizam as ações desse objeto.

Esses métodos podem ser simples, que não recebem nenhum parâmetro e não retornam nenhum valor, como o método latir(), exibido na linha 17 da Figura 1.5. O método latir() realizará alguma operação referente ao objeto da classe Cachorro, que, nesse exemplo, é apenas imprimir um latido.

Além dos métodos sem nenhum retorno e parâmetro, os métodos podem ter um tipo de retorno e/ou receber valores por parâmetros. Os retornos são informações que um método devolve no momento em que foi invocado. Os parâmetros são variáveis informadas ao método para serem utilizadas durante sua execução.

A Figura 1.8 mostra a declaração de um método que retorna um *boolean*, caso nosso Cachorro tenha conseguido pegar o graveto, que depende do tamanho do graveto que foi arremessado, o que é informado no parâmetro do método. Caso o graveto seja grande e o cachorro seja pequeno, o método retornará *false*, pois o cachorro não conseguiu pegar o graveto.

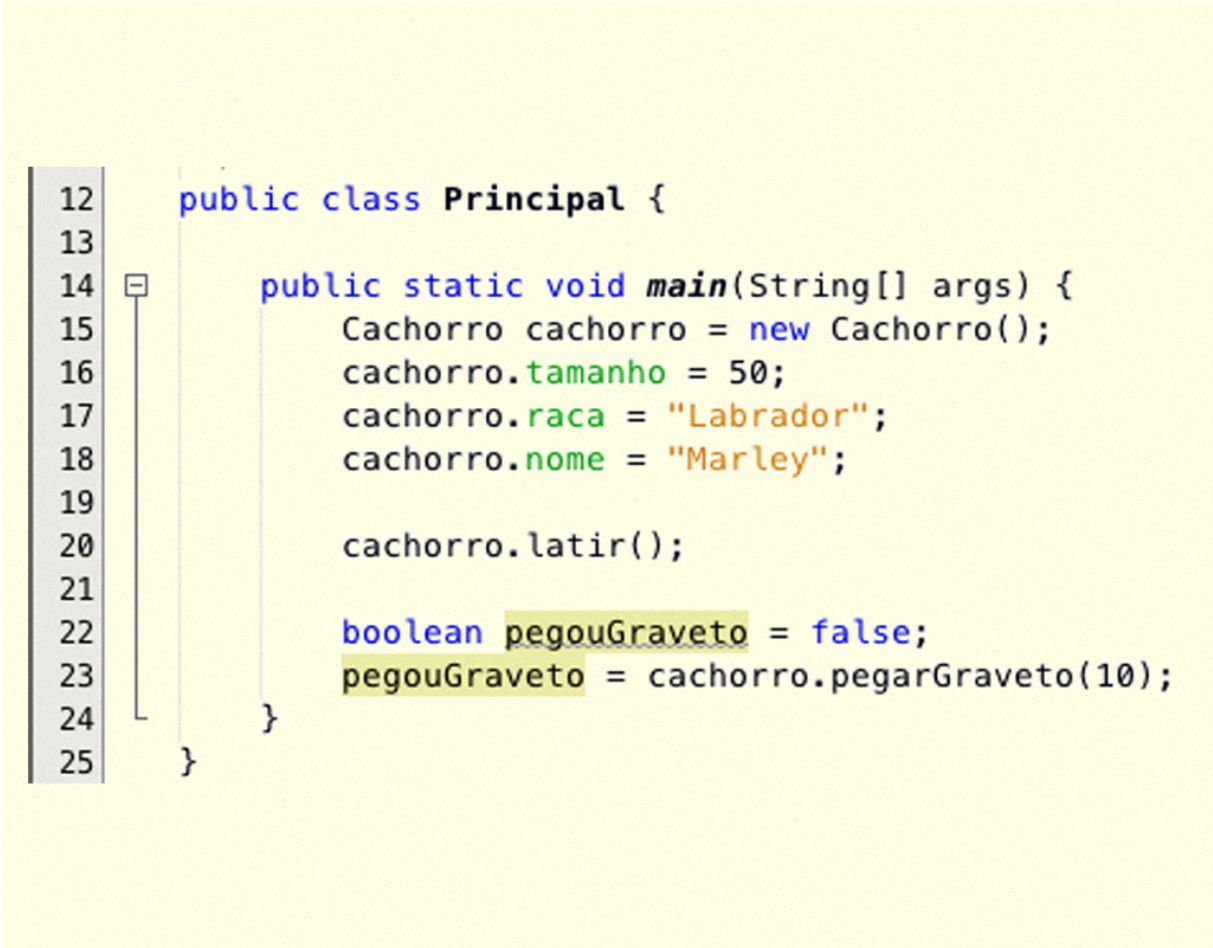
```
12    public class Cachorro {
13        int tamanho;
14        String raca;
15        String nome;
16
17        void latir() {
18            System.out.println("Au au");
19        }
20
21        boolean pegarGraveto(int tamanhoGraveto) {
22            if (tamanho < 10 && tamanhoGraveto > 10)
23                return false;
24            if (tamanho >= 10)
25                return true;
26
27            return true;
28        }
29    }
```

1FIGURA 8.14 - Método com retorno e parâmetro FONTE: NetBeans IDE.

Para utilizar um método que retorne um valor, é necessário atribuí-lo a uma variável no momento da invocação do método. As variáveis que são declaradas e utilizadas dentro de um método são chamadas de variáveis locais, pois o seu escopo é apenas dentro desse método, não sendo possível acessá-las em outro método ou no

corpo da classe. Uma variável local deve ser sempre inicializada, pois pode ocorrer erro de compilação ao utilizar uma variável local que não foi inicializada. Normalmente, é utilizado um valor que não influenciará na execução do sistema, como 0, para variáveis numéricas, *null*, para variáveis de referência, e *string* vazia (""), para variáveis *strings*.

A Figura 1.9 mostra a utilização do método pegar Graveto da Figura 1.8.



```
12    public class Principal {
13
14        public static void main(String[] args) {
15            Cachorro cachorro = new Cachorro();
16            cachorro.tamanho = 50;
17            cachorro.raca = "Labrador";
18            cachorro.nome = "Marley";
19
20            cachorro.latir();
21
22            boolean pegouGraveto = false;
23            pegouGraveto = cachorro.pegarGraveto(10);
24        }
25    }
```

1FIGURA 9.14 - Utilização de método com retorno e parâmetro FONTE: NetBeans IDE

O retorno do método, que é um *boolean* (*true* ou *false*), é atribuído à variável local pegouGraveto na linha 23 da Figura 1.9.

Como os métodos pertencem a objetos, é necessário ter um objeto instanciado para invocar os métodos. Na Figura 1.9, temos a criação de uma instância do objeto *Cachorro* na variável *cachorro na linha 15; nas linhas 20 e 23, temos a invocação dos métodos latir e pegarGraveto, respectivamente, que são definidos na classe Cachorro por meio do operador "ponto" (.).*

Um método pode ser invocado sem instanciar um novo objeto e utilizar o operador "ponto" (.), caso ele seja invocado dentro de um método da própria classe, mas ele será executado somente quando existir uma instância desse objeto. A Figura 1.10 mostra a invocação de um método dentro da sua classe.

Na linha 32 da Figura 1.10, é definido o método cocar() e, então, ele é invocado na linha 22, dentro do método pegarGraveto. O método cocar() será executado somente quando o método pegarGraveto for invocado.

```
12  public class Cachorro {
13      int tamanho;
14      String raca;
15      String nome;
16
17      void latir() {
18          System.out.println("Au au");
19      }
20
21      boolean pegarGraveto(int tamanhoGraveto) {
22          cocar();
23
24          if (tamanho < 10 && tamanhoGraveto > 10)
25              return false;
26          if (tamanho >= 10)
27              return true;
28
29          return true;
30      }
31
32      void cocar() {
33          System.out.println("Estou me coçando");
34      }
35  }
```

1FIGURA 10.14 - Utilização de método dentro da sua classe FONTE: NetBeans IDE.

Sintaxe Java

A sintaxe do Java não se diferencia muito da sintaxe do C e do C++, porém tem algumas particularidades. Nesta seção, veremos a sintaxe dos principais comandos em Java, e a sintaxe complexa pode ser encontrada no site da documentação do próprio Java.

Na lista a seguir, há algumas palavras reservadas (são argumentos que não podem ser utilizados pelo programador como nome de algum objeto, variável etc., dentro de seu programa) da linguagem Java e como são utilizadas.

- *import -> indica quais classes serão utilizadas (as classes que estão no pacote Java.lang não precisam ser importadas explicitamente, ao contrário de outras classes).*
- *class -> indica que está sendo definida uma classe.*
- *private, default, protected, public -> modificadores de acesso de classes, atributos e métodos.*
- *void -> define que não haverá retorno no método.*
- *int, string, float, ... -> definição de tipos de variáveis no Java.*
- *new -> utilizado para instanciar um novo objeto na memória.*
- *Operadores:*
- *variavel++ -> incremento do valor da variável.*
- *variavel-- -> decremento do valor da variável.*
- *== -> operador de comparação de igualdade.*

¶ Fique por dentro

O Java tem inúmeras palavras reservadas e listá-las aqui ocuparia demasiado espaço. É possível verificá-las no site a seguir, que se refere à documentação da linguagem Java.

Disponível em: **docs.oracle.com <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html>** .

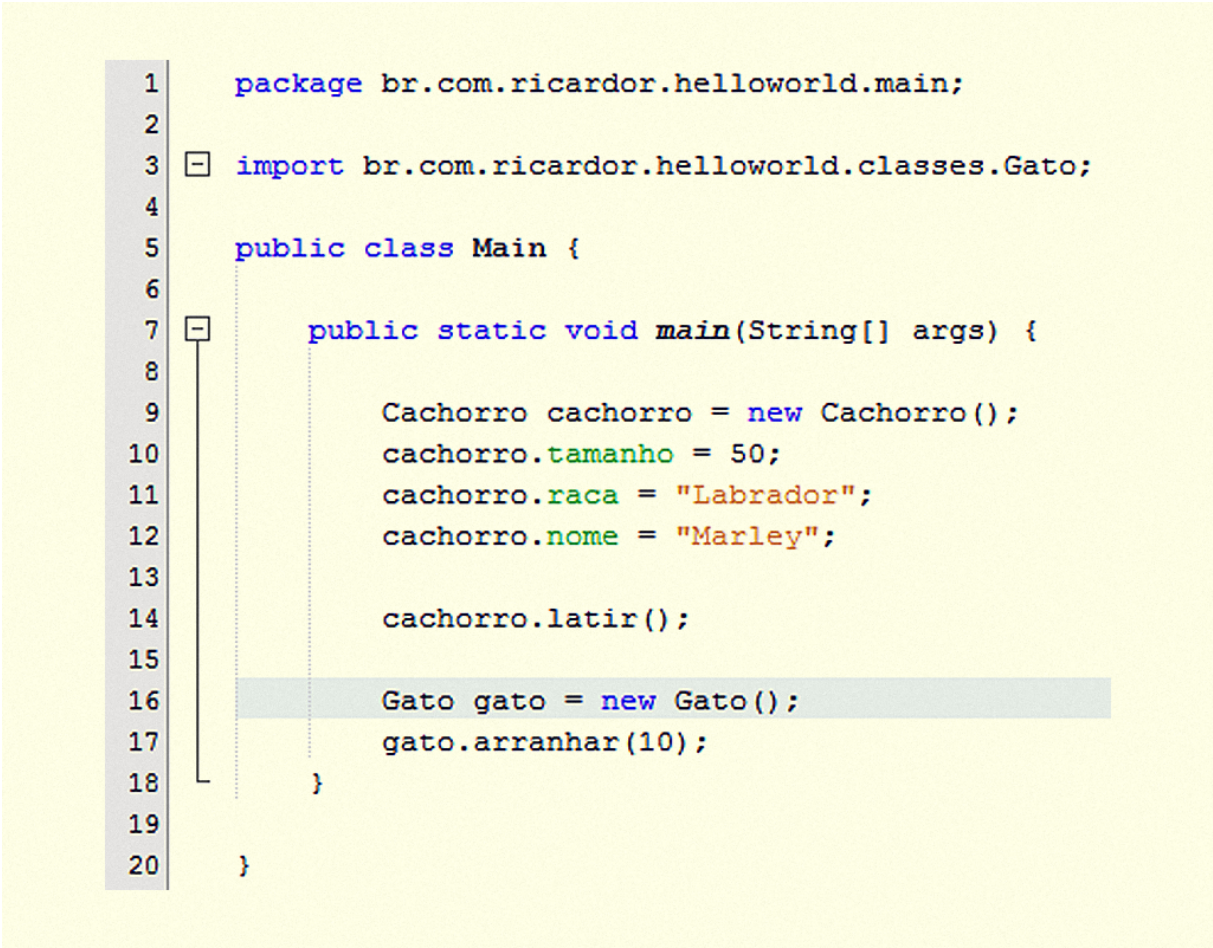
Declaração

No C e no C++, a declaração de variáveis/objetos pode ser feita em qualquer local do código; é uma boa prática de programação, porém, a declaração das variáveis, que serão utilizadas ao longo do código, ser realizada em um bloco somente. Algumas variáveis auxiliares, que serão utilizadas apenas para uma operação (por exemplo, uma variável para receber um valor temporário), podem ser declaradas antes de sua utilização.

A declaração no Java é feita, primeiramente, especificando de que tipo a variável/objeto será e, então, seu identificador (nome). Na Figura 1.11, é possível ver a declaração de um objeto do tipo **Cachorro** e do tipo **Gato**. Essa declaração poderia ser de variáveis de tipos primitivos, mas utilizando o mesmo padrão tipo Nome.

A Figura 1.11 mostra como as classes são utilizadas por meio da criação de uma nova instância de um objeto utilizando o **new** e como utilizar uma classe que está em outro pacote. Na linha 1, mostra-se a qual pacote a classe atual fará parte e, na

linha 3, uma classe de outro pacote é importada para a classe atual por meio do ***import*** *(a classe importada está no pacote br.com.ricardor.helloworld.classes e a classe que está em edição está no pacote br.com.ricardor.helloworld.main).*



```
1   package br.com.ricardor.helloworld.main;
2
3   import br.com.ricardor.helloworld.classes.Gato;
4
5   public class Main {
6
7       public static void main(String[] args) {
8
9           Cachorro cachorro = new Cachorro();
10          cachorro.tamanho = 50;
11          cachorro.raca = "Labrador";
12          cachorro.nome = "Marley";
13
14          cachorro.latir();
15
16          Gato gato = new Gato();
17          gato.arranhar(10);
18      }
19
20  }
```

1FIGURA 11.14 - Instanciando novos objetos FONTE: NetBeans IDE

A classe Gato (que não está exibida aqui) tem um método arranhar (int intensidade), que recebe um parâmetro do tipo inteiro, que determina a intensidade. Os parâmetros dos métodos podem ser de quaisquer tipos, desde primitivos, como int, string, double, byte etc., até objetos criados, como ***cachorro*** *ou* ***gato*** *(ou qualquer outro tipo de objeto).*

Estruturas condicionais

As estruturas condicionais são utilizadas para escolher determinados cursos entre as ações do programa. Caso determinada condição seja satisfeita, o programa se comportará de uma forma, alterando o fluxo de execução; caso contrário, comportar-se-á de outra forma.

As instruções de condição podem ser dos tipos: if, if...else e switch ... case.

Estrutura if, if...else

A estrutura if é utilizada para desviar a execução do código caso uma condição seja satisfeita. A Figura 1.12 exibe a sintaxe da estrutura if ... else. O que difere as duas estruturas é que a estrutura com o else impõe mais condições que deverão ser satisfeitas. Caso a condição do if não seja satisfeita, a execução do código poderá entrar no bloco de código do else, ou, então, o else permite o teste de mais condições até alguma ser satisfeita; se nenhuma condição for satisfeita, o fluxo de execução não entrará no bloco de código.

A condição sempre estará entre parênteses após o if, e o resultado do teste será um booleano true ou false. True satisfará a condição e entrará no bloco, false desviará a execução, e o bloco posterior ao else será executado.

A Figura 1.12 exibe dois tipos de comparação com uma String na cláusula if, uma utilizando o método ***equals*** *e outra utilizando a operação de igualdade ==. A diferença é que o método equals() verifica se o conteúdo da string (no exemplo, cachorro.raca) é igual; já o operador == compara se a referência é igual, comparando os bits das variáveis. No primeiro if, foi utilizado o método equals a partir do texto "Labrador", pois esse texto é considerado uma string e, utilizado dessa forma, garante que não ocorrerá nenhum erro, caso o objeto cachorro não seja instanciado, apenas retornará false para a estrutura if. Se for utilizado o método equals a partir do objeto cachorro (cachorro.raca.equals("Labrador")), no exemplo,*

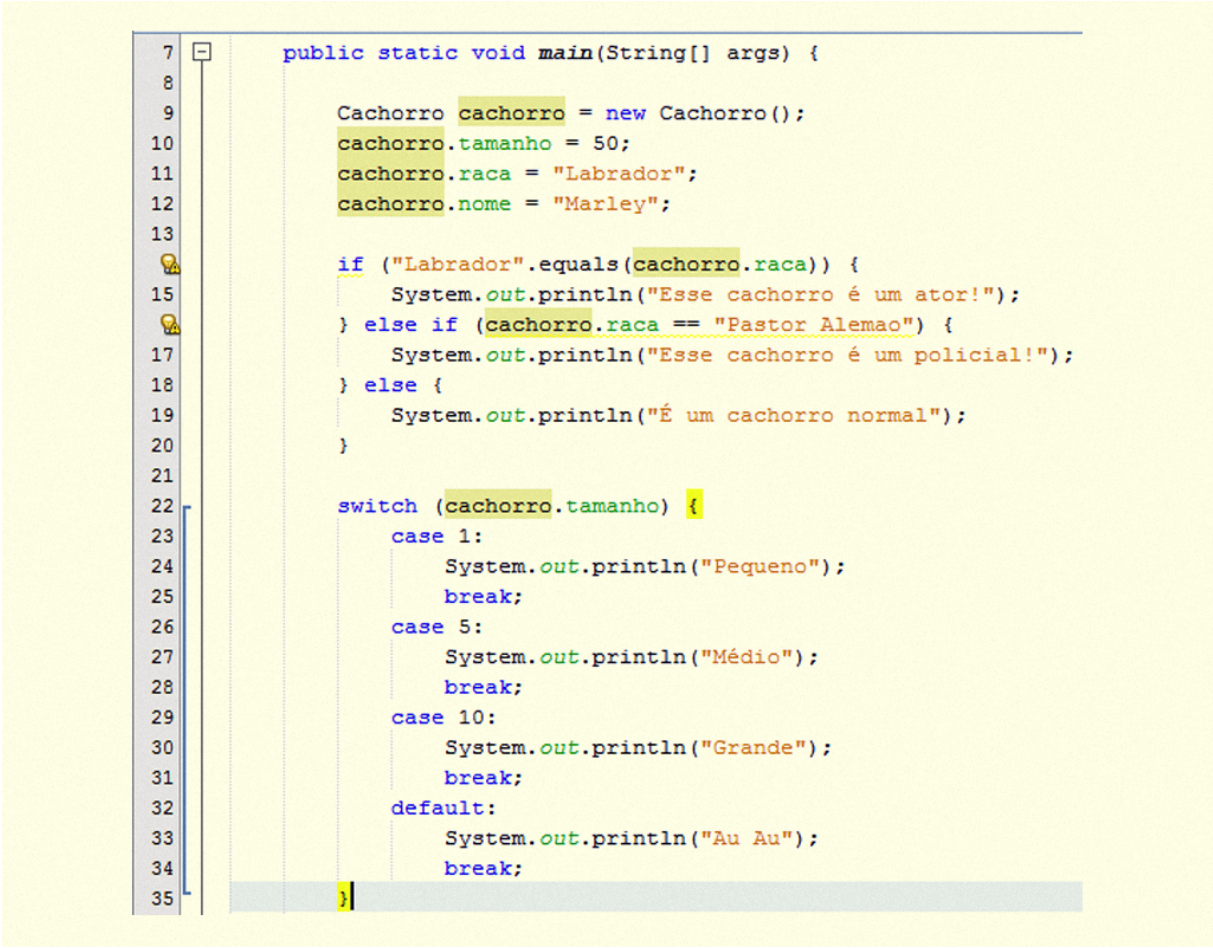
funcionaria, pois o objeto cachorro foi instanciado anteriormente por meio do operador new; caso não fosse instanciado, ocorreria um erro de o objeto ser nulo e não ter nenhuma instância.

Estrutura switch

A estrutura switch é uma forma de definir diversos desvios no código, dependendo, apenas, de uma variável ou expressão. Ele pode ser utilizado quando uma variável pode assumir diversos valores possíveis com um tratamento específico em cada um deles.

Todas as declarações do case devem ser de um mesmo tipo, que é o mesmo tipo da variável testada, e não podem haver declarações repetidas. Caso a variável assuma um valor que não é considerado nas declarações case, pode ser utilizada a palavra default para tratar esse caso específico.

A Figura 1.12 mostra como é utilizada a estrutura switch, usando o atributo tamanho da classe cachorro para comparação. O atributo tamanho é do tipo inteiro, portanto, as declarações do case são comparadas com numerais.



```
7   public static void main(String[] args) {
8
9       Cachorro cachorro = new Cachorro();
10      cachorro.tamanho = 50;
11      cachorro.raca = "Labrador";
12      cachorro.nome = "Marley";
13
14      if ("Labrador".equals(cachorro.raca)) {
15          System.out.println("Esse cachorro é um ator!");
16      } else if (cachorro.raca == "Pastor Alemao") {
17          System.out.println("Esse cachorro é um policial!");
18      } else {
19          System.out.println("É um cachorro normal");
20      }
21
22      switch (cachorro.tamanho) {
23          case 1:
24              System.out.println("Pequeno");
25              break;
26          case 5:
27              System.out.println("Médio");
28              break;
29          case 10:
30              System.out.println("Grande");
31              break;
32          default:
33              System.out.println("Au Au");
34              break;
35      }
```

1FIGURA 12.14 - Estruturas de decisão FONTE: NetBeans IDE.

Estruturas de Repetição

As estruturas de repetição são instruções que podem fazer blocos de comandos serem executados diversas vezes repetidamente, enquanto uma condição for satisfeita; são conhecidas como loops (laços) (FILGUEIRAS, 2015).

Para ser executada, a estrutura de repetição depende de que uma condição seja satisfeita e, após cada iteração do laço, a condição volta a ser testada novamente, executando o bloco de comando até que a condição falhe.

O Java tem dois tipos principais de estruturas de repetição: while e for, cada um apresentando uma variação da sua utilização.

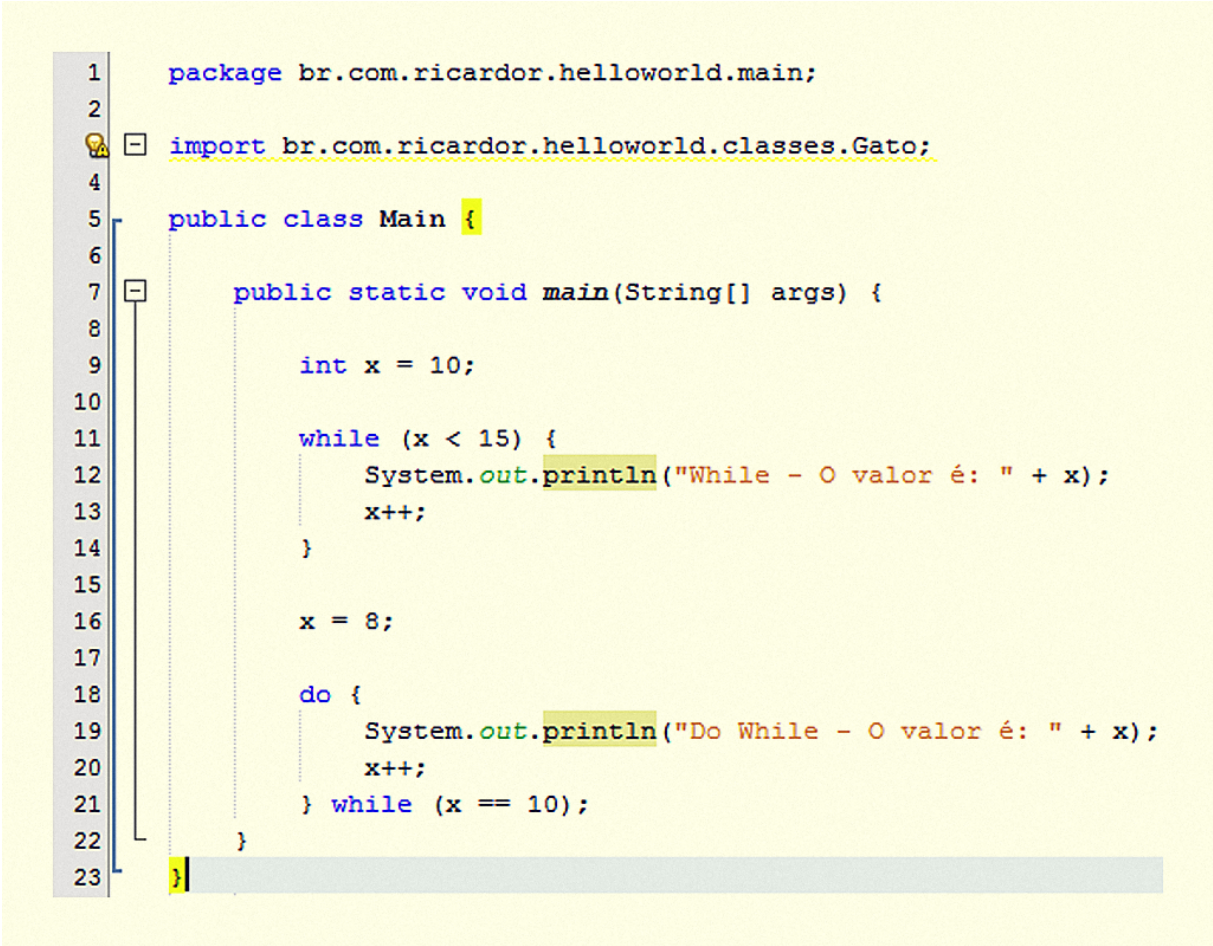
While

O termo while, traduzido para o português como "enquanto", é utilizado para realizar repetições enquanto uma condição for satisfeita.

O estado inicial dos elementos que serão utilizados no laço deverá ser inicializado antes da execução do while, pois esses elementos serão testados para que o laço ocorra (FILGUEIRAS, 2015).

Uma variação da estrutura while é a estrutura do - while, que se comporta inversamente ao while no quesito de comparação de condição. Enquanto o while testa a condição antes de entrar nos laços, o do - while, primeiro, realiza uma iteração, para, então, testar a condição. Mesmo que a condição inicial seja falsa, as instruções do laço serão executadas uma vez.

A Figura 1.13 exemplifica a utilização das estruturas while e do - while comparando um número. Na execução da estrutura do - while, vemos que o comando do laço é executado apenas uma vez, pois o valor da variável **x** *não é igual (==) a 10, então, a condição é testada e o fluxo de execução não repete as instruções do laço.*



```
1   package br.com.ricardor.helloworld.main;
2
3   import br.com.ricardor.helloworld.classes.Gato;
4
5   public class Main {
6
7       public static void main(String[] args) {
8
9           int x = 10;
10
11          while (x < 15) {
12              System.out.println("While - O valor é: " + x);
13              x++;
14          }
15
16          x = 8;
17
18          do {
19              System.out.println("Do While - O valor é: " + x);
20              x++;
21          } while (x == 10);
22      }
23  }
```

1FIGURA 13.14 - Estrutura de repetição while e do - while FONTE: NetBeans IDE.

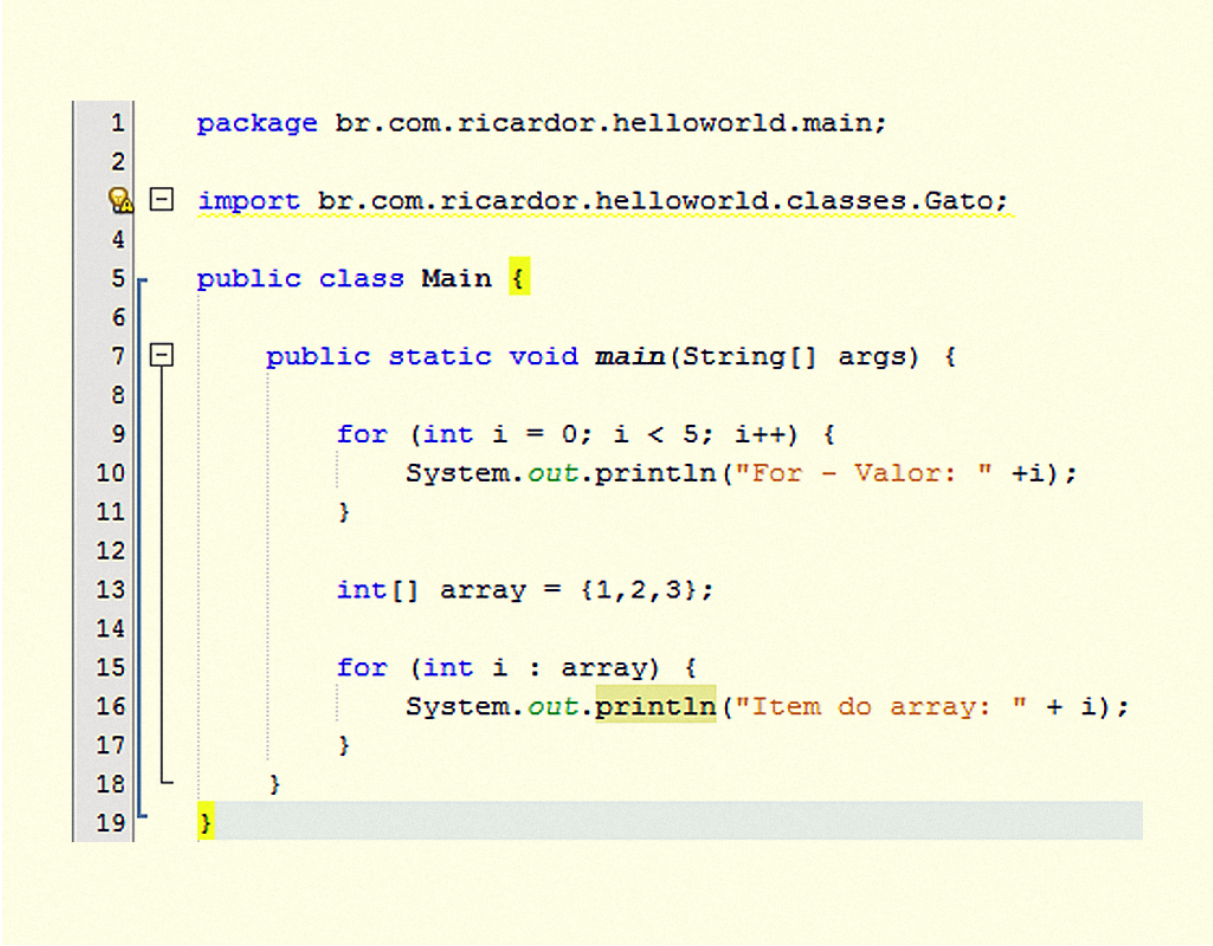
For

A estrutura de repetição for tem suas condições de inicialização, repetição e iteração (incremento ou decremento) diretamente no cabeçalho de definição do for e, após, é implementado o bloco de código referente aos laços. As condições de inicialização, repetição e iteração são separadas por ponto e vírgula (;).

O for e o while são duas formas diferentes de estruturas de repetição similares, já que um laço for pode ser escrito na forma de while e vice-versa (FILGUEIRAS, 2015).

Assim como o while, o for tem uma variação que é o Enhanced-For. O Enhanced-For foi introduzido no Java 5, com o intuito de percorrer collections (collections são estruturas de dados como arrays, LinkedList, dentre outros). Em cada iteração do

Enhanced-For, um item do array é atribuído automaticamente à variável declarada no cabeçalho do laço (FILGUEIRAS, 2015). Os laços For e Enhanced-For são mostrados na Figura 1.14.



```
1   package br.com.ricardor.helloworld.main;
2
3   import br.com.ricardor.helloworld.classes.Gato;
4
5   public class Main {
6
7       public static void main(String[] args) {
8
9           for (int i = 0; i < 5; i++) {
10              System.out.println("For - Valor: " +i);
11          }
12
13          int[] array = {1,2,3};
14
15          for (int i : array) {
16              System.out.println("Item do array: " + i);
17          }
18      }
19  }
```

1FIGURA 14.14 - Estrutura de repetição For e Enhanced-For FONTE: NetBeans IDE.

Na próxima unidade, serão abordadas características específicas da Orientação a Objetos, como herança, polimorfismo, tratamento de exceções, sobrecarga de métodos, dentre outras.

Indicação de leitura

Nome do livro:: *Java Como Programar*

Editora:: *Pearson Prentice Hall*

Autor:: *Harvey Deitel e Paul Deitel*

ISBN:: *9788576055631*

Comentário: Esse livro aborda uma grande quantidade de conteúdo voltada para a orientação a objetos, desde suas definições e conceitos até exemplos práticos utilizando diversas funcionalidades do Java.

Indicação de leitura

Nome do livro:: *Use a Cabeça! Java*

Editora:: *O'Reilly - Alta Books*

Autor:: *Bert Bates e Kathy Sierra*

ISBN:: *857608084-2*

Comentário: Um livro muito interessante para quem está iniciando no Java, com exemplos práticos e bastante simples de assimilar. Além de explicar assuntos básicos, como sintaxe do Java, utiliza uma forma bem lúdica para explicar e exemplificar a orientação a objetos.

UNIDADE II

Princípios da Orientação a Objetos

Ricardo de Almeida Rocha

Esta unidade introduzirá os principais conceitos utilizados na Orientação a Objetos, que são herança e polimorfismo, os quais abrangem sobrecarga de métodos, sobreposição de métodos, interfaces e outras características essenciais para a orientação a objetos. Além disso, a unidade abordará como estruturar um projeto Java na IDE NetBeans, por meio de pacotes e classes, e, também, uma introdução ao tratamento de exceções, buscando resolver os problemas que podem ocasionar uma falha de execução de uma aplicação.

Pacotes, classes, métodos e suas propriedades

Os objetos são escritos por meio de classes, que descrevem a estrutura, projetando o objeto com seus atributos e métodos (BATES, 2005). Os métodos especificam o comportamento desse objeto, como ele se relacionará com outros objetos, e os atributos especificam as características do objeto.

Nesta seção, veremos como uma classe é estruturada dentro de pacotes, com seu(s) construtor(es), métodos e métodos acessores.

Pacotes

Como citado na seção 2.2 da Unidade I, toda classe Java faz parte de um pacote. O objetivo principal de um pacote é manter o código organizado, separando as classes por pacotes com alguma característica em comum. O pacote também pode especificar qual entidade é a "dona" do código mediante a nomenclatura reversa do domínio.

A utilização do domínio completo para nomear um pacote deve-se ao fato de um domínio ser único, ou seja, ele só pode ter um único dono e o nome do domínio não pode ser repetido em outro domínio, isto é, o domínio abc.com.br será único e diferente do domínio abc.com. O uso do domínio completo garante que não haverá outros nomes coincidindo com os nomes do seu projeto. Por outro lado, se a utilização fosse somente o nome simples do domínio (somente abc) e a ABC dos Estados

Unidos utilizasse ABC nos seus projetos, os nomes iriam coincidir, podendo causar problemas, por isso é convencionado o nome do domínio reverso (br.com.abc) para o prefixo dos nomes dos pacotes.

Outro objetivo do pacote é evitar colisões de nomes de classes diferentes, ou seja, supõe-se que exista a classe Pessoa criada pela organização ABC e outra classe, com mesmo nome, criada pela organização DEF, porém com códigos e finalidades diferentes que podem ser utilizados por um desenvolvedor que não está vinculado a nenhuma, mas que utilizará bibliotecas de terceiro em seu projeto. Caso as classes não estivessem especificadas com o domínio, haveria duas classes Pessoa com funcionalidades diferentes (STAFUSA, 2015).

A possibilidade de ter diversos projetos no mesmo domínio implica na utilização do nome do domínio reverso, sendo especificado o nome do projeto após o domínio, por exemplo, br.com.abc.projetol e br.com.abc.projeto2, tornando mais fácil de identificar quais são os projetos e para quais domínios eles pertencem; assim os projetos ficam equivalentes à hierarquia de pastas (no explorador de arquivos do sistema operacional em uso, as pastas físicas do projeto ficariam br/com/abc/projetol, caso não fosse utilizada a nomenclatura reversa do domínio, as pastas ficariam projetol/abc/com/br, podendo ocorrer colisão de nomes e gerar diversos problemas) (STAFUSA, 2015).

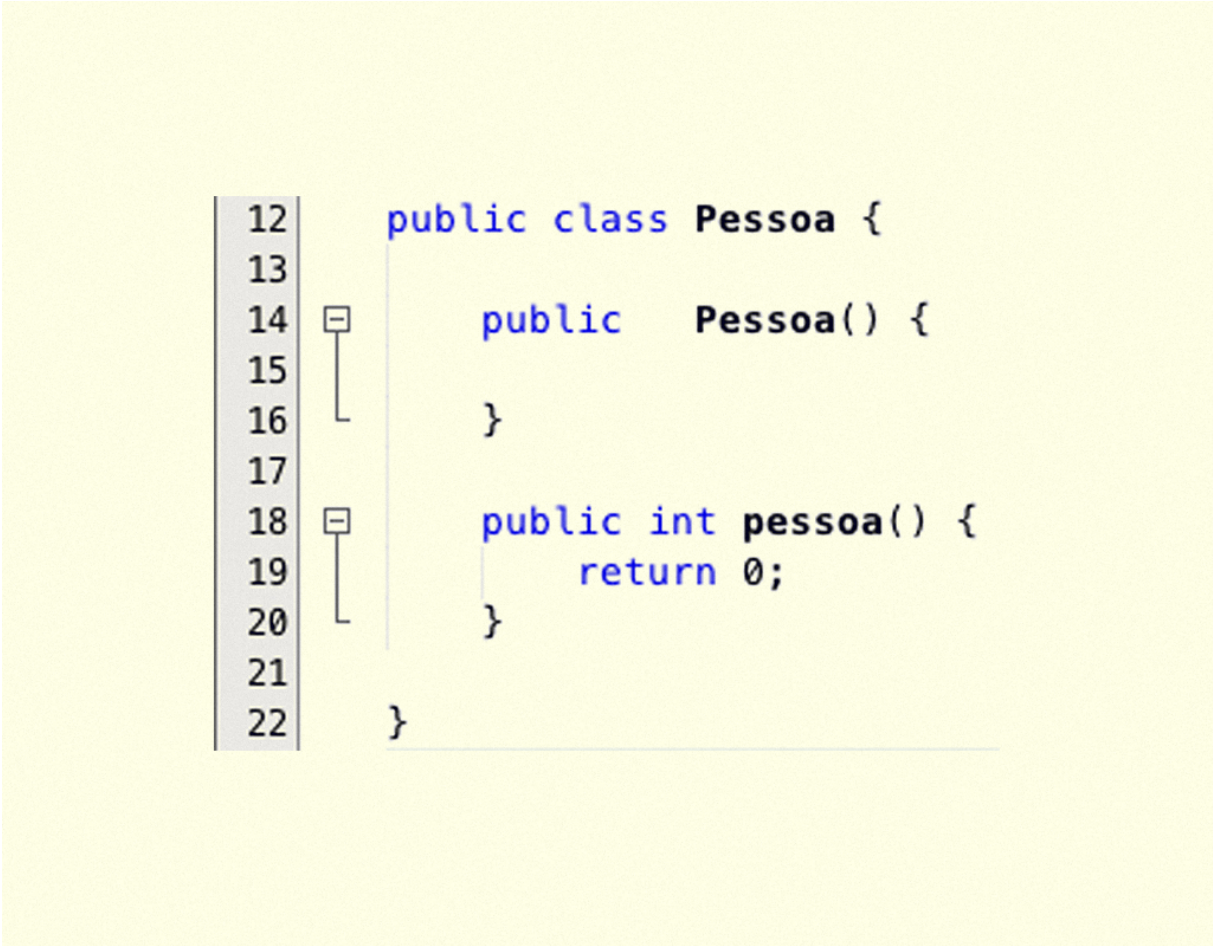
Dentro de um projeto, cada classe é criada dentro de pacotes com funcionalidades similares, por exemplo, as classes responsáveis pela interface gráfica de um projeto serão alocadas no pacote br.com.abc.projeto.GUI (GUI - *Graphic User Interface* - Interface Gráfica de Usuário), enquanto isso, as classes responsáveis por realizar acessos a dados do banco de dados poderão ser alocadas no pacote br.com.abc.projeto.DAO, melhorando a organização dos códigos-fonte do projeto.

Construtores

Os construtores são métodos que são invocados quando é criada uma nova instância da classe na memória. Segundo Barnes (2004, p. 49), "***os construtores permitem que cada objeto seja configurado adequadamente quando ele é criado***".

Um construtor é muito parecido com um método, porém não é definido especificamente como um método. Os métodos são executados por meio de uma instância já existente de um objeto, em contrapartida o construtor é executado no momento de criar um novo objeto (BATES, 2005).

O construtor é chamado pela palavra reservada **new** e, em seguida, o nome da classe. No construtor, contém o código que será executado quando o objeto for instanciado. Um construtor tem o mesmo nome da classe em que ele está alocado, sem ter definido nenhum tipo de retorno, e poderá ter quantos parâmetros forem necessários. A Figura 2.1 mostra a sintaxe de um construtor simples, sem parâmetros, para observar a diferença entre construtores e métodos.



2FIGURA 1.23 - Construtor e método FONTE: NetBeans IDE

Na classe Pessoa, há o construtor Pessoa(), que não tem nenhum retorno, e o método int pessoa(), que retorna um inteiro. Esse método foi criado apenas para exemplificar a diferença entre um método e um construtor, pois o método não está em uma nomenclatura clara, já que o nome do método deve ser autoexplicativo.

O construtor é executado antes de o objeto ser atribuído a uma referência de memória, ou seja, a função do construtor é fazer as operações necessárias para deixar o objeto pronto para uso, seja inicializar atributos do objeto com valores passados por parâmetro do construtor, ou, até mesmo, instanciar outros objetos que serão utilizados dentro dessa classe que está sendo criada (BATES, 2005).

Uma classe pode ter quantos construtores forem necessários, porém a assinatura de cada construtor deve ser diferente uma da outra, isto é, não é possível ter dois construtores que tenham um parâmetro do mesmo tipo; exemplificando: se a classe Pessoa tiver o construtor public Pessoa(int idade) definido, ela não poderá ter um outro construtor public Pessoa(int codigo), pois ambos recebem um parâmetro do tipo *int*. Se a classe não tiver nenhum construtor definido, o compilador criará automaticamente um construtor sem argumentos, apenas para instanciar o objeto. Contudo, se a classe tiver **ao menos** um construtor com argumentos, mas não tiver o construtor sem argumentos, o compilador **não** criará esse construtor. É possível definir um construtor sem argumentos e um construtor (ou vários) com argumentos, para alguma situação em que não seja conhecido qual o argumento a ser utilizado na instanciação do objeto (BATES, 2005).

Variáveis

O objeto apresenta valores que definem seus estados (atributos), que são as variáveis de instância desse objeto, e os métodos, que definem comportamentos para esse objeto. As variáveis de instância são as declaradas fora dos métodos e são específicas desse objeto; quando essas variáveis têm seus valores alterados, dizemos que o objeto teve seu estado alterado (BATES, 2005). As variáveis de instância estão ligadas ao objeto que foi instanciado por meio do new, ou seja, cada objeto instanciado será diferente um do outro e as variáveis serão específicas para cada objeto.

Além das variáveis de instância, as classes têm as variáveis locais, as quais foram declaradas dentro de um método, e elas podem ser utilizadas no escopo desse método. Fora dele, não é possível acessá-las.

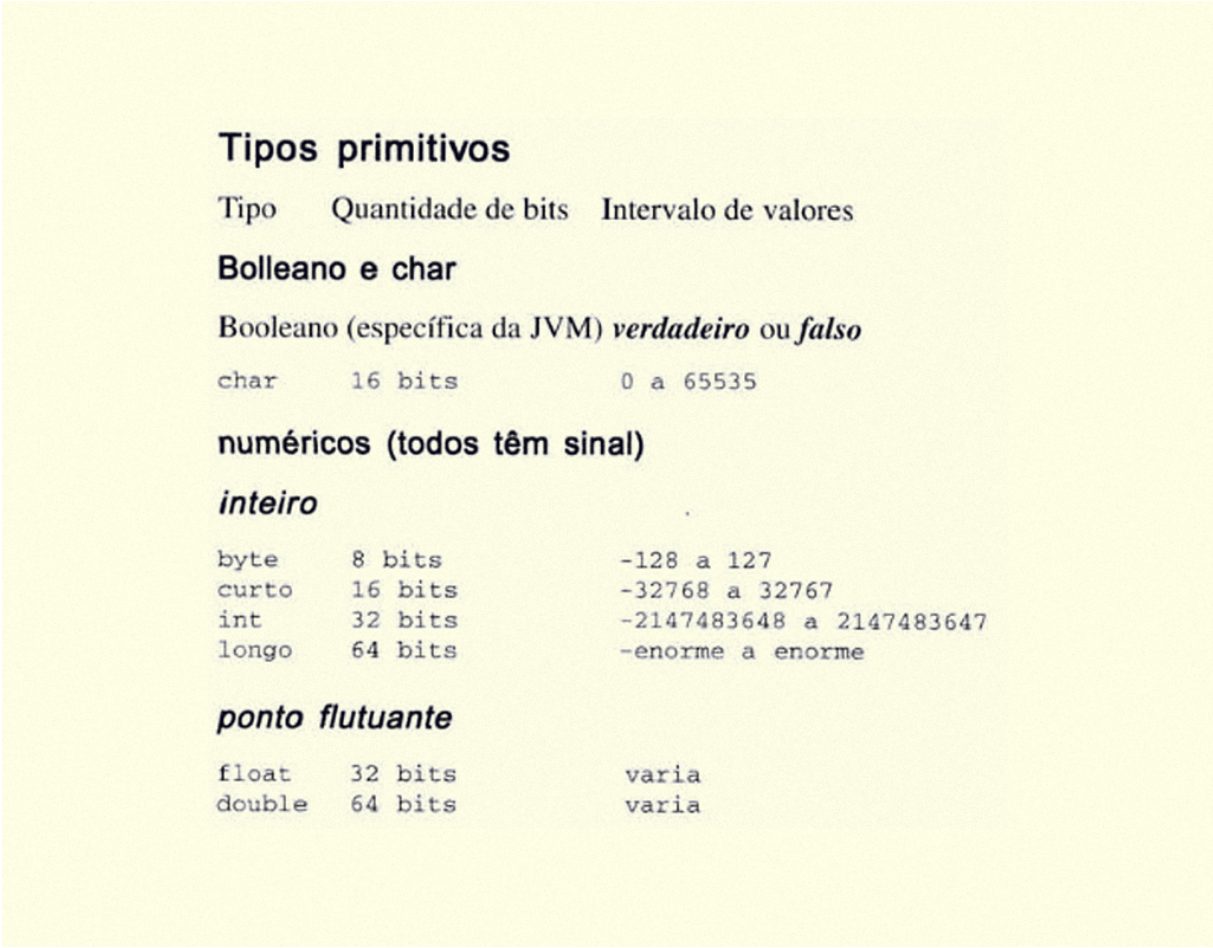
Tanto as variáveis de instância quanto as locais podem ser declaradas como tipos primitivos, que são variáveis que guardam valores em si. Esses tipos primitivos podem ser *byte*, *char*, *short*, *int*, *long*, *float*, *double* e *boolean*. A Figura 2.2 mostra a

quantidade de *bits* e os valores que os tipos primitivos suportam.

Ao acessar uma variável que foi declarada como tipo primitivo, é acessado diretamente o valor que foi atribuído a essa variável, tanto na inicialização quanto na codificação do sistema.

As variáveis locais e de instância podem ser declaradas também como variáveis de referência. Ao contrário do tipo primitivo, essas variáveis não guardam valores, mas sim referência para um local de memória em que está armazenado um objeto. Esse objeto pode ser qualquer um que foi criado durante a execução do sistema.

A Figura 2.3 representa como é realizada a referência para um objeto. A variável myDog, que é declarada como o tipo complexo Dog, aponta para o local da memória em que o objeto dog está armazenado, ou seja, a variável myDog guardará os *bits* do endereço da memória.



Tipos primitivos

Tipo	Quantidade de bits	Intervalo de valores
Bolleano e char		
Booleano (específica da JVM)	***verdadeiro*** ou ***falso***	
char	16 bits	0 a 65535
numéricos (todos têm sinal)		
inteiro		
byte	8 bits	-128 a 127
curto	16 bits	-32768 a 32767
int	32 bits	-2147483648 a 2147483647
longo	64 bits	-enorme a enorme
ponto flutuante		
float	32 bits	varia
double	64 bits	varia

2FIGURA 2.23 - Tipos primitivos e seus valores FONTE: Bates (2005, p. 37).

2FIGURA 3.23 - Variável de referência FONTE: Bates (2005, p. 40).

Argumentos

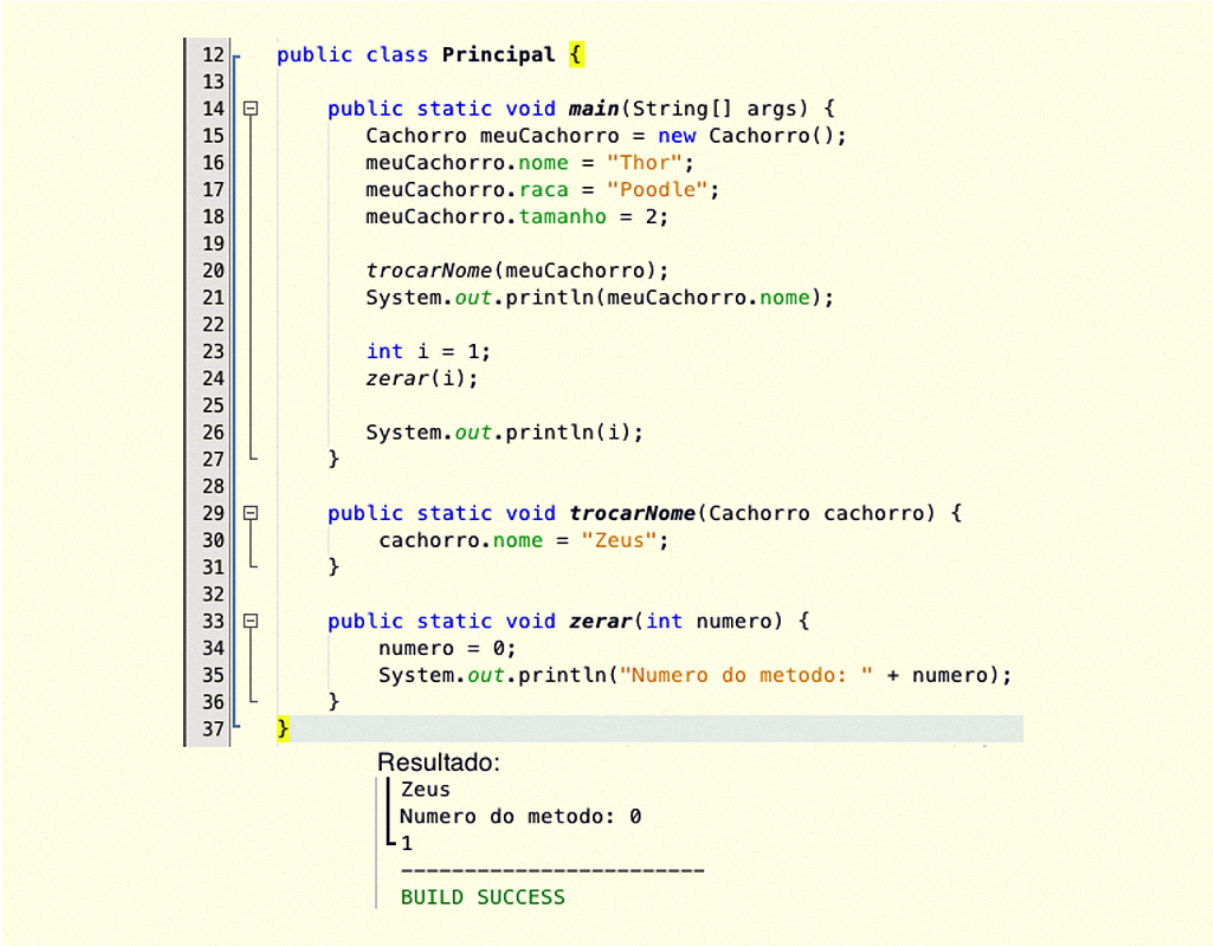
Os argumentos (parâmetros) são valores que são passados aos métodos para serem utilizados dentro de seu bloco de comando. Os argumentos, ou parâmetros, são variáveis locais (que atuarão dentro do escopo do método) com valores (BATES, 2005).

Os parâmetros, que são utilizados tanto em métodos quanto em construtores, como citado na seção anterior, são passados sempre por valor. Nos parâmetros por valor, é feita uma cópia do valor que está fora do método e utilizada dentro do método, o valor da variável que é parâmetro pode ser alterado dentro do método, mas fora essa variável terá o mesmo valor de quando o método foi invocado.

O principal motivo para a passagem de parâmetros no Java ser feita somente por valor é reduzir erros de referência a uma variável nula. Caso uma variável fosse passada por referência para um método e, nesse método, ela fosse "destruída" (deixando a sua referência na memória como nula), após a execução do método, caso a variável fosse utilizada, resultaria em um erro de apontamento a um endereço nulo (*NullPointerException*).

Quando um objeto *p* é passado pelo parâmetro *pt* em um método, é realizada uma cópia da referência do objeto *p* para a variável *pt* e utilizada dentro do método. Portanto, os atributos que forem modificados no objeto serão alterados no objeto que foi passado no parâmetro do método, pois a cópia da referência dentro desse método foi alterada, alterando, assim, o valor original da referência. Contudo, isso não acontece com variáveis de tipos primitivos que são passadas por parâmetro, pois é feita uma cópia do seu valor para uma variável local (do método).

A Figura 2.4 mostra o comportamento da passagem de parâmetros das variáveis de referência e primitivas. As linhas 15 - 18 declaram e atribuem os valores da variável meuCachorro, que é um objeto do tipo Cachorro. Na linha 20, esse objeto é passado por parâmetro para o método trocarNome (definido na linha 29), que atribui um outro valor para a variável de instância nome do objeto meuCachorro. O resultado exibido na linha 21 é o valor atribuído dentro do método. Já as variáveis de tipos primitivos se comportam inversamente, como é demonstrado nas linhas 23-26 e 33-35. Nas linhas 23-26, é declarado e atribuído o valor 1 para uma variável int, invocado um método para zerar seu valor e, então, é impresso o valor. Nas linhas 33-35, o método imprime o valor que foi atribuído para a variável local. As saídas são exibidas abaixo do código, verificando que o valor da variável não é alterado quando a execução do método termina.



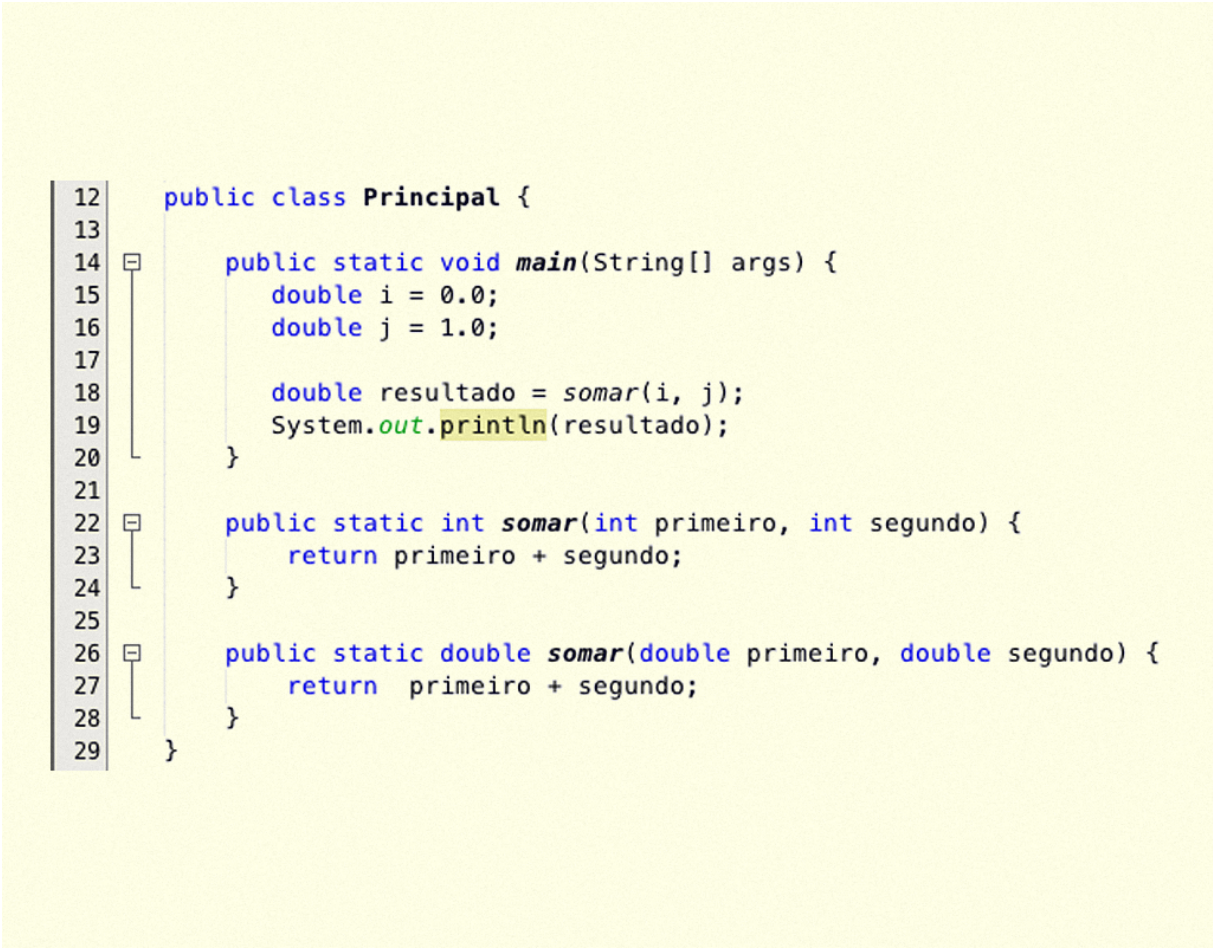
2FIGURA 4.23 - Passagem de parâmetro de variáveis FONTE: NetBeans IDE.

Sobrecarga

A sobrecarga de um método é, simplesmente, dois ou mais métodos com mesmo nome na mesma classe, ou em classes com relacionamentos por meio da herança, mas com a lista de argumentos diferentes. A sobrecarga permite criar várias versões de um método com listas de argumentos diferentes (BATES, 2005).

O ponto forte de sobrecarregar um método é a flexibilidade que pode ser dada ao utilizador mediante métodos que realizam a mesma tarefa, ou tarefas semelhantes, com tipos ou números diferentes de argumentos (BATES, 2005). A Figura 2.5 mostra um exemplo de sobrecarga de métodos, no qual o método somar tem a primeira declaração com argumentos do tipo int, e uma sobrecarga com argumentos double. Nesse caso, a sobrecarga é interessante, pois, caso o código tenha duas variáveis do

tipo double, não é necessário convertê-las para int, e sim invocar o método que tem argumentos do tipo double, como é exibido nas linhas 15, 16 e 18. O método somar é invocado com as variáveis do tipo double.

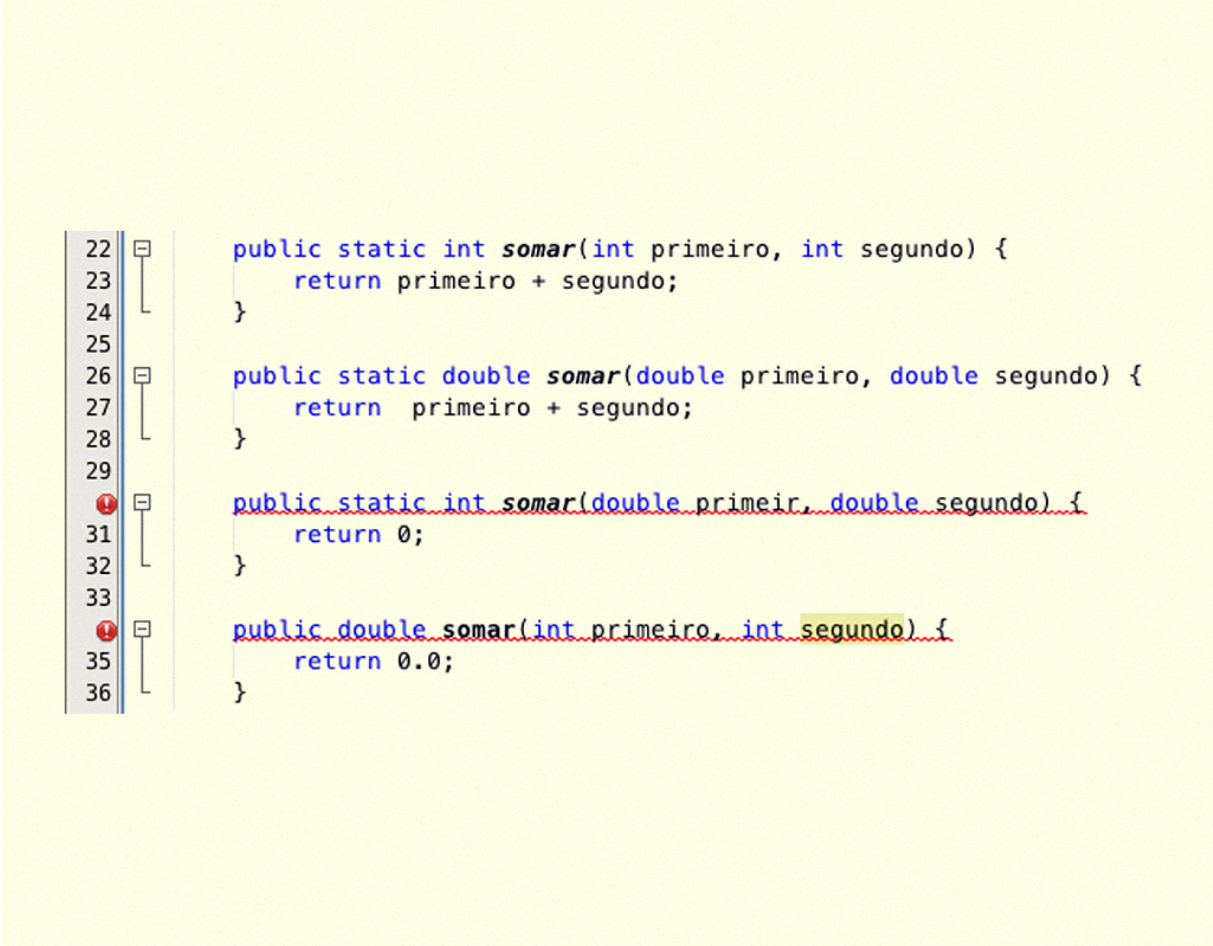


```
12  public class Principal {
13
14      public static void main(String[] args) {
15          double i = 0.0;
16          double j = 1.0;
17
18          double resultado = somar(i, j);
19          System.out.println(resultado);
20      }
21
22      public static int somar(int primeiro, int segundo) {
23          return primeiro + segundo;
24      }
25
26      public static double somar(double primeiro, double segundo) {
27          return  primeiro + segundo;
28      }
29  }
```

2FIGURA 5.23 - Exemplo de sobrecarga de métodos FONTE: NetBeans IDE.

A ordem dos argumentos do método conta como sendo métodos diferentes, ou seja, somarValor(int i, double j) é diferente de somarValor(double j, int i), lembrando de que o nome dos argumentos não influencia na sobrecarga, e sim os tipos dos argumentos. O retorno do método poderá ser alterado apenas se a lista dos argumentos for diferente de outra já declarada, pois o retorno não é considerado para a assinatura do método, somente o nome e os argumentos.

A Figura 2.6 representa alguns casos de sobrecarga de métodos que são válidos e alguns inválidos. Os métodos das linhas 22 e 26 são sobrecargas válidas, pois têm o mesmo nome e argumentos com tipos diferentes (o primeiro método tem argumentos do tipo int, e o segundo método, argumentos do tipo double); já o método da linha 30, que não é válido, apresenta como retorno um int e seus argumentos do tipo double, mas como o retorno não é considerado para verificar a validade de sobrecargas, esse método tem a mesma assinatura que o método da linha 26. O mesmo acontece para o método da linha 34, que tem argumentos do tipo int, que é a mesma assinatura do método da linha 22.



2FIGURA 6.23 - Casos de sobrecarga de métodos FONTE: NetBeans IDE.

Encapsulamento

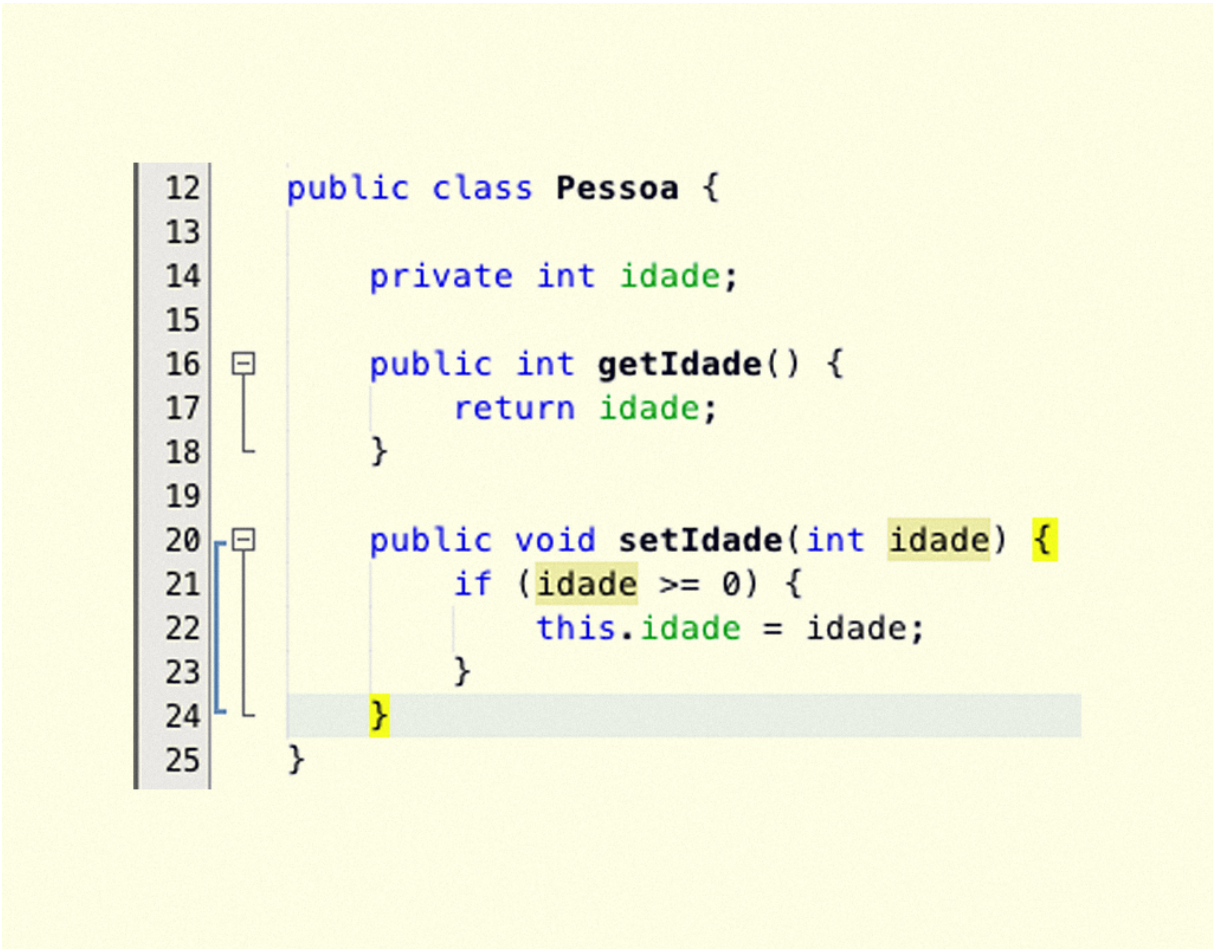
Até o momento, utilizamos somente o modificador de acesso ***public***, mas, agora, usaremos o modificador ***private***.Como o próprio nome diz, o modificador de acesso *public* fornece acesso público à variável a que ele está vinculado, ou seja, qualquer pessoa que utilizar o objeto terá acesso a essa variável. O modificador *private* é totalmente o contrário, somente métodos dentro da própria classe terão acesso à variável, como é exibido na Figura 2.7.

O acesso às variáveis de instância podem ser alterados pelos **modificadores de acesso**, que são ***public***, ***protected***, ***default*** ou ***private***. O modificador ***public*** é o menos restritivo, habilitando o acesso a qualquer lugar. O modificador ***protected*** restringe a visibilidade apenas para a própria classe ou para classes derivadas (por meio de herança), assim como classes do mesmo pacote. O modificador ***default*** especifica a visibilidade padrão, a mesma visibilidade, caso não fosse declarado explicitamente o modificador, torna os atributos, os métodos e as classes visíveis para todos os membros do mesmo pacote, enquanto o modificador ***private*** restringe a visibilidade para somente a própria classe que define objetos, atributos e métodos (RICARTE, 2007).

Podemos utilizar o modificador **final** para uma classe indicando que ela não poderá ser estendida, ou seja, nenhuma outra classe poderá herdar seus atributos e métodos. Caso uma classe tente estender uma classe final, ocorrerá um erro de compilação (RICARTE, 2007).

O modificador final não fica restrito apenas a classes, ele pode ser utilizado em métodos e em atributos. Um método final não pode ser redefinido em uma subclasse, e um atributo final não pode ter seu valor alterado, ou seja, é definida uma constante com o valor definido no momento da sua declaração ou no construtor da classe (RICARTE, 2007).

Os métodos podem ter argumentos definidos como final, sinalizando que esses não podem ser modificados.



```
12  public class Pessoa {
13
14      private int idade;
15
16      public int getIdade() {
17          return idade;
18      }
19
20      public void setIdade(int idade) {
21          if (idade >= 0) {
22              this.idade = idade;
23          }
24      }
25  }
```

2FIGURA 7.23 - Modificador de acesso private e métodos acessores FONTE: NetBeans IDE.

Segundo David (2009, on-line), o encapsulamento tem a seguinte funcionalidade:

> O Encapsulamento serve para controlar o acesso aos atributos e métodos de uma classe. É uma forma eficiente de proteger os dados manipulados dentro da classe, além de determinar onde esta classe poderá ser manipulada.

Além de somente ocultar os dados e fornecer a possibilidade de validação, Bates (2005, p. 58) cita que outro objetivo do encapsulamento tem envolvimento direto com a Engenharia de Software, já que pode evitar um possível retrabalho em todos os códigos que utilizam determinada classe em que não foi utilizado o encapsulamento. Supomos que as variáveis de instância são declaradas como modificadores de acesso ***public*** e são acessadas diretamente, sem passarem por nenhum método acessor, e diversos códigos utilizam essa classe. Após um tempo, é necessário modificar o projeto e incluir uma validação para que os dados de determinada variável não sejam negativos, isso influenciará todo o código que utiliza a classe, pois terá que ser escrita uma validação antes de atribuir o valor à variável de instância; contudo, se o encapsulamento fosse utilizado, seria incluída somente a validação dentro do método acessor, sem travar nenhum outro código que utilizasse a classe, gerando menos retrabalho.

A solução para a exposição dos dados das classes são os métodos acessores (ou métodos de configuração), que utilizam o conceito de encapsulamento para serem definidos. Os métodos acessores são métodos que, como seu nome define, fornecem meios para acessar os atributos das classes. Assim como os métodos normais de uma classe, os métodos acessores definem comportamento - o de realizar o acesso, seja ele para recuperar ou para atribuir um valor a um determinado atributo. Por padrão, a nomenclatura para um método acessor de recuperação de atributo é getNomeDoAtributo() e o método para atribuir um valor ao atributo é setNomeDoAtributo(valorDoAtributo), no qual o valorDoAtributo deve ser do mesmo tipo do atributo declarado, como é exibido na Figura 2.7, linhas 16 e 20, respectivamente.

Para o funcionamento correto do encapsulamento, além dos métodos acessores, são utilizados outros tipos de modificadores de acesso nas variáveis.

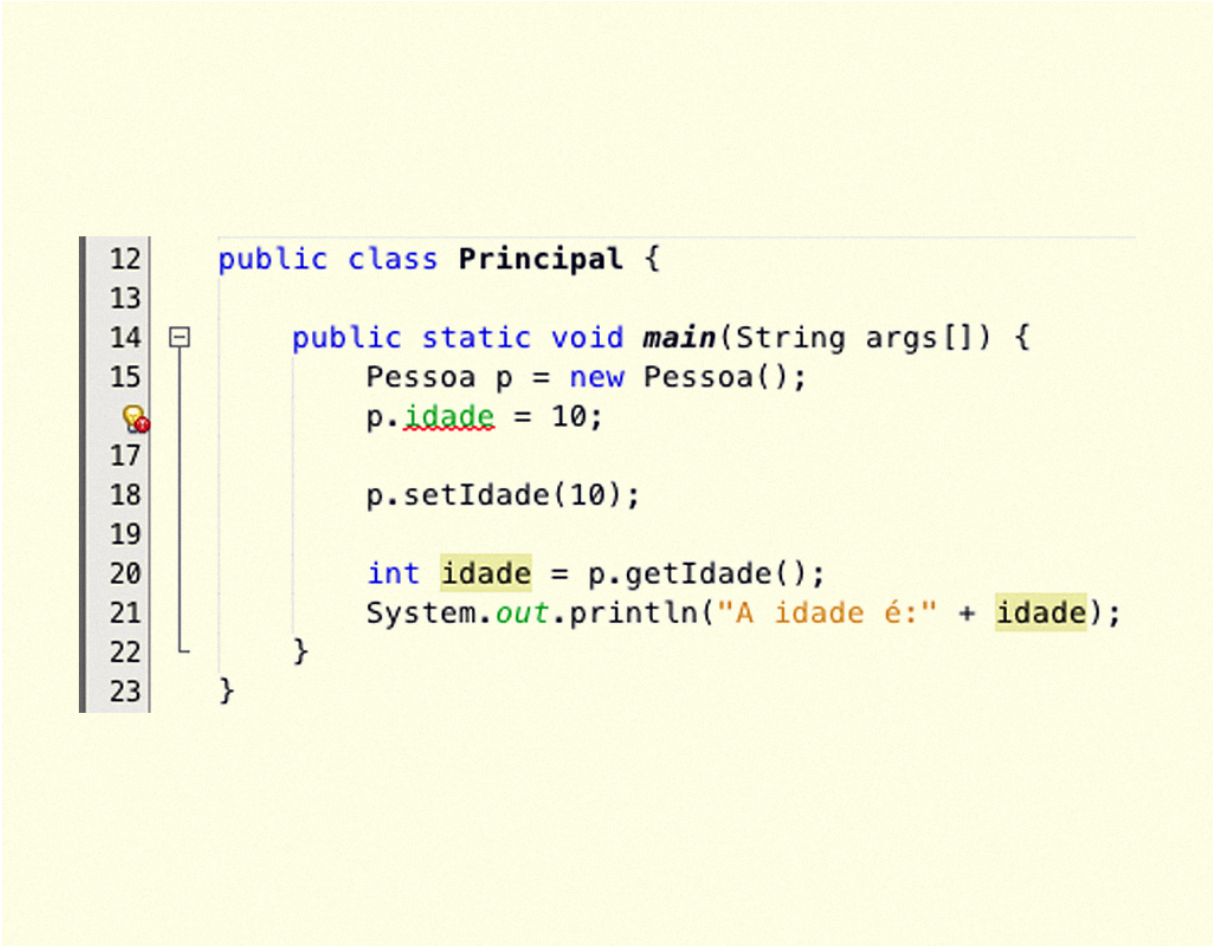
A variável de instância idade (linha 14, Figura 2.7) poderá ser acessada somente dentro da classe Pessoa, pois está com seu acesso restrito pelo ***private***. Entretanto os métodos acessores getIdade() e setIdade(int idade) estão marcados como ***public***,

podendo ser acessados de fora dessa classe.

A Figura 2.7 mostra, também, a utilização do encapsulamento por meio dos métodos *get* e *set*. Por padrão, o Java utiliza a seguinte nomenclatura para métodos acessores que setarão variáveis: **setNomeDaVariável**, recebendo por parâmetro o valor que será atribuído à variável. Já o método que retorna o valor da variável de instância é **getNomeDaVariável**, apresentando o retorno do mesmo tipo que a variável é definida. Os métodos acessores podem trabalhar tanto com tipos primitivos quanto com objetos.

No método **setIdade,** foi realizada uma validação para conferir se o valor do parâmetro idade é maior que zero, para, então, atribuí-lo à variável de instância idade. Como temos duas variáveis com o mesmo nome (a variável de instância idade e o argumento do método), é utilizada a palavra reservada **this** (linha 22, Figura 2.7), para fazer referência à variável de instância. O exemplo definido no método é simples e é realizada a atribuição apenas se a idade é maior que zero; poderíamos, porém, lançar uma exceção, alertando acerca do possível problema, mas as exceções serão estudadas na seção 4. No método **getIdade()**,a variável de instância foi utilizada sem o *this*, pois, nesse método, não há nenhuma variável com o mesmo nome, então, o compilador sabe que a referência é feita para a variável de instância.

A Figura 2.8 mostra a utilização dos métodos acessores da classe Pessoa (Figura 2.7), sendo que, na linha 18, é atribuído um valor para o método setIdade e, na linha 20, o valor da idade é atribuído a uma variável local do método. É possível ver que, na linha 16 da Figura 2.8, o compilador está exibindo um erro, pois estamos tentando acessar diretamente a variável idade, porém ela está marcada como ***private***, portanto, não é possível acessá-la fora da classe de origem. O acesso ao valor da variável de instância idade é feito pelo método acessor getIdade() na linha 20. Se removermos a linha 16, o texto que será exibido na linha 21 é 10, pois é o valor que foi setado na linha 18.



```
12  public class Principal {
13
14      public static void main(String args[]) {
15          Pessoa p = new Pessoa();
            p.idade = 10;
17
18          p.setIdade(10);
19
20          int idade = p.getIdade();
21          System.out.println("A idade é:" + idade);
22      }
23  }
```

2FIGURA 8.23 - Utilização encapsulamento FONTE: NetBeans IDE.

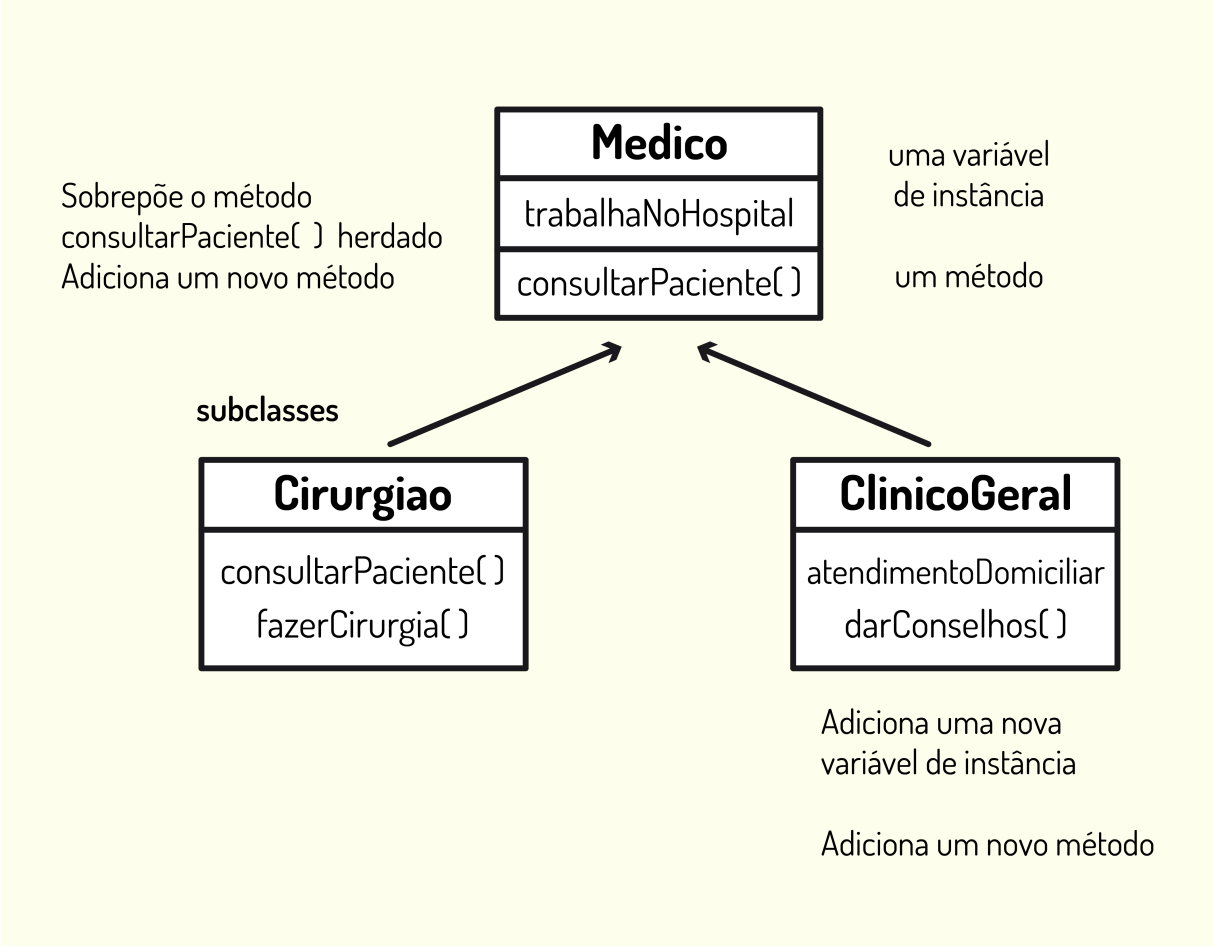
Herança

Um dos conceitos da Orientação a Objetos é diminuir a repetição de códigos o máximo possível, com isso, são evitados o retrabalho e os possíveis erros que foram tratados em um local, mas em outro não. A herança é uma forma de reutilização de códigos em que novas classes são criadas a partir de classes já existentes, utilizando características já implementadas que são compartilhadas com a nova classe (DEITEL, 2010).

A herança é muito utilizada na Orientação a Objetos, pois promove a reutilização e o reaproveitamento do código existente. As subclasses, criadas a partir de uma classe base que é mais genérica, são mais especializadas e herdam todas as características da superclasse (DAVID, 2009).

No Java, uma subclasse **estende** a superclasse, herdando todas as suas variáveis de instância e métodos, mas a subclasse poderá adicionar novos métodos e variáveis de instâncias que serão exclusivos da subclasse (BATES, 2005).

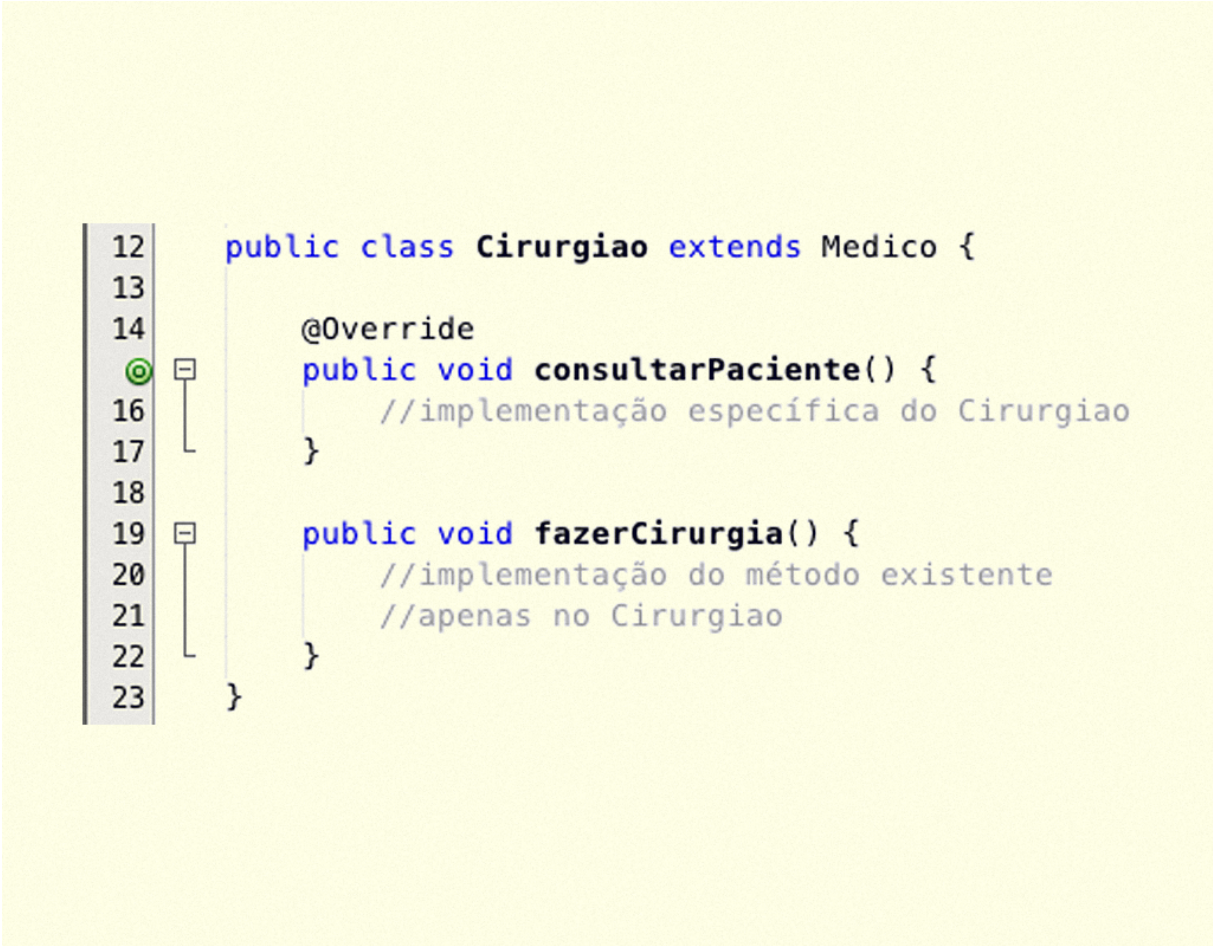
A Figura 2.9 mostra um exemplo de um projeto com herança, no qual a superclasse é Medico e suas subclasses são Cirurgiao e ClinicoGeral.



2FIGURA 9.23 - Exemplo de herança FONTE: Bates (2005, p. 136).

A superclasse Medico tem uma variável de instância trabalhaNoHospital e um método consultarPaciente(); como Cirurgiao e ClinicoGeral são tipos específicos de um Medico, eles herdam as características do Medico e ambos adicionam um novo método em cada. A classe Cirurgiao adiciona o método fazerCirurgia(), que é específico de um cirurgião, e não é, necessariamente, específico a todos os tipos de médicos. Já o ClinicoGeral tem uma nova variável de instância (atendimentoDomicialiar), que não é uma característica do Cirurgiao, e também adiciona um novo método darConselhos(). A classe Cirurgiao sobrepõe o método consultarPaciente(), ou seja, o método será reescrito com característica própria do Cirurgiao.

A herança é implementada por meio da palavra reservada extends na declaração da classe. A Figura 2.10 mostra a implementação da classe Cirurgiao, que estende a superclasse Medico.



```
12  public class Cirurgiao extends Medico {
13
14      @Override
15      public void consultarPaciente() {
16          //implementação específica do Cirurgiao
17      }
18
19      public void fazerCirurgia() {
20          //implementação do método existente
21          //apenas no Cirurgiao
22      }
23  }
```

2FIGURA 10.23 - Implementação da classe Cirurgiao FONTE: Bates (2005, p. 136).

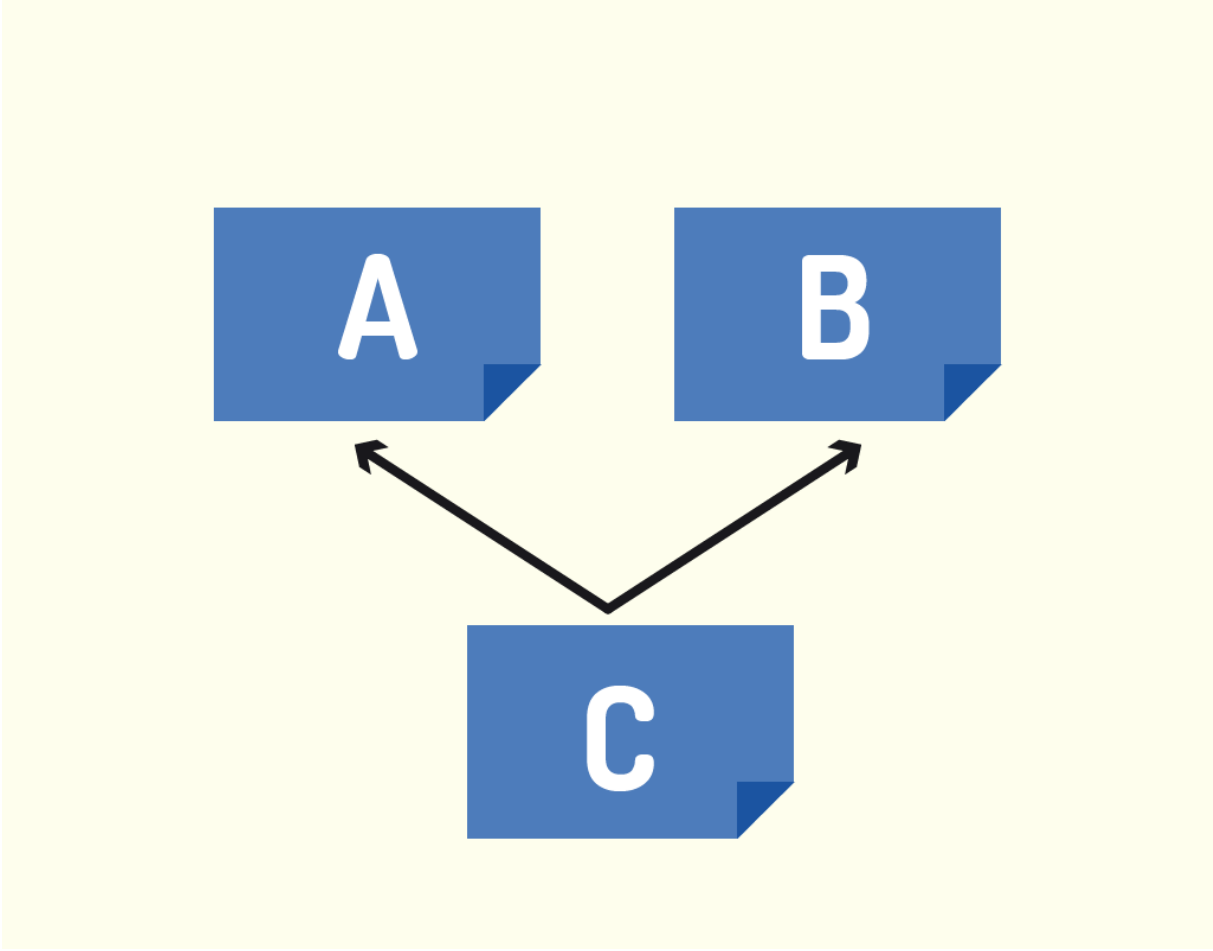
Reflita

Quando o objeto Cirurgiao é instanciado e é invocado o método consultarPaciente(), qual implementação será executada? A implementação da superclasse Medico, ou a implementação específica da classe Cirurgiao?

E se a classe instanciada fosse a classe ClinicoGeral, qual implementação do método consultarPaciente() seria executada?

Herança Múltipla

A herança múltipla permite que uma classe tenha mais que uma superclasse, ou seja, uma classe terá duas ou mais origens. Com isso, uma classe C, que é filha da classe A, pode ser filha, também, da classe B. A Figura 2.11 mostra como seria a herança múltipla de uma classe C com suas superclasses A e B.



2FIGURA 11.23 - Exemplo herança múltipla FONTE: Elaborada pelo autor.

A vantagem da herança múltipla é que ela permite uma maior capacidade de especificação entre as classes. Contudo, um dos problemas da herança é um método com assinatura igual ser implementado de uma forma na classe A e de outra forma na classe B. Caso a classe C realize uma chamada desse método de sua superclasse, de qual superclasse o método seria executado? Da superclasse A ou da superclasse B?

Nas linguagens que aceitam herança múltipla, isso pode ser definido de acordo com a especificação da linguagem, por exemplo, poderá utilizar o método da primeira classe que foi declarada como superclasse.

As linguagens mais utilizadas atualmente não aceitam herança múltipla, um dos motivos é o aumento significativo da complexidade de utilização da linguagem. Algumas linguagens foram criadas para serem de fácil utilização, portanto, a incorporação de uma funcionalidade poderosa, mas complexa poderia desincentivar a utilização dessa linguagem, como também estaria saindo do escopo em que foi criada.

É-um e Tem-um

A herança cria uma relação entre duas classes em que uma estende a outra, ou seja, a subclasse complementa a superclasse. Para a utilização correta da herança, o ideal é aplicar o teste **É-Um** (*Is-a*, em inglês). A herança deve ser utilizada apenas nos casos em que o teste É-Um é verdadeiro, caso contrário, será utilizada a herança em subclasses que não são derivadas da superclasse, podendo gerar futuras inconsistências no código (BATES, 2005).

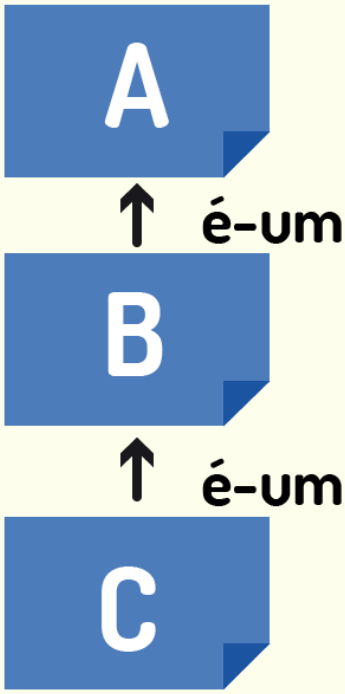
A seguir, estão exemplos do teste É-Um que estão corretos:

- O triângulo É-UMA forma.
- O cirurgião É-UM médico.
- O gato É-UM animal.

Ou seja, esses casos podem ser utilizados em uma derivação com herança e funcionarão. Se a classe B estende a classe A, a classe B É-UMA classe A(BATES, 2005).

O teste É-UM deve funcionar em qualquer lugar da árvore de herança, se a herança foi bem projetada. Caso a classe C estenda a classe B, e a classe B estenda a classe A, a classe C passará no teste É-UM tanto para a classe B quanto para a classe A (BATES, 2005).

A Figura 2.12 mostra o teste é-um explicado anteriormente, no qual a classe A é a superclasse, a classe B herda de A e a classe C herda de B, então, B é-um A, C é-um B e C é-um A.

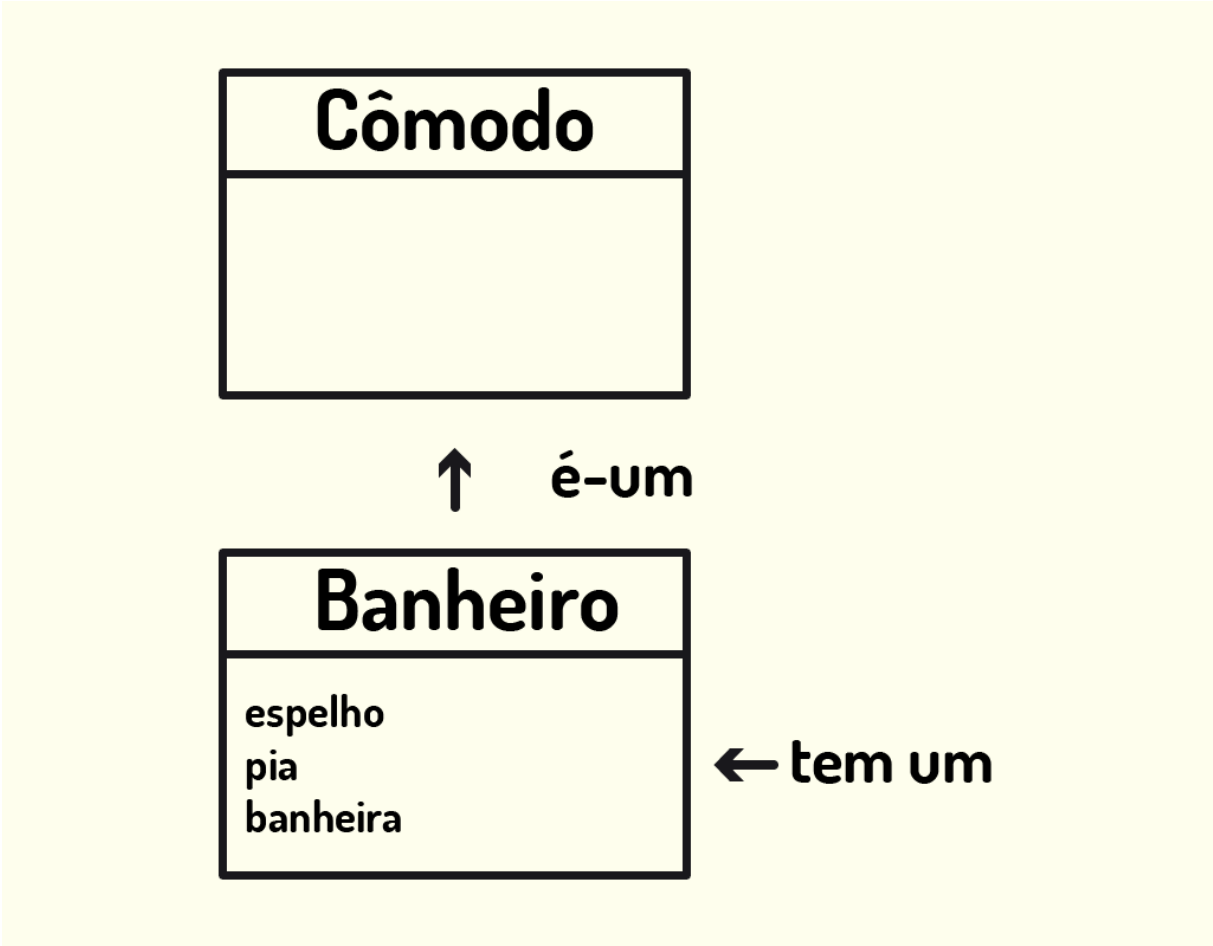


2FIGURA 12.23 - Teste é-um FONTE: Elaborada pelo autor.

Um detalhe importante é que o teste É-UM é unidirecional, ou seja, se a classe B É-UMA classe A, não é correto afirmar que a classe A É-UMA classe B.

Muitas vezes, quando o teste É-UM não está correto, ele pode ser visto como **Tem-um** (*Has-a*, em inglês), por exemplo, um banheiro TEM-UMA banheira, mas um banheiro não É-UMA banheira, muito menos o inverso, uma banheira não É-UM banheiro (BATES, 2005).

Normalmente, a aplicação do TEM-UM é utilizada para verificar quais variáveis de instância (atributos) uma classe poderá ter. Utilizando o exemplo do banheiro, ele TEM-UM espelho, TEM-UMA pia, contudo ele É-UM cômodo de uma casa (dependendo de como é o projeto da aplicação dessa casa), como exibido na Figura 2.13.



2FIGURA 13.23 - Tem-um FONTE: Elaborada pelo autor.

Classes abstratas

Uma classe abstrata é uma classe que não pode ser instanciada, ou seja, não é possível criar uma nova referência na memória para um objeto por meio do comando new para uma classe abstrata (CAELUM, on-line).

Se uma classe não pode ser instanciada, qual seria a vantagem de ter uma classe abstrata? A classe abstrata idealiza um tipo, ou seja, é um rascunho de como suas subclasses deverão ser ou se comportar.

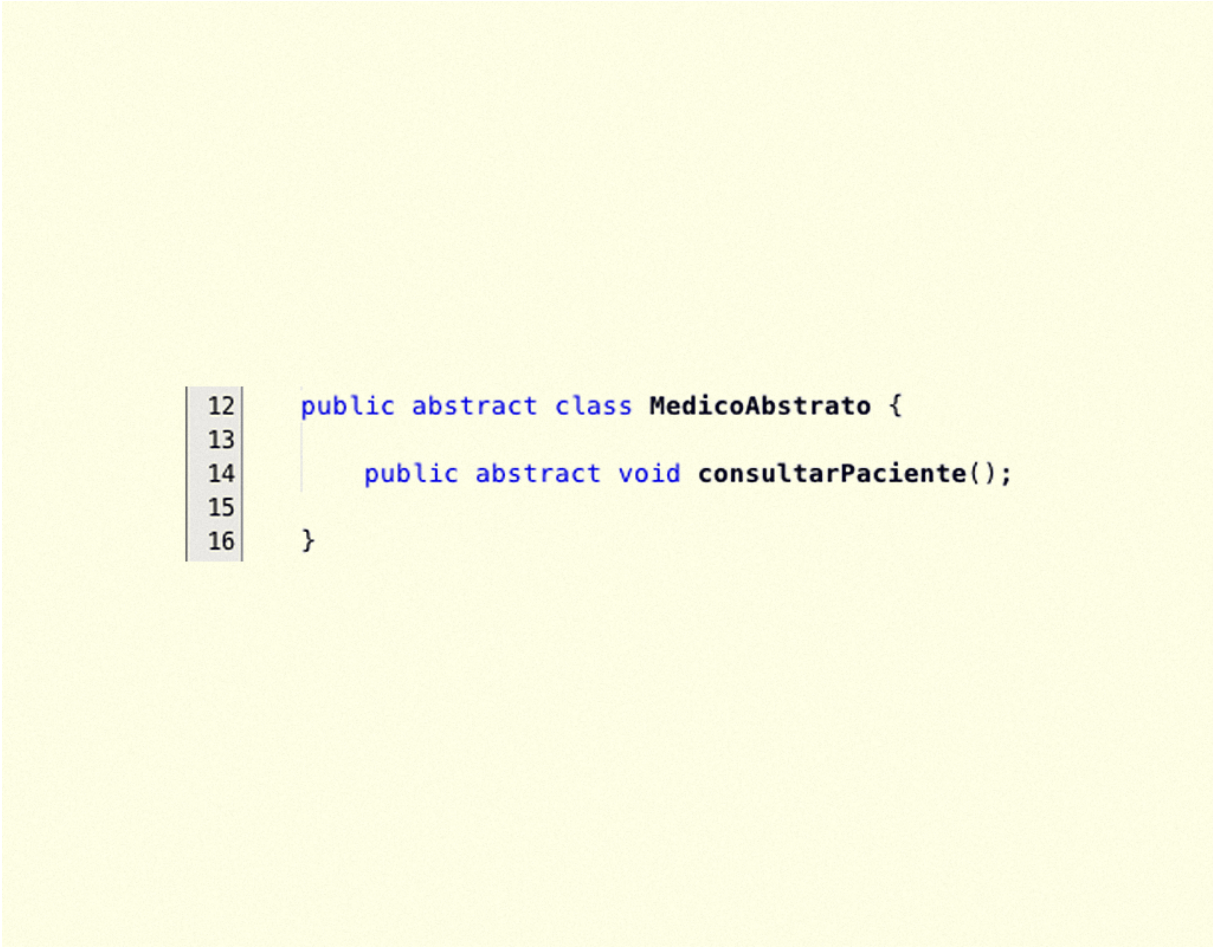
Uma classe torna-se abstrata quando é utilizado o modificador abstract na declaração da classe. Se o programador tentar instanciar uma classe abstrata, o compilador gerará um erro e não será possível compilar o código.

As classes abstratas são úteis nos casos em que uma determinada classe não pode ser instanciada, pois poderia gerar inconsistência no projeto, como no exemplo de herança em que há a classe Medico e dela derivam duas classes que são Cirurgiao e ClinicoGeral. Nesse caso, estaríamos trabalhando apenas com esses dois tipos de médicos, portanto, não faria sentido ter um objeto do tipo Medico instanciado, já que ele é genérico e não tem uma implementação específica para seus métodos; porém, em outros projetos, pode ser que seja útil instanciar um objeto do tipo Medico, por isso, seu uso deve ser específico para cada tipo de projeto.

As classes abstratas podem ter implementação de algum método que é geral para suas subclasses (que poderão se comportar do mesmo modo, ou reescrever o método), mas, também, pode apenas definir métodos que deverão ser implementados pelas suas subclasses. Esses últimos são os métodos abstratos.

Um método abstrato é um método sem implementação, porém com a palavra *abstract* na sua assinatura. Com isso, todas as classes que são herdadas da classe abstrata deverão implementar esse método, caso não implementem, será gerado erro de compilação, e uma classe poderá ter um método abstrato somente se ela for abstrata (CAELUM, *on-line*).

A seguir, está a declaração de um método abstrato.



```
12  public abstract class MedicoAbstrato {
13
14      public abstract void consultarPaciente();
15
16  }
```

2FIGURA 14.23 - Classe abstrata FONTE: NetBeans IDE.

O método abstrato não tem os sinais de início e término de códigos ({ e }), termina com ponto e vírgula (;) e pode ter argumentos definidos, ou não ter argumentos, como no exemplo anterior.

Polimorfismo

Polimorfimo pode ser definido como:

> Um princípio a partir do qual as classes derivadas de uma única classe base são capazes de invocar os método que, **embora apresentem a mesma assinatura, comportam-se de maneira diferente** [...].
>
> (BALBO, 2010, on-line, grifos do autor).

Segundo Horstmann (2004, p. 356), "***o termo polimorfismo vem do grego e significa muitas formas***". Ou seja, uma classe base pode ter funcionalidades (atributos e objetos) que serão utilizadas em objetos distintos, mas implementadas diferentemente em cada objeto.

O polimorfismo pode ser aplicado com os exemplos citados anteriormente na seção a respeito da Herança. Um Cirurgiao **é-um** Médico, assim como o ClinicoGeral também, ou seja, o Médico pode ter várias formas, como Cirurgião e/ou Clínico Geral. O polimorfismo faz com que um objeto tenha a capacidade de ser referenciado de diversas formas (porém somente de um tipo).

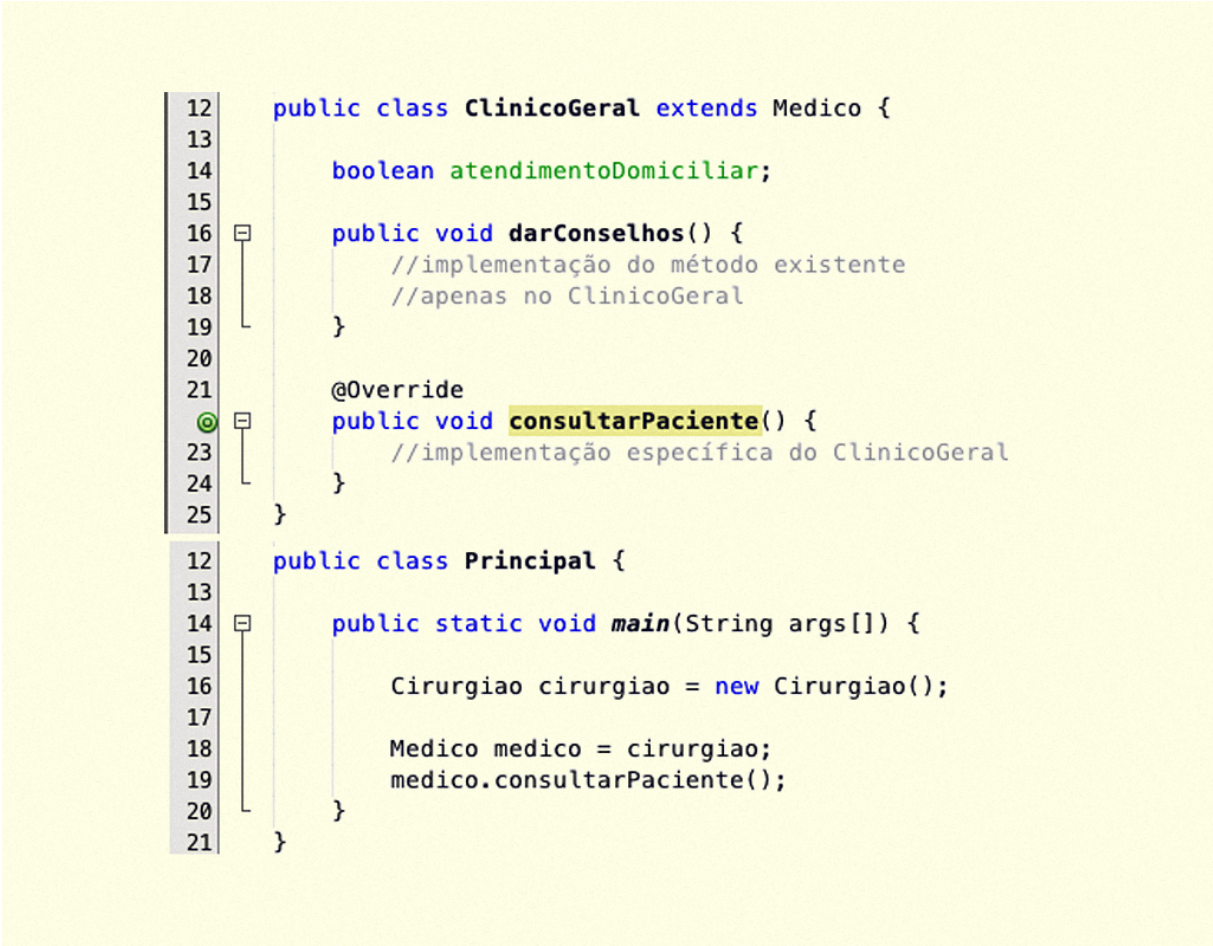
O exemplo mais geral de polimorfismo, no Java, é que todas as classes são do tipo Objeto, ou seja, a classe Cirurgião, além de ser um Médico, também é um Objeto, portanto, se tivermos um método que receba um Objeto por parâmetro, poderemos passar um Objeto do tipo Cirurgião (como Cirurgião descende de Objeto, o compilador não acusará erro no código, porém poderá ocorrer erro durante a execução do programa, já que o código pode não estar preparado para receber determinado tipo de Objeto).

Voltando ao nosso exemplo dos Médicos, se alterarmos a classe ClinicoGeral e inserirmos o método consultarPaciente(), implementado de forma específica para o Clínico Geral, como é exibido na Figura 2.15, nas linhas 12-25 da classe ClinicoGeral, e fossem executadas as linhas 16-19 da classe Principal, a implementação do método consultarPaciente() seria executada por meio da classe Cirurgião. Isso se deve ao

fato de que foi criada uma referência para um objeto Cirurgiao, e, como a classe Medico é a superclasse do Cirurgiao, a atribuição da linha 18 da classe Principal da Figura 2.15 é executada corretamente, caracterizando o polimorfismo.

¶ Reflita

Com base na Figura 2.15, se fosse declarado e instanciado um objeto do tipo ClinicoGeral, seria possível atribuir esse objeto ao objeto Cirurgiao (declarado na linha 16)?

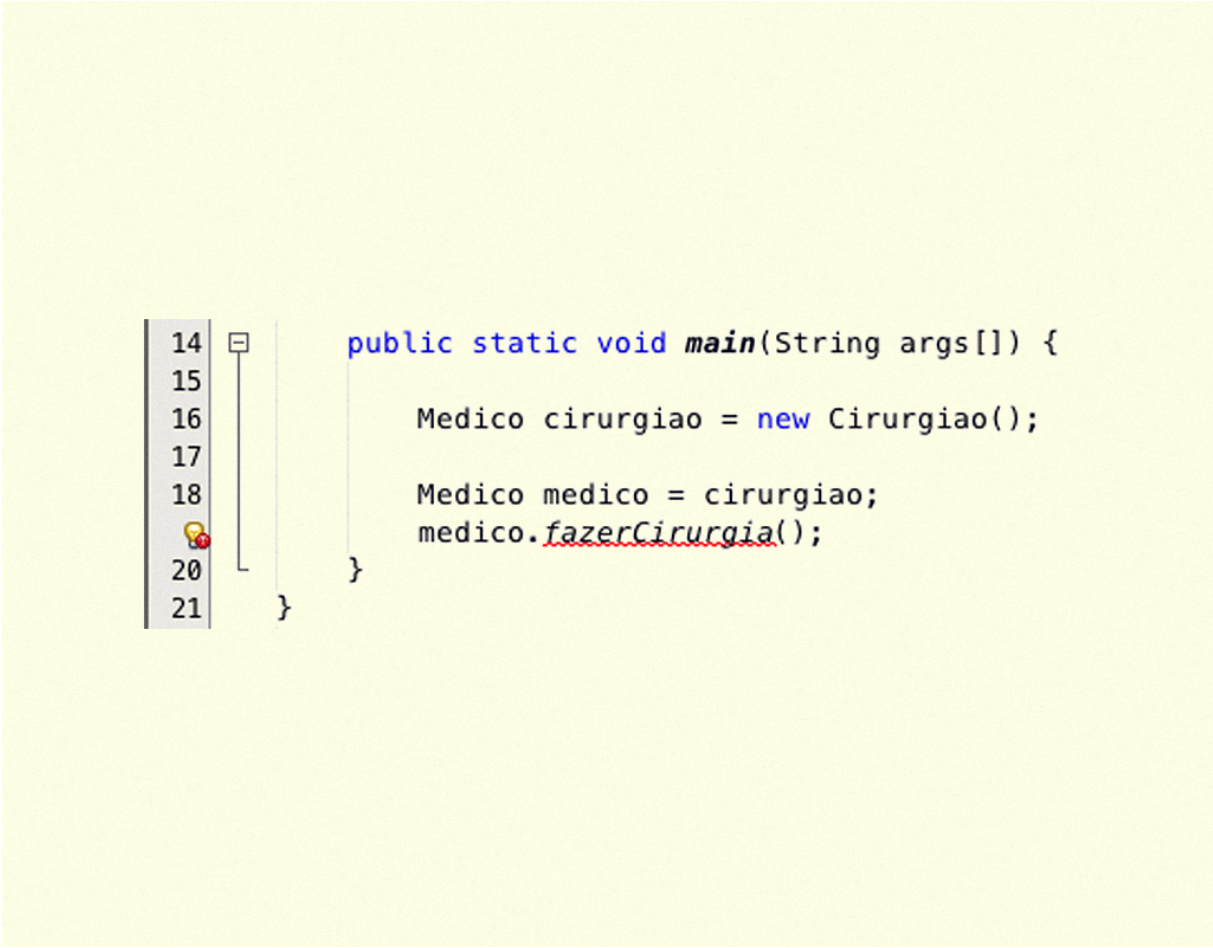


```
12  public class ClinicoGeral extends Medico {
13
14      boolean atendimentoDomiciliar;
15
16      public void darConselhos() {
17          //implementação do método existente
18          //apenas no ClinicoGeral
19      }
20
21      @Override
22      public void consultarPaciente() {
23          //implementação específica do ClinicoGeral
24      }
25  }
```

```
12  public class Principal {
13
14      public static void main(String args[]) {
15
16          Cirurgiao cirurgiao = new Cirurgiao();
17
18          Medico medico = cirurgiao;
19          medico.consultarPaciente();
20      }
21  }
```

2FIGURA 15.23 - Polimorfismo FONTE: NetBeans IDE.

Caso a classe Principal da Figura 2.15 fosse alterada para a da Figura 2.16, o código funcionaria normalmente, pois o objeto referenciado será do tipo Cirurgiao, por mais que seja declarado como Medico, pois Medico é a superclasse de Cirurgiao. Contudo, na linha 19, não é possível invocar o método fazerCirurgia(), que está declarado e definido na classe Cirurgiao, pois o objeto que está invocando o método é do tipo Medico. Mesmo que o objeto medico esteja recebendo um objeto referenciado como Cirurgiao, o compilador entende que o objeto medico é do tipo Medico, e, na classe Medico, não existe o método fazerCirurgia(), tanto que o compilador emite mensagem de erro de compilação que o método não existe.



2FIGURA 16.23 - Polimorfismo FONTE: NetBeans IDE.

Se em algum método da aplicação for necessário receber um Medico por argumento, esse medico poderá ser qualquer objeto referenciado como Medico, Cirurgiao ou ClinicoGeral, pois, mesmo Cirurgiao e ClinicoGeral sendo classes mais específicas, elas se encaixam como Medicos; por exemplo, um paciente está doente e foi para um plantão de um hospital, ele poderá ser atendido por qualquer tipo de médico que esteja atendendo no plantão, porém, se precisar de uma cirurgia, somente o cirurgião poderá realizar.

Sobreposição de métodos

A sobreposição de métodos (*override*), ou reescrita de métodos, permite reescrever nas subclasses os métodos criados na superclasse. Ao contrário da sobrecarga de métodos, o método que será sobreposto deverá ter a assinatura idêntica, ou seja, mesmo nome, tipo de retorno e tipos e quantidades de parâmetros (ALVES, on-line).

Os métodos podem ser sobrepostos em classe concreta ou abstrata. Como um construtor é um método, ele também pode ser sobreposto. No método sobreposto, podem ser utilizadas algumas palavras-chave para fazer menção à superclasse ou a atributos utilizados diretamente na classe. A palavra-chave **super** é utilizada para informar ao programa que será utilizado o construtor da superclasse.

A principal funcionalidade de um método sobreposto é alterar a forma como o método será executado, principalmente se a subclasse tem características específicas e se comporta de uma forma diferente da superclasse.

No exemplo dos médicos citado na seção do Polimorfismo, o método consultarPaciente() é um método sobreposto por ambas as classes que derivam de Medico. Quando um método é sobreposto, o NetBeans pede para indicar o método com a anotação @Override acima do método, como mostra a Figura 2.17.

```
12  public class Cirurgiao extends Medico {
13
14      @Override
15      public void consultarPaciente() {
16          System.out.println("Esse é o cirurgião");
17      }
18
19      public void fazerCirurgia() {
20          //implementação do método existente
21          //apenas no Cirurgiao
22      }
23  }
```

2FIGURA 17.23 - Sobreposição de métodos FONTE: adaptada de Bates (2005, p. 126).

Utilizando o exemplo da seção de Polimorfismo, quando for referenciado um objeto Cirurgiao (podendo ser declarado do tipo Cirurgiao ou Medico por causa do polimorfismo) e invocado o método consultarPaciente(), será executada a implementação da classe Cirurgiao, imprimindo o texto da linha 16. Caso essa classe não tivesse o método sobreposto, seria executada a implementação da classe Medico, pois é sua superclasse e nela há uma implementação para esse método.

Interfaces

Uma interface nada mais é que uma classe com todos seus métodos abstratos, ou seja, sem nenhuma implementação, somente com a especificação da funcionalidade que uma classe deve conter (RICARTE, 2007).

Ricarte (2007, p. 28) define interface Java como:

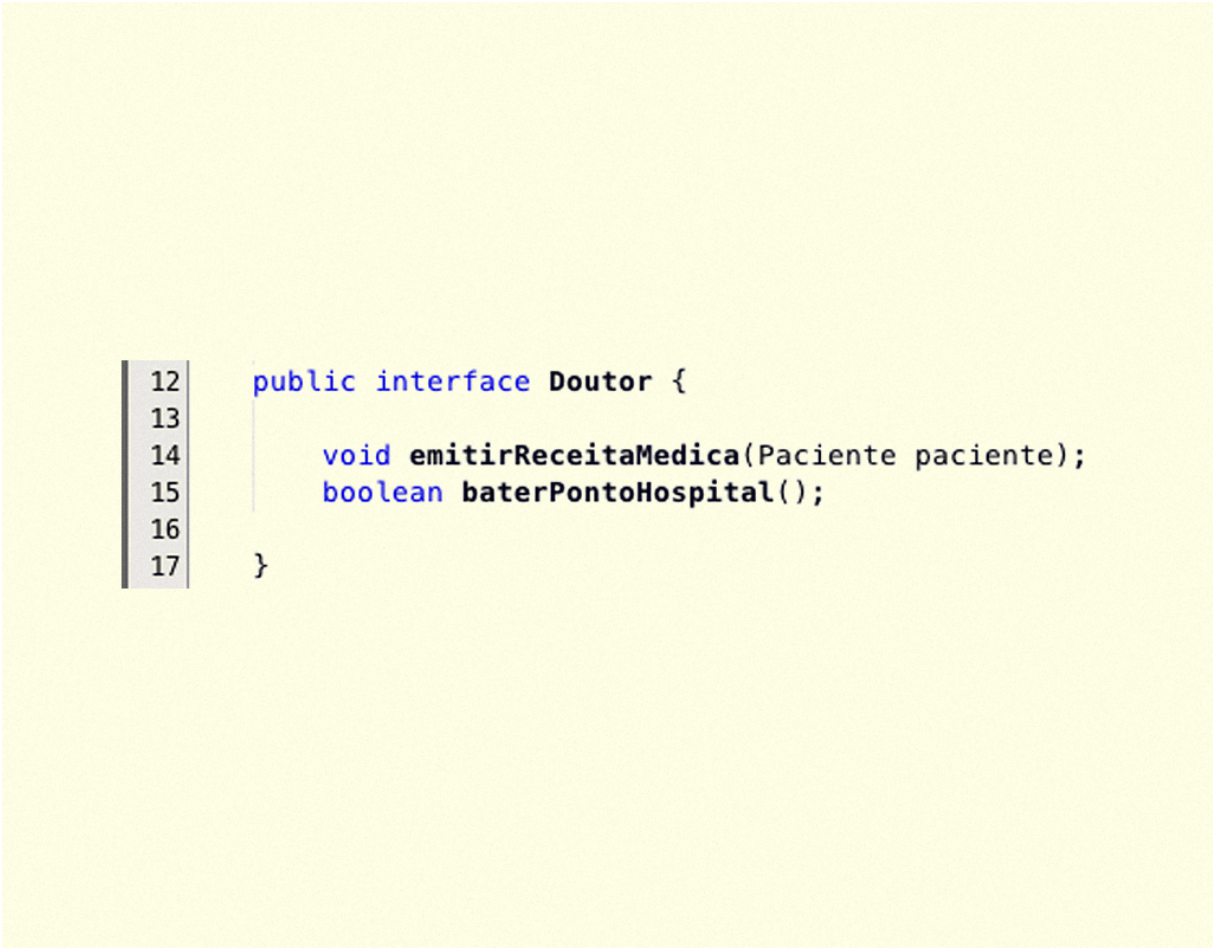
> uma classe abstrata para a qual todos os métodos são implicitamente abstract e public e, todos os atributos são implicitamente static e final. [...] uma interface implementa uma classe abstrata pura.

Qual seria a diferença de uma interface para uma classe abstrata? A interface não tem nenhum corpo, apenas declaração, já uma classe abstrata deverá ter, pelo menos, um método abstrato, porém pode ter definição (implementação) de outros métodos e atributos (RICARTE, 2007). O corpo de uma interface define apenas assinaturas de métodos e constantes, sem nenhuma implementação e não pode ter atributos.

Uma classe abstrata é estendida (como visto na seção de Herança) por classes derivadas, enquanto uma interface é implementada, mediante a palavra-chave ***implements***, por outras classes (RICARTE, 2007). A interface estabelece um contrato com as classes que irão implementá-la, pois, quando uma classe implementa uma interface, ela deve implementar **todos** os métodos declarados na interface, ou seja, todas as funcionalidades especificadas na interface serão oferecidas pela classe (RICARTE, 2007).

Ao contrário da herança, na qual uma classe pode herdar todas as características (atributos, métodos, constantes, dentre outras) apenas de uma superclasse, uma classe pode implementar diversas interfaces, desde que respeite o contrato e implemente todos os métodos declarados em todas as interfaces.

A Figura 2.18 mostra a declaração de uma interface Doutor com dois métodos que deverão ser implementados por uma classe concreta. Lembrando de que, na implementação dos métodos pela classe, esses métodos serão sobrepostos, portanto, deve-se marcá-los com a anotação @Override.



2FIGURA 18.23 - Interface Doutor FONTE: NetBeans IDE.

Bates (2005, p. 165) mostra algumas dicas interessantes para decidir quando criar uma classe, uma subclasse, uma classe abstrata ou uma interface:

- caso a classe não passe no teste É-UM com nenhum outro tipo, crie uma nova classe que não estenda nada.

- caso uma classe necessite ser mais específica que uma outra classe já existente e necessite sobrepor/adicionar novos comportamentos, estenda uma classe (crie uma subclasse utilizando herança).
- utilize uma classe abstrata para definir um modelo para as subclasses, se tiver que implementar algo na superclasse. Como ela não pode ser instanciada, não há a possibilidade de ter um objeto da classe abstrata.
- utilize uma interface para definir funções que as outras classes devem realizar.

Tratamento de exceções

Para Deitel (2010, p. 336), "***uma exceção é uma indicação de um problema que ocorre durante a execução de um programa. O nome exceção indica que é algo que não ocorre frequentemente***".

Uma exceção pode ocorrer de diversas maneiras, principalmente em pontos que não foram tratados pelo programador em situações específicas, por exemplo, tentar utilizar métodos de um objeto que está nulo e não foi ainda referenciado. Nesse caso, a exceção ocorrerá e a execução do programa será abortada.

O tratamento de exceções em Java permite que essa exceção seja contornada e o programa continue sua execução em um estado satisfatório, com nenhum dado corrompido ou incorreto.

O exemplo citado anteriormente, de um objeto não referenciado, gera uma exceção *NullPointerException*, pois é utilizada uma referência nula quando se espera um objeto.

O tratamento de exceções remove a linha de execução da linha principal e a desvia para o bloco de código para realizar o tratamento desse erro. A escolha da exceção a ser tratada pode ser feita pelo programador, desde todas as exceções em

geral, até as exceções de um certo tipo ou as exceções de um grupo de tipos relacionados (exceções que têm relação por meio de uma superclasse em sua hierarquia) (DEITEL, 2010).

Classes de exceção

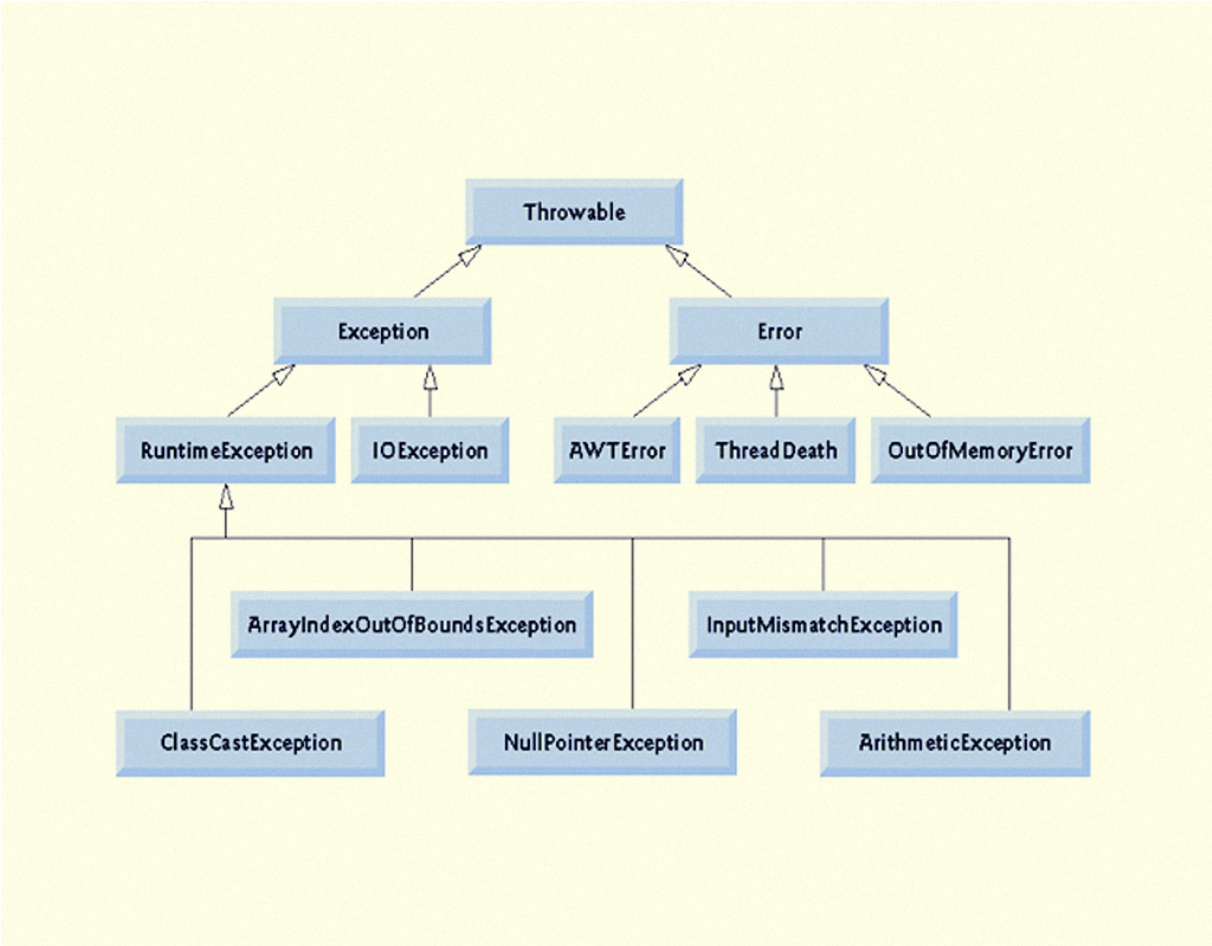
As exceções podem ser de dois tipos (VINICIUS, 2013, on-line):

- ***Checkedexceptions* (exceções verificadas)**: são erros que acontecem fora do controle do programa, mas devem ser tratadas pelo desenvolvedor.
- ***Unchecked exceptions* (exceções não verificadas)**: também chamadas de *Runtime Exceptions*, pois são, normalmente, em tempo de execução, podem ser evitadas se forem tratadas e analisadas pelo desenvolvedor. Caso não haja um tratamento, o programa irá parar em tempo de execução.

Além das exceções, temos os ***Errors***, que diz respeito a um tipo específico de exceção que não pode ser tratado pela aplicação, impossibilitando-a de continuar executando.

Como todo outro objeto em Java, uma exceção é uma instância de uma classe em uma hierarquia de classes. As exceções sempre serão subclasses da classe ***Exception***, que, juntamente com a classe ***Error***, são herdadas da superclasse ***Throwable*** (Figura 2.19).

A diferença entre Erro e Exceção é que o primeiro não poderá ser tratado, enquanto todas as subclasses de *Exception* são exceções e devem ser tratadas, exceto a classe *RunTimeException*, que é um erro e não precisa de tratamento, sendo utilizados os blocos ***try/catch*** e/ou ***throw/throws*** (VINICIUS, 2013, on-line).



2FIGURA 19.23 - Hierarquia das classes de exceções FONTE: Vinicius (2013, *on-line*).

Identificar se as exceções serão verificadas ou não verificadas não é algo exato, porém os seguintes conselhos podem ser considerados (BARNES, 2004):

- utilizar exceções não verificadas em situações que devem levar à falha do programa, provavelmente proveniente de um erro lógico que evitará que o programa prossiga; enquanto as exceções verificadas devem ser utilizadas nos momentos em que um problema surgiu, mas que pode ser tratado para sua recuperação.
- utilizar exceções não verificadas em situações que deveriam ser evitadas, como um índice inválido em um *array*, resultando na exceção *IndexOutOfBoundsException*, que poderia ser evitada com mais atenção do desenvolvedor, deduzindo-se que as exceções verificadas são utilizadas para

verificar situações que estão fora do controle do desenvolvedor, como um disco cheio ao realizar a gravação de um arquivo.

Lançando/propagando exceções

As exceções podem ser capturadas (como será visto na próxima seção) e então tratadas para verificar qual erro e o que pode ser feito com elas, ou pode ser propagada para ser tratada em um outro método, porque, por algum motivo, não pôde ser tratada no método que gerou a exceção (VINICIUS, 2013, on-line).

Os métodos que são executados são colocados em uma pilha; à medida que o fluxo de execução termina determinado método, ele é removido da pilha, e o fluxo vai para o próximo método. A propagação de exceções faz com que a exceção não seja tratada nesse método e seja propagada para um nível acima da pilha (VINICIUS, 2013, on-line).

A propagação é definida pela cláusula ***throws*** na declaração de um método. Ou seja, a cláusula *throws* declara quais exceções podem ser lançadas pelo método, auxiliando outros desenvolvedores que poderão utilizar o código, deixando explícito qual erro pode acontecer. A cláusula ***throws*** é exibida na Figura 2.20.

2FIGURA 20.23 - Cláusula throws FONTE: Vinicius (2013, *on-line*).

Ao contrário da cláusula *throws*, que é declarada no cabeçalho do método e propaga a exceção que pode ocorrer no método para outro método, a cláusula ***throw*** lança uma nova exceção, caso ocorra algum erro dentro do método, para essa nova exceção ser tratada em outro método. A Figura 2.21 mostra a utilização do ***throw***.

```
public class Exemplo_Throw {

    public static void saque(double valor) {
        if (valor > 400) {
            IllegalArgumentException erro = new IllegalArgumentException();
            throw erro;
        } else {
            System.out.println("Valor retirado da conta: R$ " + valor);
                }
        }

        public static void main(String[] args) {
            saque(1500);
        }
}
```

2FIGURA 21.23 - Cláusula throw FONTE: Vinicius (2013, *on-line*).

A diferença entre as duas cláusulas é que a ***throw*** cria uma nova exceção, na Figura 2.21 foi criada a exceção IllegalArgumentException(), que poderia ser criada com uma string no seu construtor, informando qual erro aconteceu. O exemplo foi somente uma demonstração de criação de uma exceção, por isso a exceção foi lançada em um teste com um valor aleatório. A exceção poderia ser lançada diretamente, sem ser atribuída a nenhuma variável, por exemplo:

throw new IllegalArgumentException("Valor solicitado maior que o saldo disponível");

Essa exceção foi criada com uma mensagem de erro no construtor de exceção, portanto, o método que captura a exceção poderá ver qual erro ocorreu.

Capturando exceções

Quando um método contém um código que pode ser perigoso de executar, ou seja, pode ocorrer uma exceção que será lançada/propagada por meio do *throw/throws*, ele deverá saber como lidar com o possível erro (BARNES, 2004).

A utilização da cláusula ***try*** indica que o bloco de código seguinte (dentro da cláusula) estará realizando algo arriscado, e a cláusula ***catch*** serve para tratar a exceção lançada dentro do bloco *try*. Contudo, se a exceção não for capturada dentro do bloco *catch*, ou seja, se a exceção não for a esperada pelo bloco *catch*, o sistema terá sua execução interrompida (VINICIUS, 2013, on-line).

Uma utilização frequente do bloco *try/catch* ocorre em operações com banco de dados; nas transações de *Rollback*, caso ocorra uma exceção, as alterações não são persistidas no banco de dados, executando um *Rollback* e impedindo que o banco de dados possa ter dados inconsistentes (VINICIUS, 2013, on-line).

Dentro do bloco do *try*, no momento em que um método é chamado e lança uma exceção, a execução da instrução é interrompida naquele método e é desviada para a cláusula *catch* correspondente (caso a cláusula trate a exceção que ocorreu). Caso não ocorra exceção durante a execução do código que está na cláusula *try*, a cláusula catch não será executada e o fluxo de instruções seguirá normalmente pelo código (BARNES, 2004).

O bloco *try/catch* pode ter uma terceira cláusula, que é chamada de ***finally***, que sempre é executada, finalizando a sequência de comandos, independente de ter ocorrido algum erro no sistema. O bloco *finally* é opcional, ao contrário do *try/catch* (se for utilizado o *try*, o catch deve vir obrigatoriamente após o *try* e será executado somente se acontecer um erro na execução), não sendo necessário aparecer, porém, quando aparecer, deve vir sempre após o bloco do *catch* (VINICIUS, 2013, on-line).

Normalmente, um bloco *finally* é utilizado para encerrar a sequência de comandos que vieram anteriormente, como finalizar uma conexão com um banco de dados, fechar algum arquivo em edição, dentre outros.

A Figura 2.22 mostra a utilização dos blocos *try/catch/finally* em uma operação de divisão, a qual não pode ter um denominador igual a zero e não pode receber letras para a operação.

```
15  public class ExemploDivisao {
16
17      public static int calculaQuociente(int numerador, int denominador) throws ArithmeticException{
18          return numerador / denominador;
19      }
20
21      public static void main(String[] args) {
22          Scanner sc = new Scanner(System.in);
23          boolean continua = true;
24
25          do {
26              try {
27                  System.out.print("Numerador: ");
28                  int numerador = sc.nextInt();
29
30                  System.out.print("Denominador: ");
31                  int denominador = sc.nextInt();
32
33                  int resultador = calculaQuociente(numerador, denominador);
34                  System.out.println("Resultado: "+resultador);
35
36                  continua = false;
37
38              } catch (InputMismatchException erro1) {
39                  System.err.println("Não é permitido inserir letras, informe apenas números inteiros!");
40                  sc.nextLine(); //descarta a entrada errada do usuário
41              } catch(ArithmeticException erro2) {
42                  System.err.println("O número do divisor deve ser diferente de 0!");
43              } finally {
44                  System.out.println("Execução do Finally!");
45              }
46          } while(continua);
47      }
48  }
```

2FIGURA 22.23 - Exemplo try/catch/finally FONTE: Vinicius (2013, *on-line*).

O exemplo solicita ao usuário informar dois números (numerador e denominador) para realizar a divisão entre eles. Na linha 25, é realizado um laço *do - while*, que continuará executando até o momento em que a variável continua receba o valor

false, que ocorre na linha 36, quando a divisão foi realizada com sucesso, caso contrário, o laço será executado até o momento que não ocorra nenhum erro.

A chamada ao método calculaQuociente é realizada dentro do bloco *try*, pois o método lança uma exceção ArithmeticException, que é declarada na assinatura do método (linha 17). O tratamento das exceções ocorre nas linhas 38, tratando a exceção InputMismatchException, a qual verifica se houve algum problema com os dados inseridos, como no exemplo que espera um inteiro e, se for inserida uma letra, o Java automaticamente lançará a exceção. Na linha 41, é tratada uma exceção do tipo ArithmeticException, que ocorre quando houve algum erro aritmético, como uma divisão por 0.

Caso o tratamento da exceção da linha 38 fosse retirado, o sistema iria interromper sua execução, pois, se fosse informada uma letra, aconteceria um erro que não foi tratado e que não era esperado que acontecesse.

Nas linhas 38 e 41, são tratadas múltiplas exceções em um bloco *try/catch*, não sendo necessário tratar apenas uma exceção, mas podendo tratar várias exceções que podem acontecer, da mais específica para a mais geral. Como todas as exceções são derivadas da classe *Exception*, uma única cláusula *catch* com essa classe iria capturar a exceção, porém teria menos detalhes, não foi capturada com uma classe de instância mais específica (BARNES, 2004).

Verificação dos erros

Dentro de uma exceção, é possível verificar algumas mensagens do erro que ocorreu. Foi mencionado anteriormente que era possível lançar uma nova exceção informando uma mensagem de erro no seu construtor, agora, veremos alguns métodos que são nativos das exceções que mostram algumas informações sobre o erro.

Esses métodos são da classe ***Throwable***, que é a superclasse de todos os erros e as exceções, e são gerados para sua subclasses.

Os principais métodos de captura de erros são (VINICIUS, 2013, on-line):

- **printStackTrace**: imprime a pilha de erro encontrada na execução, exibindo onde foi que ocorreu o erro;
- **getStackTrace:** recupera as informações que são impressas pelo método acima;
- **getMessage:** método que retorna a mensagem que contém a lista dos erros armazenados na exceção no formato String.

O método getMessage() é constantemente utilizado, pois informa a mensagem completa do erro que ocorreu, sendo mais fácil a identificação do erro e, também, personalizando a mensagem de erro, para que não chegue ao usuário final as mensagens de erro padrão do Java, o que pode assustar o usuário.

Criação de classes de exceção

Podem existir casos em que as classes de exceção padrão não tenham clareza e detalhamento suficiente para informarem ao usuário qual a natureza do problema. Nesses casos, é interessante ter uma classe mais específica que pode ser definida utilizando herança (BARNES, 2004).

As novas classes de exceções podem ser subclasses de Exception, tornando-se novas classes de exceções verificadas, enquanto as que são herdadas de RuntimeException seriam exceções não verificadas (BARNES, 2004).

As classes de exceção suportam a passagem de um argumento *String* para o construtor, que seria a mensagem de diagnóstico para essa exceção. O principal motivo de construir classes específicas de exceção é informar mais detalhes acerca do erro que aconteceu em determinada parte do sistema. Um exemplo citado por Barnes (2004, p. 301) é a busca de um dado, por meio de uma chave, em uma lista de

endereço. Quando a busca falha, a classe de exceção personalizada informa qual erro aconteceu e qual índice (chave) foi utilizado para realizar a busca, que, em determinada situação da aplicação, pode ser útil para solucionar o problema.

O exemplo da classe de exceção criada por Barnes (2004) está exibido na Figura 2.23. A classe NoMatchingDetailsIException é herdada da classe Exception e tem um construtor recebendo uma *String*, que é a chave que foi utilizada como parâmetro para detalhar mais a mensagem de erro. Foi criado um método getKey() para retornar a chave, por meio do encapsulamento, para não acessar diretamente um atributo da classe, e um método toString(), que é o método de diagnóstico que retorna a mensagem de erro personalizada para essa classe. Como essa classe é herdada da classe *Exception*, além da utilização dos métodos criados especificamente para essa nova exceção, podem ser utilizados métodos padrões das exceções, como printStackTrace() e getMessage().

```
/**
 * Captura uma chave que falhou em corresponder a uma entrada
 * no catálogo de endereços.
 *
 * @author David J. Barnes e Michael Kölling.
 * @version 2002.05.14
 */
public class NoMatchingDetailsException extends Exception
{
  // A chave sem correspondência.
    private String key;

    /**
     * Armazena os detalhes no erro.
     * @param key A chave sem correspondência.
     */
    public NoMatchingDetailsException(String key)
    {
        this.key = key;
    }

    /**
     * @return A chave no erro.
     */
    public String getKey()
    {
        return key;
    }

    /**
     * @return Uma string de diagnóstico que contém a chave em
     * erro.
     */
    public String toString()
    {
        return "No details matching: " + key + " were found.";
    }
}
```

2FIGURA 23.23 - Classe de exceção personalizada FONTE: Barnes (2004, p. 301).

Indicação de leitura

Nome do livro:: Programação Orientada a Objetos com Java. Uma introdução prática usando o BlueJ

Editora:: Pearson - Prentice Hall

Autor:: David J. Barnes e Michael Kölling

ISBN:: 857605012-9

Comentário: O livro apresenta muitos exemplos práticos acerca de diversos temas do Java, principalmente em relação ao Tratamento de Exceções, que pode ser um assunto complexo, referente às hierarquias e a como realizar o tratamento em um sistema que será utilizado por um usuário final.

UNIDADE III

Interface Gráfica Utilizando Swing

Ricardo de Almeida Rocha

A interface gráfica é muito utilizada em sistemas que terão o uso de usuários finais, facilitando a interação do usuário com o sistema. Esta unidade aborda o tópico de interfaces gráficas, explorando a utilidade e a forma como os principais componentes do Swing são implementados. Todo componente tem eventos que são os responsáveis por realizarem a interação da interface gráfica com o código que contém as regras de negócio do sistema.

Toda interface gráfica implementada em Java utiliza gerenciadores de layout que determinam como os componentes são distribuídos e organizados em uma janela. O Swing tem diversos gerenciadores nativos, porém estudaremos os mais utilizados; além dos gerenciadores de layout nativos, é possível o desenvolvedor criar um próprio ou utilizar gerenciadores de terceiros.

Por fim, é realizado um estudo analisando as duas APIs de componentes gráficos disponíveis no Java, o AWT, que, atualmente, não é mais utilizado, e o Swing, que é o sucessor do AWT.

Interface gráfica na Orientação a Objetos

A interface gráfica é o que realiza a interação do sistema com o usuário. A necessidade de uma interface gráfica surgiu pois os aplicativos de linha de comando eram fracos, inflexíveis e pouco amigáveis, o que tornava difícil a interação entre usuário e sistema (BATES, 2005).

O Java suporta duas bibliotecas gráficas estritamente para o *desktop*, AWT e Swing, e a biblioteca gráfica do JavaFX, que pode ser utilizada tanto para *desktop* quanto para aplicações web. A biblioteca AWT foi a primeira para interfaces gráficas a ser disponibilizada no Java, porém, a partir do Java 1.2, foi substituída pelo Swing (CAELUM 1, on-line).

O JavaFX surgiu com o "boom" das aplicações web, que deixaram de ser apenas páginas estáticas e passaram a introduzir componentes complexos que aumentavam a interatividade do usuário com a aplicação. Essas aplicações eram chamadas de RIA (*Rich Internet Applications* - Aplicações de Internet Ricas); as linguagens mais utilizadas eram Adobe Flex, Silverlight e o JavaFX. Assim como as outras duas, a codificação das interfaces em JavaFX ocorre por linguagem de marcação baseada no XML, chamada de FXML. Como é uma linguagem que se iniciou na web, apresenta algumas características interessantes herdadas de aplicações web, como a customização da aparência dos componentes mediante o CSS, que é amplamente utilizado na web. Como o JavaFX é codificado pela API Java, a sua interação com os componentes nativos do Java é mais prática, facilitando a utilização de modelos que separam as interfaces das regras de negócio.

As bibliotecas AWT e Swing são bibliotecas gráficas oficiais e disponibilizadas em qualquer JRE ou JDK. Também, existem bibliotecas gráficas de terceiros, dentre essas, a que mais se destaca é o SWT, criado pela IBM e utilizado por padrão no Eclipse (CAELUM 1, on-line).

A partir do Swing, as interfaces gráficas do Java são padronizadas, independente do ambiente em que estão rodando, aumentando sua portabilidade. O ***look-and-feel*** (a interface propriamente dita) não é modificado em relação ao sistema operacional (Windows, Linux, Mac etc.), com isso, as aplicações terão sempre a mesma interface (cores, tamanhos etc.) (CAELUM 1, on-line).

O *look-and-feel* é a "cara" da aplicação, composta por todos os detalhes que estão sendo exibidos, como cores, tamanho, formatos, dentre outros. O Java tem um *look-and-feel* próprio e também oferece outras opções, como os *look-and-feel* nativos do Windows e OS X da Mac, Motif, GTK e Nimbus. Além desses, é possível utilizar *look-and-feel* de terceiros. A Figura 3.1 mostra alguns *look-and-feel* utilizados na mesma aplicação.

Assim como todas as outras classes no Java, as classes de interface gráfica são tratados como objetos, ou seja, para a criação, exibição e interação de uma interface gráfica são utilizadas instâncias de objetos implementados para representar a interface gráfica.

Uma janela principal de um programa é a classe JFrame (ou uma subclasse que estenda o JFrame) instanciado por um objeto, e esse objeto é utilizado no método *main*.

Qualquer componente gráfico na API do Swing é uma classe que está no pacote javax.Swing. e é instanciada dentro do JFrame e de qualquer outro componente gráfico que será utilizado no sistema. As classes de componentes que iniciam com a letra "J" fazem parte do pacote Swing, enquanto os componentes que não iniciam com "J" pertencem ao pacote AWT.

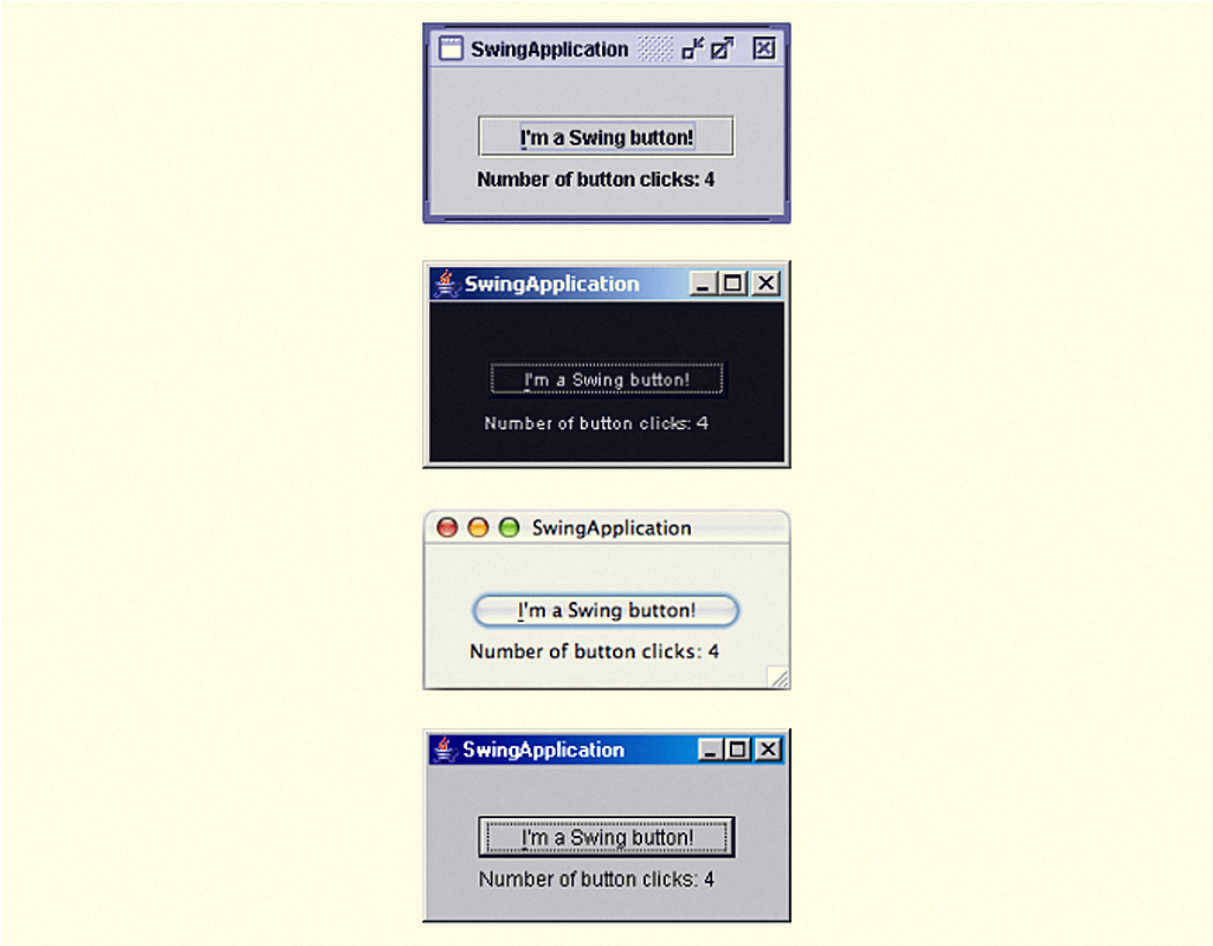
Componentes

A programação de uma interface gráfica é realizada por meio de componentes que são oferecidos pela biblioteca, como botões, campos para entrada de texto, tabelas, janelas, abas, dentre outros. A biblioteca de componentes do Swing está no pacote javax.swing.

JFrame e JPanel

O JFrame é o componente que representa a janela que estará visível na tela. É nele que serão inseridos os elementos da interface, como botões, campos de texto, caixa de seleção etc. Como padrão das janelas de sistema operacional, ele terá os botões de ação para Minimizar, Maximizar e Fechar a janela (BATES, 2005).

Como citado anteriormente e mostrado na Figura 3.1, o JFrame terá as características (*look-and-feel*) referentes ao sistema operacional que está sendo executado, ou então outra customizável. Os componentes utilizados para as interfaces da Figura 3.1 são os mesmos, mas a forma como eles são desenhados para serem apresentados ao usuário é baseada na configuração do *look-and-feel*.



3FIGURA 1.19 - Look-and-feel do Java FONTE: CAELUM 1, *on-line*.

O Swing disponibiliza diversos tipos de componentes a serem adicionados ao JFrame, como JButton, JRadioButton, JCheckBox, JLabel, JTextField, JTable, dentre outros.

O JPanel é um painel que agrupa os componentes, organizadamente, que estarão em determinada tela. O JPanel é adicionado a um JFrame, e os componentes serão adicionados a esse painel. O painel é utilizado nos gerenciadores de layout que veremos na sequência.

A Figura 3.2 mostra a utilização de um painel (JPanel) em uma janela (JFrame) e a sua interface gráfica quando executada. Na execução da interface gráfica, é possível verificar que o JPanel não tem uma visualização gráfica, servindo como

um container "invisível" para adicionar outros componentes, enquanto o JFrame é a janela da aplicação em si. A linha 21 adiciona o painel para a janela da aplicação, e os componentes são adicionados no painel nas linhas 24 e 28.

```java
public class PanelFrame extends JFrame {

    JPanel panel;
    JButton button;
    JTextField textField;

    public PanelFrame() {
        super("Panel no JFrame");
        setLayout(new FlowLayout());

        panel = new JPanel();
        add(panel);

        button = new JButton("Botão");
        panel.add(button);

        textField = new JTextField(10);
        textField.setText("Olá");
        panel.add(textField);
    }

    public static void main(String args[]) {
        PanelFrame pf = new PanelFrame();
        pf.setDefaultCloseOperation(EXIT_ON_CLOSE);
        pf.setSize(300, 300);
        pf.setVisible(true);
    }
}
```

3FIGURA 2.19 - Utilização de JPanel em um JFrame FONTE: NetBeans IDE.

JOptionPane

Como na maioria dos programas com interface gráfica, e ao contrário do que utilizamos até o momento, os dados são informados por meio de caixas de diálogo (ou somente diálogos), realizando a interação entre o usuário e o sistema. De modo geral, as caixas de diálogos são janelas que os programas utilizam para exibir mensagens ao usuário ou obter informação para o sistema. Os diálogos são exibidos por métodos estáticos da classe JOptionPane, como mostrado no trecho de código a seguir (DEITEL, 2010):

```
String numero = JOptionPane.showInputDialog("Insira o primeiro número");
```

Os dados informados pelo usuário por meio de uma caixa de diálogo são atribuídos a uma variável String, então, deve ser realizada a conversão para o tipo desejado (no exemplo anterior, seria realizada a conversão para inteiro), tomando cuidado com as possíveis exceções no momento da conversão.

O JOptionPane pode ser utilizado para exibir mensagens, e não apenas receber dados do usuário. As mensagens são exibidas pelo método .showMessageDialog(null, mensagem, titulo, JOptionPane.PLAIN_MESSAGE). Esse método tem quatro parâmetros: o primeiro refere-se ao lugar para posicionar o diálogo, caso seja informado *null*, a caixa será posicionada no centro da tela; o segundo é a mensagem que será exibida; o terceiro é o título da caixa de diálogo; o quarto é uma constante que indica o tipo de mensagem e se, junto com essa mensagem, será exibido algum ícone de mensagem. Essa é uma sobrecarga desse método, porém ele tem outras, que devem ser vistas consultando a API do Java.

A Figura 3.3 mostra as constantes e os ícones que serão exibidos.

O JOptionPane é uma caixa de diálogo **modal**, ou seja, enquanto ela estiver visível, não é possível o usuário interagir com outra parte do programa até que a caixa de diálogo seja fechada, pelos botões OK, Cancel ou outros disponíveis, dependendo da caixa de diálogo exibida (DEITEL, 2010).

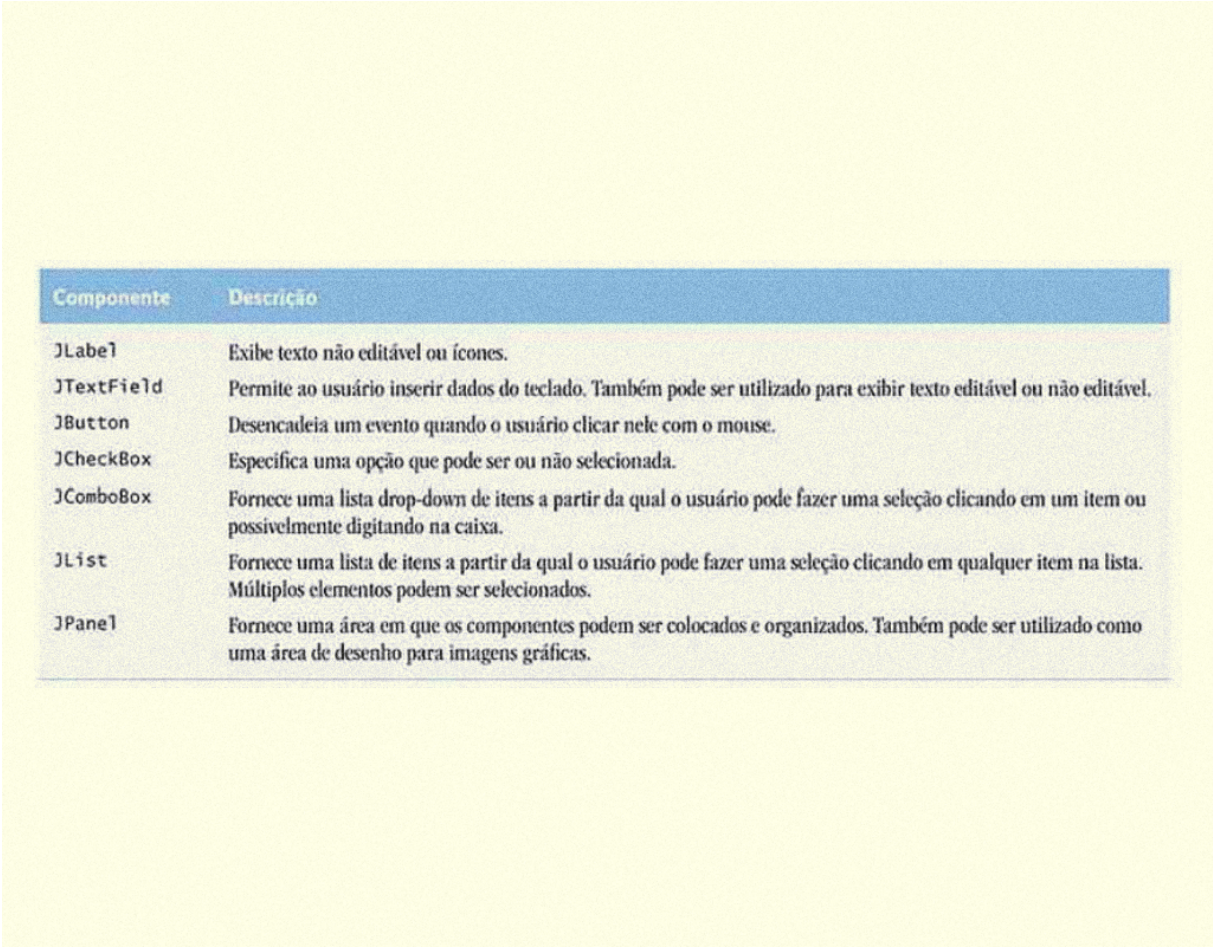
Tipo de diálogo de mensagem	Ícone	Descrição
ERROR_MESSAGE		Indica um erro ao usuário.
INFORMATION_MESSAGE		Indica uma mensagem informativa ao usuário.
WARNING_MESSAGE		Alerta o usuário de um potencial problema.
QUESTION_MESSAGE		Propõe uma questão ao usuário. Normalmente, esse diálogo exige uma resposta, como clicar em um botão Yes ou No.
PLAIN_MESSAGE	Nenhum ícone	Um diálogo que contém uma mensagem, mas nenhum ícone.

3FIGURA 3.19 - Constantes JOptionPane FONTE: Deitel (2010, p. 423).

Componentes GUI do Swing

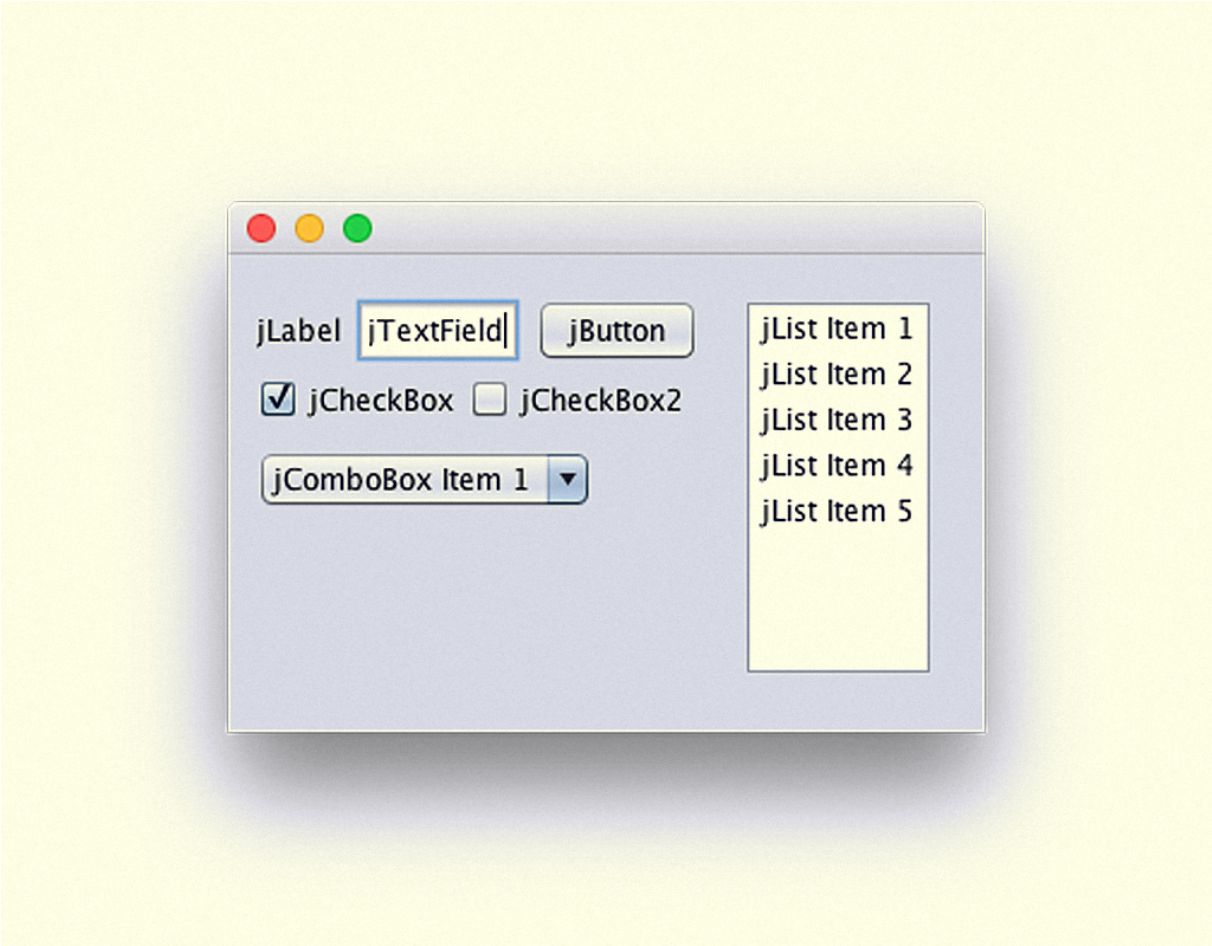
Além do JOptionPane, o Swing tem vários componentes, alguns mais robustos e com utilização mais prática, por exemplo, muitas vezes, não é necessária uma janela em modal para receber dados do usuário, e sim um campo de texto disposto na janela da aplicação, por meio da utilização de um JTextField.

A Figura 3.4 exibe alguns dos componentes básicos do Swing, e a Figura 3.5 mostra como eles são apresentados quando a aplicação está em execução.



Componente	Descrição
JLabel	Exibe texto não editável ou ícones.
JTextField	Permite ao usuário inserir dados do teclado. Também pode ser utilizado para exibir texto editável ou não editável.
JButton	Desencadeia um evento quando o usuário clicar nele com o mouse.
JCheckBox	Especifica uma opção que pode ser ou não selecionada.
JComboBox	Fornece uma lista drop-down de itens a partir da qual o usuário pode fazer uma seleção clicando em um item ou possivelmente digitando na caixa.
JList	Fornece uma lista de itens a partir da qual o usuário pode fazer uma seleção clicando em qualquer item na lista. Múltiplos elementos podem ser selecionados.
JPanel	Fornece uma área em que os componentes podem ser colocados e organizados. Também pode ser utilizado como uma área de desenho para imagens gráficas.

3FIGURA 4.19 - Componentes básicos do Swing FONTE: Deitel (2010, p. 423).



3FIGURA 5.19 - Exibição dos componentes básicos do Swing FONTE: NetBeans IDE.

JLabel e JTextField

O JLabel é um rótulo que tem como objetivo indicar a finalidade de um componente. O conteúdo exibido pelo JLabel é somente leitura, ou seja, não é editável pelo usuário. O JLabel pode exibir um texto, um ícone ou um texto e um ícone (DEITEL, 2010).

Para utilizar um ícone no label, deve-se utilizar um objeto da classe ImageIcon(pacote javax.swing) que aceita diversos tipos de imagens, sendo os mais utilizados GIF (Graphics Interchange Format), PNG (Portable Network Graphics) e JPEG (Joint Photographic Experts Group). Mais informações a respeito são encontradas nos livros de indicação de leitura.

O JTextField é um componente que oferece a opção de inserir um texto em uma única linha pela interface gráfica, da mesma forma que era feito pela linha de texto. O JTextField estende da classe JTextComponent, que fornece diversos recursos para componentes com base em texto no Swing. O componente JPasswordField é derivado do JTextField e trabalha especificamente com senhas, ocultando os caracteres digitados, ao contrário do JTextField, que deixa esses caracteres visíveis (DEITEL, 2010).

O conteúdo informado pelo usuário é atribuído a uma variável do tipo String (pois estamos trabalhando com texto) mediante o método .getText() que vem da superclasse JComponent.

JButton

O componente JButton é um botão de comando que, ao ser clicado, é acionada uma ação específica. O botão tem um rótulo (label) que deve identificar de modo único cada botão na interface gráfica.

A funcionalidade de um botão é, especificamente, acionar um evento que será tratado para o objetivo daquele botão. A classe JButton tem como superclasse JAbstractButton, que também é superclasse dos botões de estado (DEITEL, 2010).

Os eventos gerados por um botão são do tipo ActionEvents. Esses eventos serão abordados na seção de Tratamento de Eventos.

JCheckBox e JRadioButton

Os componentes JCheckBox, JToggleButton e JRadioButton são chamados de botões de estado, pois têm valores de ativado/desativado ou verdadeiro/falso. As classes JCheckBox e JRadioButton são derivadas de JToggleButton, e todos são herdados de JAbstractButton (DEITEL, 2010).

O JCheckBox são caixas de multisseleção, ou seja, dentre as opções que são listadas, pode ter qualquer combinação de opções selecionadas, desde nenhuma até todas as opções. Os estados das checkboxes são controlados por meio de variáveis booleanas, ou seja, quando não estão marcados, estão com o valor falso, quando estão marcados, viram verdadeiros.

O JRadioButton é chamado de botões de opção e é exibido como um grupo em que apenas um botão pode ser selecionado por vez (DEITEL, 2010). Ao contrário do JCheckBox, que poderia ter diversas seleções, o Radio Button assume apenas um estado por vez, porém a sua semelhança com o CheckBox diz respeito ao fato de ambos terem dois estados, selecionados ou não.

Ao adicionar o Radio Button a um projeto, é necessário utilizá-lo juntamente com a classe JButtonGroup, que é responsável por realizar o controle das opções selecionadas.

Ambos os botões geram um evento que indica a mudança de estado do botão. Esse evento é lançado quando uma opção que está desmarcada é clicada, tornando-se selecionada, ou, ao contrário, quando uma opção é desmarcada, gerando o evento itemStateChanged.

JList

O componente JList caracteriza uma lista com uma série de itens, na qual o usuário pode selecionar um ou mais itens. Essa classe estende diretamente a classe JComponent (DEITEL, 2010).

A classe JList suporta listas com única seleção, nas quais apenas um item é selecionado, e lista com múltiplas seleções, em que diversos valores podem ser selecionados. O tipo de seleção que a lista terá é definido pelo método .setSelectionMode(), que recebe uma constante como parâmetro. A constante ListSelectionModel.SINGLE_SELECTION especifica que será uma lista de única seleção, a constante ListSelectionModel.SINGLE_INTERVAL_SELECTION indica

que a seleção será de vários itens em um intervalo, e a constante ListSelectionModel.MULTIPLE_INTERVAL_SELECTOIN permite que a lista se comporte com múltipla seleção sem restrição (DEITEL, 2010).

A lista dispara um evento no momento em que ocorre a seleção de um item, disparando um evento do tipo ListSelectionEvent, executando ou invocando o método valueChanged sempre que o valor da seleção é alterado.

O construtor da JList recebe um array de Objects em que contém os itens que serão exibidos na lista. O método getSelectedIndex() retorna um número inteiro que indica o índice do item que foi selecionado na lista, quando a lista permite somente uma seleção. Quando é utilizada uma lista com múltiplas seleções, os valores selecionados são retornados em uma List, e não mais somente o índice, por meio do método getSelectedValuesList().

Fique por dentro

Mais informações acerca dos componentes gráficos do Swing podem ser encontradas nos documentos de tutoriais do Java no link a seguir. Esse link contém dados a respeito de todos os componentes do Swing, como a utilização, os métodos, os eventos e todas as características.

Disponível em: **https://docs.oracle.com <https://docs.oracle.com/javase/tutorial/uiswing/components/>** .

Tratamento de eventos

Quando ocorre uma ação do usuário sobre algum componente da interface gráfica do sistema, um evento acontece. Os componentes podem ter diversos eventos relacionados, ou seja, diversas situações que ocorrem em determinado componente podem acionar os eventos.

Segundo Bates (2005, p. 254), "***[...] a origem de um evento é um objeto que pode converter ações do usuário (um clique com o mouse, o pressionamento de uma tecla, o fechamento de uma janela) em eventos***". Ou seja, a grande maioria da interação do usuário com a interface gráfica pode gerar eventos. Como citado anteriormente, os componentes da interface gráfica com os quais o usuário está interagindo são objetos, assim como os eventos que são gerados por meio dessa interação.

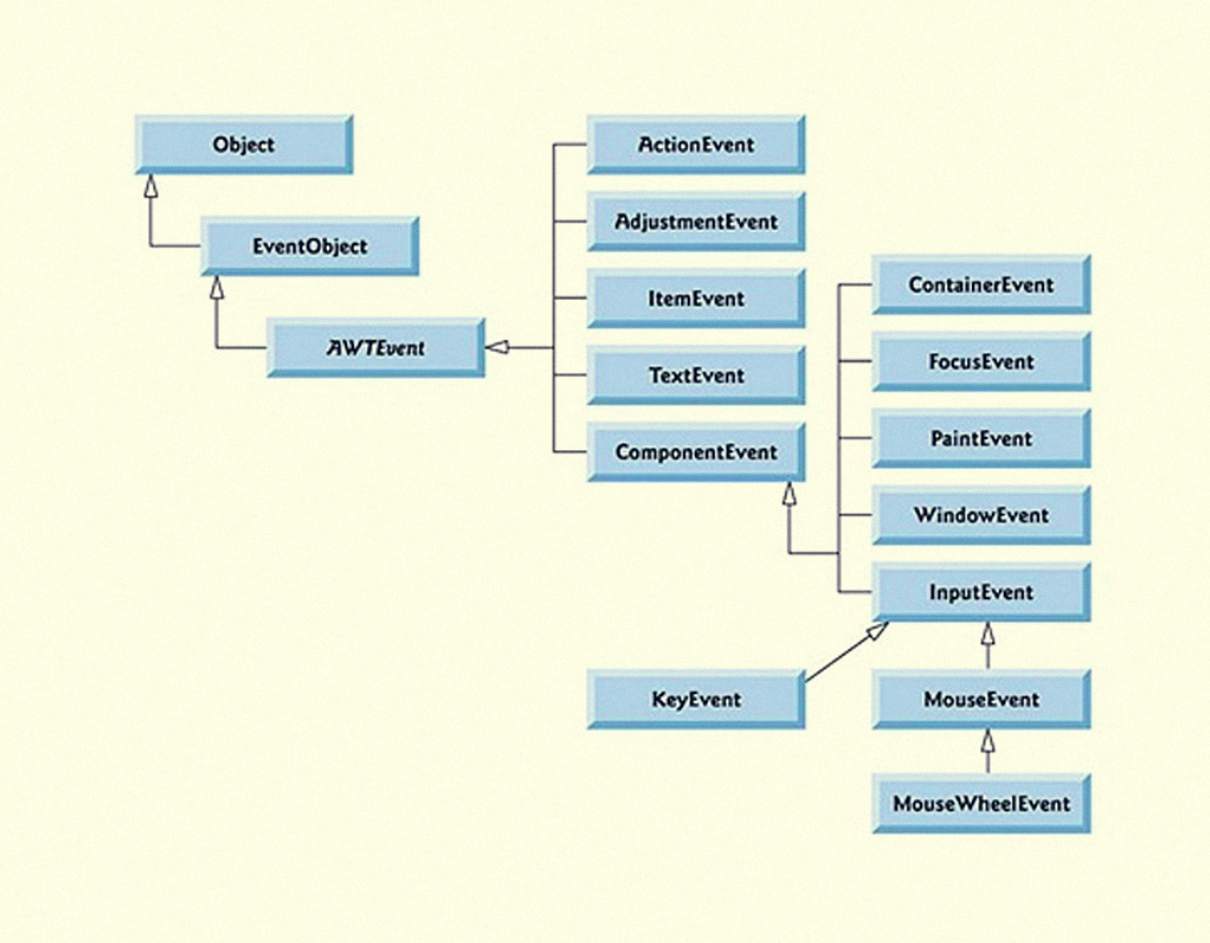
Para identificar que um evento ocorreu, por exemplo, que um botão foi clicado, são utilizados *listeners* (tradução livre: ouvintes) que ficam monitorando os eventos recebidos por um componente gráfico. A cada evento recebido pelo componente, é verificado se há um *listener* vinculado a esse evento, ou seja, se há algum *listener* ouvindo esse evento. Caso haja, o botão executa o método relacionado ao evento que esse *listener* está ouvindo.

Os *listeners* são implementados, por padrão em Java, utilizando o sufixo *Handler* (tradução livre: manipulador, mas pode ser interpretado como controlador), por exemplo, um *listener* para um clique de um botão (JButton) pode ser chamado de ButtonHandler.

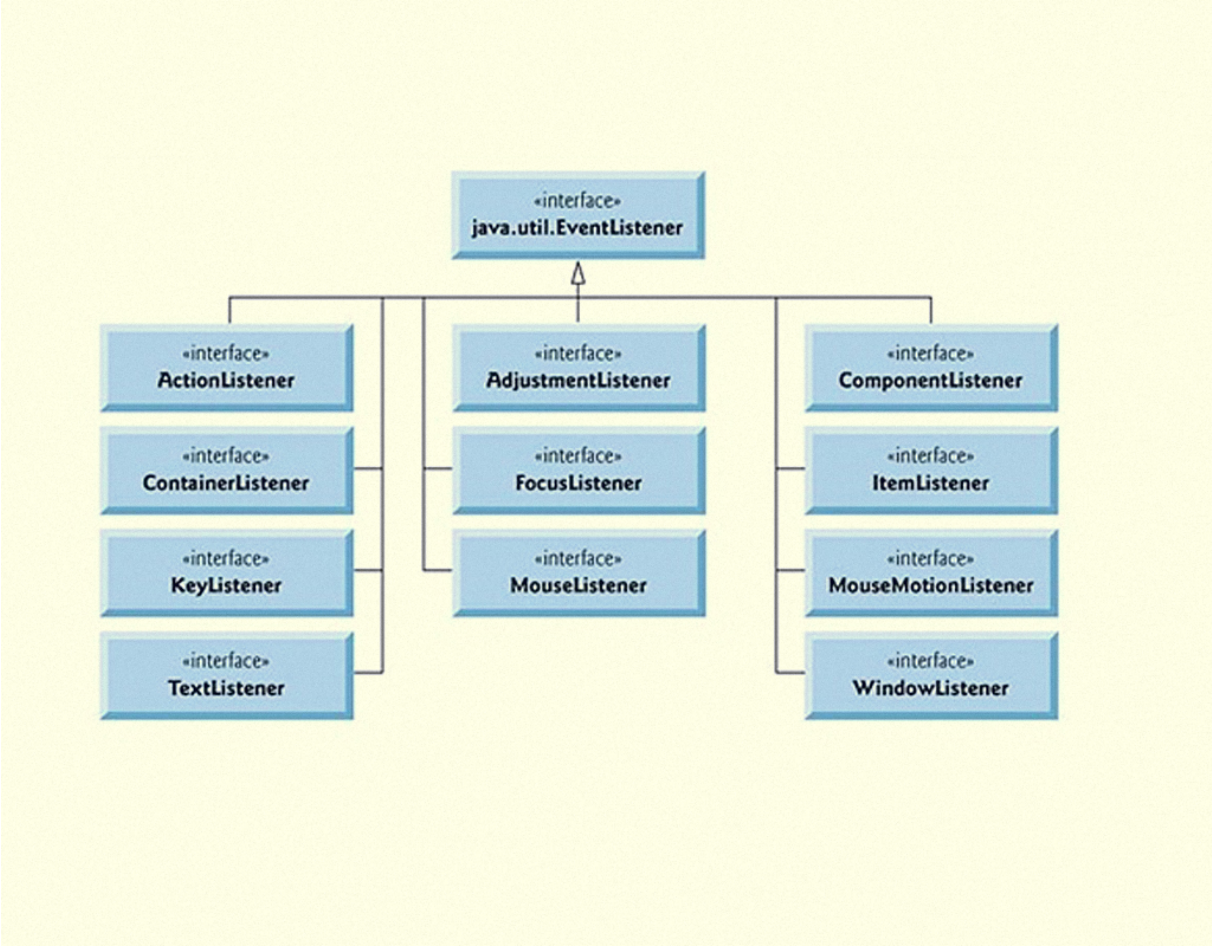
Lembrando de que estamos em Java, então, os botões são objetos, assim como os eventos também são objetos e, consequentemente, os *listeners* também serão objetos, ou seja, será criada uma classe para cada *listener* desejado. Os *listeners* são interfaces, dessa forma, as classes criadas para os *listeners* deverão implementar determinados métodos para que a manipulação de eventos ocorra corretamente.

A Figura 3.6 mostra como é a hierarquia de eventos do pacote java.awt.event, na qual as classes de eventos estendem a classe AWTEvent. Esses tipos de eventos são utilizados tanto com componentes AWT quanto com componentes Swing. Alguns eventos específicos de componentes Swing estão no pacote javax.swing.event (DEITEL, 2010).

Para cada tipo de evento, existe uma classe de *listener* de eventos correspondente que implementa uma, ou mais, interface dos *listeners* desse evento. A Figura 3.7 exibe algumas das interfaces de *listeners* do pacote java.awt.event, que são as interfaces que deverão ser implementadas quando criado um novo *listener* para um determinado evento (DEITEL, 2010).



3FIGURA 6.19 - Hierarquia de eventos FONTE: Deitel (2010, p. 433).



3FIGURA 7.19 - Hierarquia de listeners FONTE: Deitel (2010, p. 434).

Considerando um botão JButton como exemplo, ao ser clicado, são disparados eventos, e um desses eventos é um objeto da classe java.awt.event.ActionEvent. Quando o ActionEvent é recebido pelo JButton, é verificado se existe algum *listener* relacionado a esse evento (o JButton permite adicionar *listeners* para tratar um ActionEvent por meio do método addActionListener, que recebe como parâmetro um objeto do tipo java.awt.ActionListener), então, o JButton passará o controle para o(s) ActionListener(s) adicionado(s) para tratar o evento.

Tratamento de eventos nos componentes Swing

Nesta seção, verificaremos como realizar o tratamento de alguns eventos nos componentes do Swing introduzidos anteriormente.

JTextField

Como citado anteriormente, os componentes de interface gráficas podem gerar diversos eventos, e cada evento pode ser processado pelo *handler* (*listener*) apropriado para o evento. No componente JTextField, quando o usuário pressiona a tecla Enter, é disparado um evento do tipo ActionEvent, que é tratado por um objeto que implementa a interface ActionListener (DEITEL, 2010).

O componente JPasswordField, que trabalha especificamente com senhas, estende o JTextField; este tem as mesmas características para tratamento de eventos.

Um exemplo para tratar eventos é exibido na Figura 3.8, utilizando um JTextField.

```
19  public class TFFrame extends JFrame {
20
21      private JTextField textField1;
22
23      public TFFrame() {
24          super("Eventos no TextField");
25          setLayout(new FlowLayout());
26
27          textField1 = new JTextField(20);
28          add(textField1);
29
30          TFHandler tfHandler = new TFHandler();
31          textField1.addActionListener(tfHandler);
32      }
33
34      private class TFHandler implements ActionListener {
35
36          @Override
37          public void actionPerformed(ActionEvent e) {
38              String s = "";
39
40              if (e.getSource() == textField1)
41                  s = String.format("textField1 %s", e.getActionCommand());
42
43              JOptionPane.showMessageDialog(null, s);
44          }
45      }
46
47      public static void main(String args[]) {
48          TFFrame tfFrame = new TFFrame();
49          tfFrame.setDefaultCloseOperation(EXIT_ON_CLOSE);
50          tfFrame.setSize(300, 300);
51          tfFrame.setVisible(true);
52      }
53  }
```

3FIGURA 8.19 - Tratamento de eventos em um JTextField FONTE: adaptada de Deitel (2010, p. 429).

Nesse exemplo, quando é pressionada a tecla Enter no componente JTextField, a variável *string* **s** recebe o nome do componente ("textField1") e a mensagem que foi digitada no componente, que é atribuída ao formatador "%s" (linha 41). Essa variável **s** é exibida em um JOptionPane na linha 43.

Ao analisarmos a Figura 3.8, vemos que, na linha 19, é definida a classe TFFrame, que se estende do JFrame, caracterizando que haverá componentes no JFrame. As linhas 24 e 25 chamam, respectivamente, o construtor da classe pai JFrame pelo super, passando por argumento o título da janela, e o método para definir o gerenciador de layout que estaremos utilizando, no caso, FlowLayout (veremos os Gerenciadores de Layout mais à frente).

A linha 27 cria e instancia um novo objeto JTextField, informando, no seu construtor, o número 20, que indica que o campo digitável do textField aceitará, no máximo, 20 caracteres.

Na linha 30, começa-se a tratar o evento que estará na nossa janela. Nela, instanciamos um novo objeto do tipo TFHandler, que é definido na linha 34, e atribuímos esse objeto (que é um *listener* de eventos) para o objeto textField1 pelo método .addActionListener(). Esse método indica ao componente textField1 que, quando ocorrer um evento do tipo ActionEvent, será chamado o handler TFHandler para tratar do evento.

O *handler* do evento é implementado nas linhas 34-43. Podemos ver que é uma classe privada, dentro da classe TFFrame, ou seja, essa classe poderá ser usada somente dentro do escopo do TFFrame. Caso estivermos em outra classe, não poderemos utilizar esse objeto. Essa classe privada implementa a interface ActionListener, que fará o objeto ser um *listener* do evento ActionEvent; então, pela definição de interface, essa classe deverá implementar os métodos definidos na interface. Nesse caso, é obrigatória a implementação do método actionPerformed(ActionEvent e), que é o método que será invocado quando o evento ocorrer. Observamos que o método está com a anotação @Override, indicando a sobreposição (sobrescrita) desse método.

A linha 40 faz uma comparação para verificar se o objeto que gerou o evento é o textField1, pois, caso tivéssemos mais que um textField na interface e eles tivessem como *listener* essa mesma classe, quando o evento fosse lançado pelos outros textField, esse método seria invocado. A linha 41 cria a *string* que será exibida no JOptionPane com o texto "textField1" e o conteúdo digitado no textField pelo método getActionCommand(), que está no evento ***e*** que é passado por argumento no método actionPerformed.

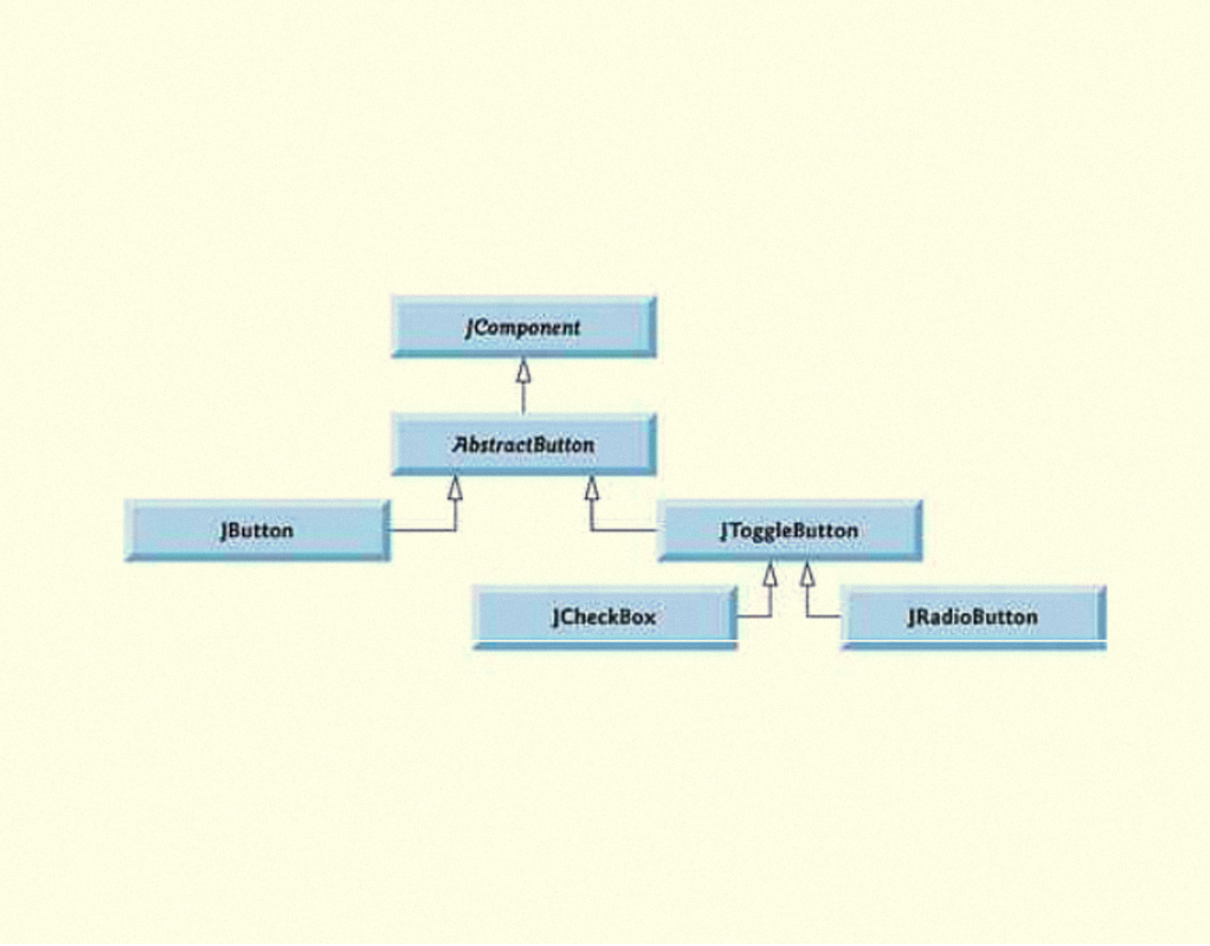
Nas linhas 47-52, é definida a classe *main* para rodar a aplicação, instanciando o objeto TFFrame, que é a janela principal da nossa interface, informando: que o comportamento padrão é terminar o programa ao fechar a janela, o seu tamanho

(.setSize()) e se a janela será visível.

JButton

Um botão é um dos componentes de uma interface gráfica que tem grande importância e aparece diversas vezes em uma GUI. Um botão pode ser de vários tipos, como JButton que é o botão tradicional e JToggleButton, que é a superclasse de JCheckBox e JRadioButton, que são caixas de múltipla escolha e única escolha. Ambas as classes, JButton e JToggleButton, derivam da classe AbstractButton, que declara os recursos comuns dos botões do Swing (DEITEL, 2010).

A Figura 3.9 mostra a hierarquia das classes de botões.



3FIGURA 9.19 - Hierarquia do botão no Swing FONTE: Deitel (2010, p. 436).

O JButton gera eventos do tipo ActionEvent, assim como o JTextField, portanto, qualquer objeto que implemente o listener ActionListener pode tratar do evento.

A Figura 3.10 mostra o evento ActionEvent de um JButton sendo tratado, porém, para manipular o evento, o handle é criado em uma classe anônima, ao contrário do exemplo utilizado no JTextField, em que foi criada uma classe privada.

```
18  public class ButtonFrame extends JFrame {
19
20      JButton button;
21
22      public ButtonFrame() {
23          super("Eventos no JButton");
24          setLayout(new FlowLayout());
25
26          button = new JButton("Eu sou um botão");
27          add(button);
28
29          button.addActionListener(
30              new ActionListener()
31              {
32                  @Override
33                  public void actionPerformed(ActionEvent e) {
34                      button.setText("Fui clicado " + e.getActionCommand());
35                  }
36              }
37          );
38      }
39
40      public static void main(String args[]) {
41          ButtonFrame bf = new ButtonFrame();
42          bf.setDefaultCloseOperation(EXIT_ON_CLOSE);
43          bf.setSize(300, 300);
44          bf.setVisible(true);
45      }
46
47  }
```

3FIGURA 10.19 - Tratamento de eventos em um JButton FONTE: adaptada de Deitel (2010).

Como no exemplo do JTextField, o botão dispara um evento ActionEvent no momento em que é clicado, mas o tratamento do clique é realizado na classe anônima, que é definida no momento de instanciar o listener do evento (linha 30), como visto na Figura 3.10.

A vantagem de utilizar uma classe anônima é que ela restringe seu uso apenas para o escopo necessário, por exemplo, caso tivéssemos um *handler* que executasse um comando para fechar a aplicação quando um determinado botão fosse clicado e esse handler fosse uma classe normal, ele poderia ser utilizado incorretamente em outro componente, ocasionando o término da execução.

O escopo da classe anônima é bem restrito, sendo utilizado somente no local em que foi implementada, e a classe anônima tem acesso aos membros da sua classe de primeiro nível, que, no exemplo da Figura 3.10, é a classe ButtonFrame (DEITEL, 2010).

A desvantagem de uma classe anônima ocorre quando seu código começa a ficar muito extenso, com isso, dificulta a leitura do código, mesmo indentado.

¶ Reflita

Qual seria a forma de solucionar o problema citado anteriormente sem utilizar uma classe anônima, utilizando uma classe de Handler declarada normalmente?

No JButton, como em diversos outros componentes do Swing, é possível atribuir mais de um *listener* para o mesmo tipo de evento, ou seja, poderíamos atribuir outro *listener* para um ActionEvent, declarando uma nova classe que implementasse um

ActionListener. A execução dos *listeners* seria na ordem inversa em que for adicionado, ou seja, o funcionamento é uma pilha, os *listeners* vão sendo adicionados a uma pilha e, então, são executados à medida que são "removidos" da pilha.

Além dos eventos de clique, poderiam ser adicionados eventos com o mouse (MouseEvents), que são tratados com *listeners* que implementem a interface MouseListener. Poderiam ser tratados eventos quando o mouse "entrar" (focar) no botão, quando ele sair, quando pressionar, quando soltar. A utilização dos eventos do mouse depende de como funcionará a aplicação, mas, normalmente, é mais utilizado o evento de clique do botão. A API do Java tem informações acerca dos eventos que podem ocorrer.

JCheckBox e JRadioButton

Assim como o JButton, os componentes JCheckBox e JRadioButton são herdados da classe JAbstractButton, porém o JCheckBox e JRadionButton são componentes que mantêm o estado, ou seja, eles têm valores de ativado/desativado e verdadeiro/falso.

Quando é alterado o valor de uma das opções desses componentes, é disparado um evento ItemEvent, em que é possível verificar quais valores foram alterados e tratá-los mediante um objeto que implemente a interface ItemListener.

A diferença entre o CheckBox e o RadioButton é que o CheckBox permite que vários estados estejam assinalados ao mesmo tempo, porém o RadioButton é mutuamente exclusivo, ou seja, somente um estará assinalado, por isso, quando é utilizado JRadioButton, deve-se utilizar um agrupador ButtonGroup para realizar o relacionamento lógico entre os botões RadioButton.

O tratamento de eventos é similar ao JButton, porém, ao invés de implementar um ActionListener, é implementado um ItemListener, que é adicionado ao objeto pelo método .addItemListener().

JList

O JList é um componente que exibe listas, sejam elas de múltipla seleção, em que vários itens são selecionados, ou de única seleção, podendo selecionar apenas um item.

Nas listas de única seleção, é disparado o evento ListSelectionEvent quando um valor na lista é selecionado. A interface que implementa o *listener* referente a esse evento é o ListSelectionListener, sendo necessário implementar o método valueChanged(ListSelectionEvent event). Nesse método, é possível receber o valor selecionado na lista por meio do índice referente ao valor na lista.

A Figura 3.11 mostra a implementação de uma lista simples para exemplificar como é tratado o evento ListSelectionEvent e a utilização do valor escolhido.

```
23  public class ListFrame extends JFrame {
24
25      private JList lista;
26      private String[] direcoes = { "Direita", "Esquerda", "Frente", "Trás" };
27
28      public ListFrame() {
29          super("Teste da Lista");
            setLayout(new FlowLayout());
31
32          lista = new JList(direcoes);
33          lista.setVisibleRowCount(3);
34          lista.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
35
            add(new JScrollPane(lista));
37
38          lista.addListSelectionListener(
                new ListSelectionListener() {
40                  @Override
                    public void valueChanged(ListSelectionEvent e) {
42                      JOptionPane.showMessageDialog(null, direcoes[lista.getSelectedIndex()]);
43                  }
44              }
45          );
46      }
47
48      public static void main(String[] args) {
49          ListFrame listFrame = new ListFrame();
50          listFrame.setSize(300, 300);
51          listFrame.setDefaultCloseOperation(EXIT_ON_CLOSE);
52          listFrame.setVisible(true);
53      }
54  }
```

3FIGURA 11.19 - Tratamento de eventos em uma JList FONTE: adaptada de Deitel (2010, p. 445).

Ao realizarmos uma análise na Figura 3.11, pode-se ver que, na linha 26, definimos um *array* de *string* contendo os itens que irão popular a lista. Esses valores são *string*, mas poderíamos ter um outro *array* com os valores correspondentes à *string* da descrição do valor, por exemplo, o *array* de *string* que será exibido na lista seria o nome das cores "Azul", "Verde" etc., e um outro *array* com os valores correspondentes, como Colors.BLUE, Colors.GREEN e assim por diante. Esse array de string é atribuído para a lista na linha 32 por meio do construtor do JList.

O comportamento de seleção única da lista é definido na linha 34 pelo método setSelectionMode(ListSelectionMode.SINGLE_SELECTION).

A lista foi adicionada a um JScrollPane na linha 36, para que seja exibida a barra de rolagem, caso a lista tenha mais itens do que é possível exibir. Caso a lista não seja adicionada a um JScrollPane, ela exibirá, automaticamente, o maior número de itens possíveis, mesmo que o número de itens visíveis seja menor.

Na linha 39, é definido o listener do evento ListSelectionListener por meio de uma classe anônima que implementa o método valueChanged, que é disparado quando é escolhido um valor na lista. A linha 42 utiliza o índice do valor escolhido para exibir a mensagem a respeito de qual foi o valor selecionado.

Uma lista de múltipla seleção tem comportamento similar, porém, ao invés de retornar o índice do valor selecionado, é retornado um *array* de objetos com os valores selecionados.

A Figura 3.12 mostra o exemplo anterior implementado com uma lista de múltipla seleção.

```
21  public class ListFrame extends JFrame {
22
23      private JButton botaoSelecao;
24      private JList lista;
25      private JList listaSelecionada;
26      private String[] direcoes = { "Direita", "Esquerda", "Frente", "Trás" };
27
28      public ListFrame() {
29          super("Teste da Lista");
30          setLayout(new FlowLayout());
31
32          lista = new JList(direcoes);
33          lista.setVisibleRowCount(4);
34          lista.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
35
36          botaoSelecao = new JButton(">>>>");
37          listaSelecionada = new JList();
38
39          botaoSelecao.addActionListener(
40              new ActionListener() {
41                  @Override
42                  public void actionPerformed(ActionEvent e) {
43                      listaSelecionada.setListData(lista.getSelectedValuesList()
44                              .toArray());
45                  }
46              }
47          );
48
49          add(new JScrollPane(lista));
50          add(botaoSelecao);
51          add(new JScrollPane(listaSelecionada));
52      }
53
54      public static void main(String[] args) {
```

3FIGURA 12.19 - Tratamento de eventos em uma JList com múltipla seleção FONTE: adaptada de Deitel (2010, p. 447).

Na lista de múltipla seleção, não há um evento para indicar que o usuário selecionou múltiplos valores, por isso, é utilizado o evento de um botão, ou outro componente da GUI, que utilizará os itens selecionados para realizar alguma ação. A linha 34 define nossa lista como sendo de múltipla seleção, como podemos observar na Figura 3.12.

No caso do exemplo da Figura 3.12, o botão botaoSelecao realiza a cópia dos itens selecionados para uma outra lista (listaSelecionada), que é definida na linha 43, na execução do evento do clique do botão. Os dados selecionados são obtidos com o método getSelectedValuesList(), utilizando o método .toArray() para transformá-lo em *array*, e, então, inseridos na JList listaSelecionada pelo método setListData.

Antes do Java 7, era possível retornar os objetos selecionados por meio do método .getSelectedValues(), que retornava um array de Objects, porém esse método foi marcado como *deprecated* (que significa que o método não é mais utilizado) e substituído pelo getSelectedValuesList().

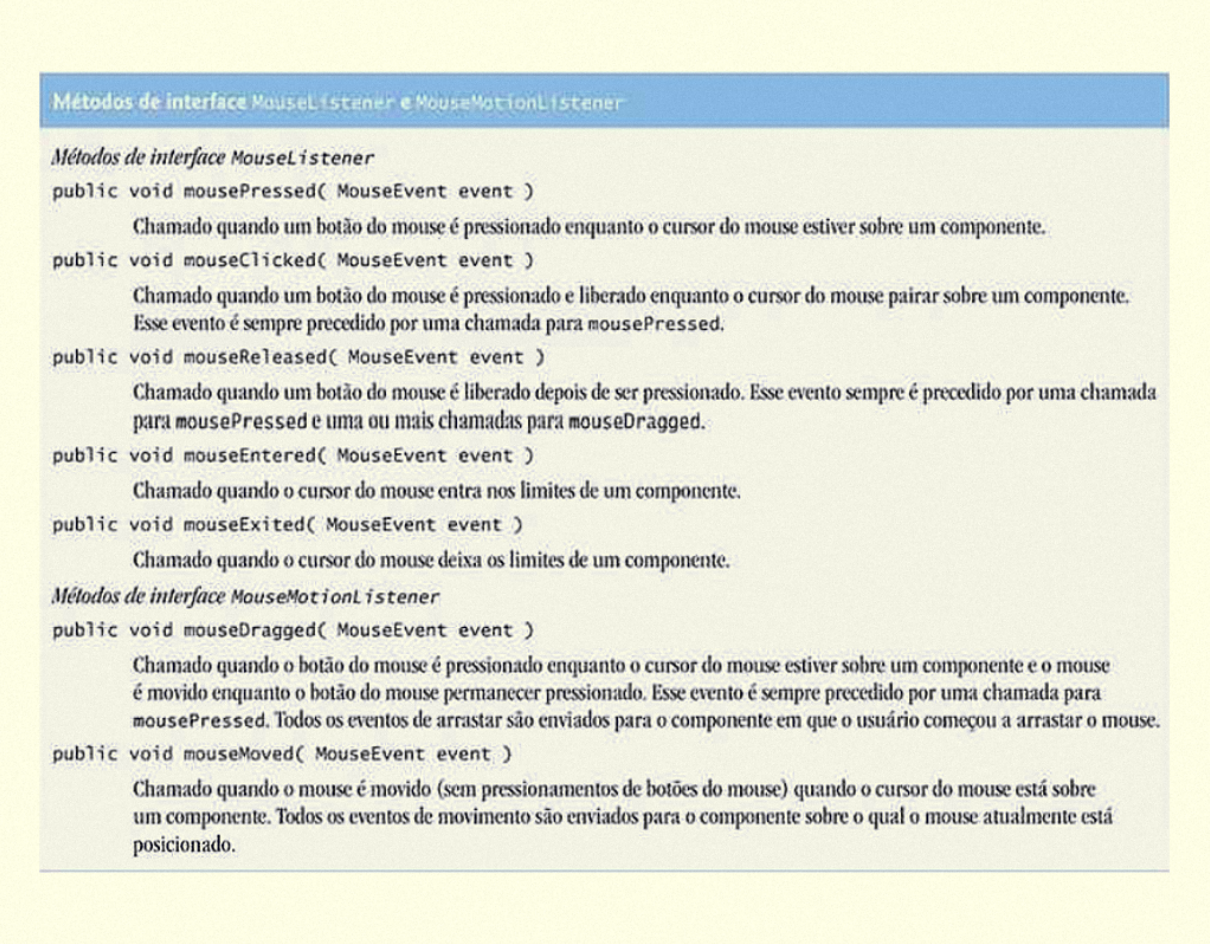
Eventos de mouse e teclado

O Swing também suporta os eventos disparados pelo movimento do mouse e por ações no teclado.

Evento de mouse

Os eventos de mouse, ou MouseEvents, são tratados pelas interfaces dos *listeners* específicos para o mouse, como MouseListener e MouseMotionListener.

A Figura 3.13 mostra os métodos que são definidos pelas interfaces dos *listeners*.



Métodos de interface MouseListener e MouseMotionListener

Métodos de interface MouseListener

`public void mousePressed( MouseEvent event )`

Chamado quando um botão do mouse é pressionado enquanto o cursor do mouse estiver sobre um componente.

`public void mouseClicked( MouseEvent event )`

Chamado quando um botão do mouse é pressionado e liberado enquanto o cursor do mouse pairar sobre um componente. Esse evento é sempre precedido por uma chamada para `mousePressed`.

`public void mouseReleased( MouseEvent event )`

Chamado quando um botão do mouse é liberado depois de ser pressionado. Esse evento sempre é precedido por uma chamada para `mousePressed` e uma ou mais chamadas para `mouseDragged`.

`public void mouseEntered( MouseEvent event )`

Chamado quando o cursor do mouse entra nos limites de um componente.

`public void mouseExited( MouseEvent event )`

Chamado quando o cursor do mouse deixa os limites de um componente.

Métodos de interface MouseMotionListener

`public void mouseDragged( MouseEvent event )`

Chamado quando o botão do mouse é pressionado enquanto o cursor do mouse estiver sobre um componente e o mouse é movido enquanto o botão do mouse permanecer pressionado. Esse evento é sempre precedido por uma chamada para `mousePressed`. Todos os eventos de arrastar são enviados para o componente em que o usuário começou a arrastar o mouse.

`public void mouseMoved( MouseEvent event )`

Chamado quando o mouse é movido (sem pressionamentos de botões do mouse) quando o cursor do mouse está sobre um componente. Todos os eventos de movimento são enviados para o componente sobre o qual o mouse atualmente está posicionado.

3FIGURA 13.19 - Listeners de eventos do mouse FONTE: Deitel (2010, p. 449).

Assim como os eventos definidos anteriormente, os eventos do mouse estão relacionados a um *handler* que implementa uma interface dos *listeners*. Esse *listener* é vinculado a um componente, desde um componente pequeno a um JPanel ou JFrame. Então, o método do *listener* é invocado quando o mouse realiza determinado evento no componente que está monitorando esse evento (DEITEL, 2010).

Os métodos de tratamento dos eventos do mouse recebem como argumento um objeto MouseEvent com as informações acerca de como ocorreu o evento, incluindo as coordenadas X e Y de onde está o ponteiro do mouse no momento do evento, que são medidas em relação ao canto superior esquerdo, que é chamado de 0,0 (DEITEL, 2010).

O Swing também contempla a interface MouseWheelListener, que trata as ações com a roda do mouse por meio do evento MouseWheelEvent (DEITEL, 2010).

Evento de teclado

Os eventos de teclado, que também podem ser chamados de eventos-chave (traduzido do inglês KeyEvents) são disparados quando uma tecla é pressionada e liberada.

As classes que implementam a interface KeyListener devem implementar os métodos keyPressed, keyReleased e keyTyped, significando, respectivamente, que a tecla foi pressionada, que a tecla foi liberada e qual a tecla que foi digitada, que não seja uma tecla de ação (por exemplo, Enter, teclas de seta, Home, End, PgUp, PgDown etc.) (DEITEL, 2010).

Todos os métodos citados anteriormente recebem um KeyEvent como argumento que contém informações acerca do evento e é uma subclasse de InputEvent, e o método keyReleased sempre é chamado depois de qualquer evento keyPressed ou keyTyped, pois, se a tecla for pressionada, ela será liberada em algum momento (DEITEL, 2010).

Durante os eventos de teclado, é possível verificar qual tecla foi pressionada por meio do método getKeyCode da classe KeyEvent, que retorna o código da tecla virtual pressionada. O método getKeyText transforma o código da tecla em uma *string*, contendo o nome da tecla (DEITEL, 2010).

¶ Fique por dentro

> A API da classe KeyEvent tem a relação de todos os códigos das teclas que são representadas por constantes. O endereço para consulta da API da classe KeyEvent do Java 7 é: **https://docs.oracle.com <https://docs.oracle.com/javase/7/docs/api/java/awt/event/KeyEvent.html>** .

A classe KeyEvent ainda tem o método isActionKey, que determina se a tecla pressionada é uma tecla normal ou uma tecla de ação. O método getModifiers da classe InputEvent, que é superclasse da KeyEvent, retorna se a tecla pressionada é uma tecla modificadora, como Alt, Shift e Ctrl (DEITEL, 2010).

Gerenciadores de *Layout*

O Java tem objetos que controlam a forma como os componentes são distribuídos na janela quando o *layout* é programado por linha de código.

Segundo Deitel (2010, p. 460),

> Os gerenciadores de layout organizam componentes GUI em um contêiner para propósitos de apresentação. (...) é utilizado para obter as capacidades de layout em vez de determinar a posição e o tamanho exatos de cada componente.

O objetivo final dos gerenciadores de *layout* é que o desenvolvedor tenha foco nas funcionalidades e nas capacidades dos componentes, deixando os gerenciadores se preocuparem com os detalhes básicos do *layout* (DEITEL, 2010).

Os gerenciadores de *layout* controlam os componentes que serão inseridos no componente ao qual está associado, por exemplo, se um frame tiver um painel e esse painel tiver um botão, o gerenciador de *layout* do painel controlará as características do botão, como tamanho e inserção do botão no local escolhido pelo gerenciador. O gerenciador de *layout* do *frame* controlará a inserção do painel (BATES, 2005).

O gerenciador de *layout* é utilizado apenas em componentes que contenham outros componentes, como um JFrame, um JPanel, enquanto um botão não terá um gerenciador de *layout*, pois não contém outros componentes, é um componente simples (BATES, 2005).

O gerenciador de *layout* controla os componentes que serão inseridos dentro do componente ao qual ele está relacionado, ou seja, se um painel tiver diversos elementos, o tamanho e o local desses elementos serão controlados pelo gerenciador de *layout* do painel, mesmo que esses elementos tenham seus próprios gerenciadores de *layout*. Já os elementos que estiverem dentro desses elementos serão inseridos de acordo com o gerenciador de *layout* do elemento que os contém (BATES, 2005).

Os gerenciadores de layout têm diversas abordagens, e cada componente container pode ter seu próprio gerenciador. A distribuição e o tamanho dos componentes vão variar da política de cada gerenciador.

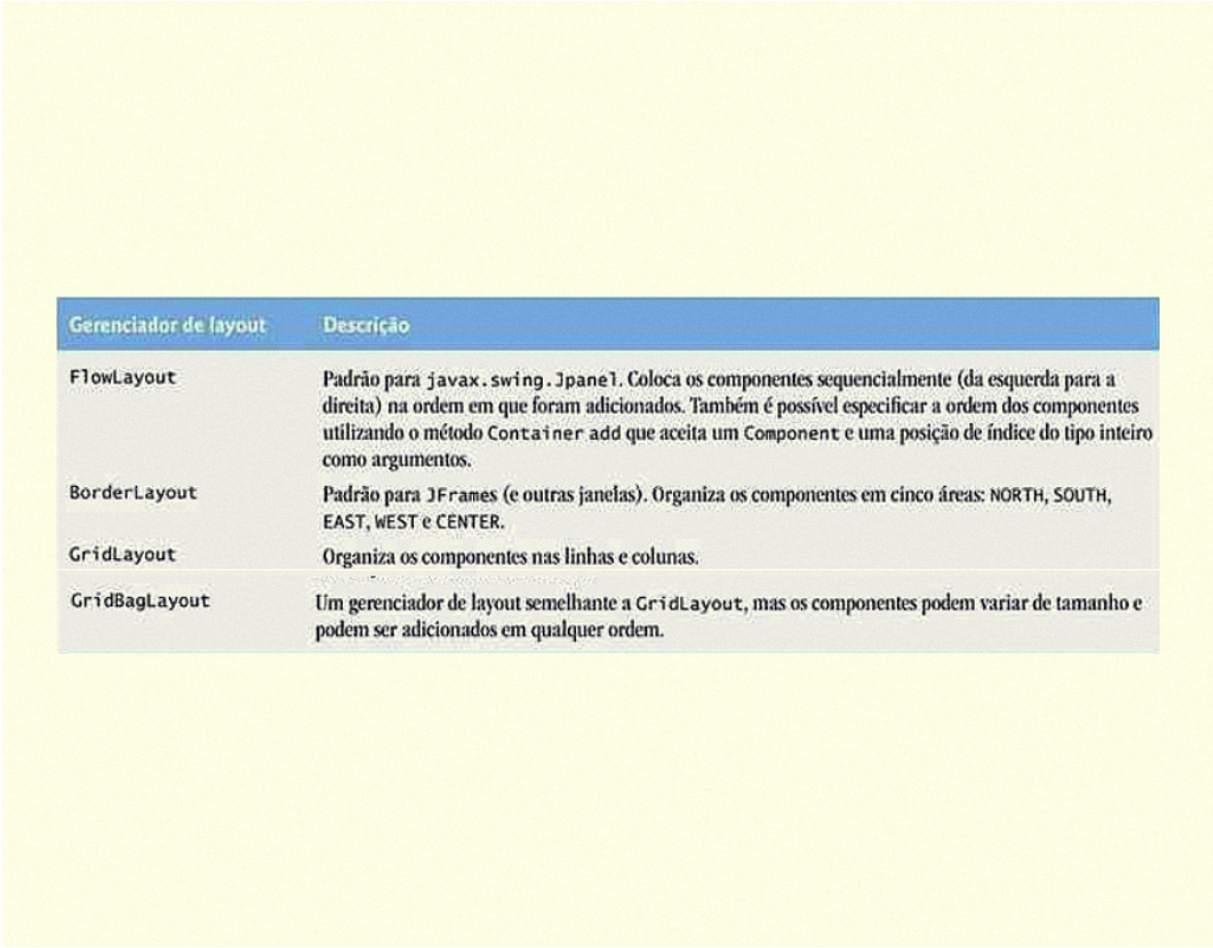
Todos os gerenciadores de *layout* implementam a interface LayoutManager do pacote java.awt. O *layout* é informado pelo método setLayout, da classe Container, que recebe por parâmetro um objeto que implementa a interface LayoutManager. O método setLayout é chamado dentro de um objeto que possa conter outros componentes, ou seja, que estenda a classe Container, por exemplo, JFrame, JPanel etc. (DEITEL, 2010).

Para dispor os componentes em uma interface gráfica, existem três formas distintas (DEITEL, 2010), descritas a seguir.

- **Posicionamento absoluto:** é a forma que fornece maior nível de controle sobre a aparência da interface gráfica. Quando o layout de um container é setado para *null*, é possível controlar a posição absoluta de cada componente da GUI em relação ao canto superior esquerdo, que é a posição (0,0), por meio dos métodos setSize, setLocation e/ou setBounds. É a programação da GUI mediante linha de código, o que pode ser trabalhoso.
- **Gerenciadores de Layout**: a distribuição dos componentes e o tamanho deles em uma GUI são automatizados pela política do gerenciador de *layout*, porém alguns detalhes e controle sobre os componentes são perdidos.
- **Programação visual por meio de IDE**: os IDEs fornecem ferramentas de design de interface gráfica que permitem arrastar e soltar o componente no local desejado. Os próprios IDEs geram o código Java que será executado ao criar a interface gráfica, e o tratamento de evento é feito por duplos cliques no componente. Os componentes podem ter suas características personalizadas por meio de caixas com as opções a serem customizadas.

Os gerenciadores de *layouts* mais conhecidos no Java são FlowLayout, BorderLayout, GridLayout e GridBagLayout, que são os gerenciadores que serão abordados neste material. Além desses, existem o GroupLayout, Box Layout; também, há gerenciadores de *layout* de terceiros. Esses gerenciadores de *layout* não serão abordados, porém serão disponibilizados locais para encontrar informações a respeito deles.

A Figura 3.14 mostra as características dos gerenciadores de layout que serão abordados.



Gerenciador de layout	Descrição
`FlowLayout`	Padrão para `javax.swing.Jpanel`. Coloca os componentes sequencialmente (da esquerda para a direita) na ordem em que foram adicionados. Também é possível especificar a ordem dos componentes utilizando o método `Container` add que aceita um `Component` e uma posição de índice do tipo inteiro como argumentos.
`BorderLayout`	Padrão para `JFrames` (e outras janelas). Organiza os componentes em cinco áreas: `NORTH`, `SOUTH`, `EAST`, `WEST` e `CENTER`.
`GridLayout`	Organiza os componentes nas linhas e colunas.
`GridBagLayout`	Um gerenciador de layout semelhante a `GridLayout`, mas os componentes podem variar de tamanho e podem ser adicionados em qualquer ordem.

3FIGURA 14.19 - Gerenciadores de layout FONTE: adaptada de Deitel (2010, p. 461 e 789).

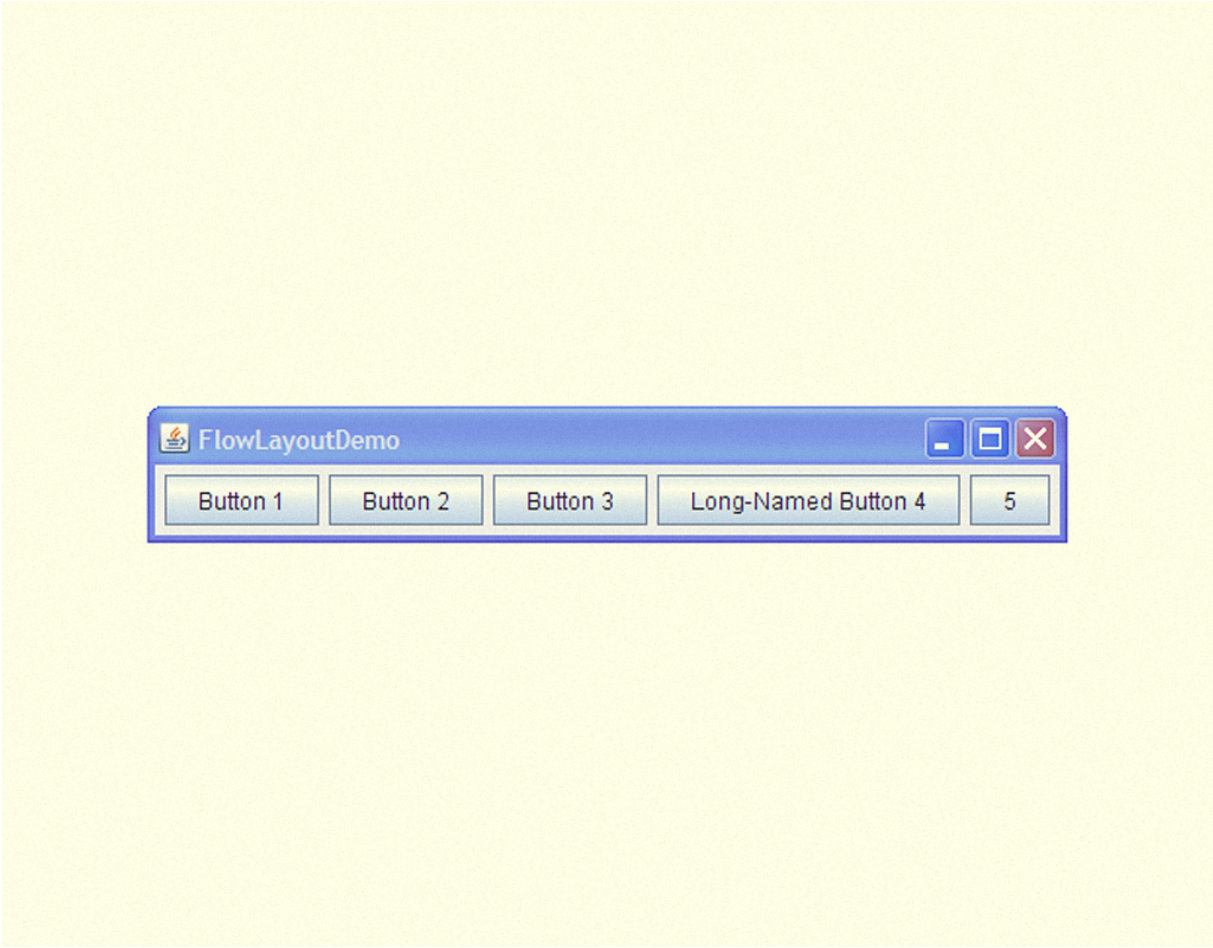
FlowLayout

O FlowLayout é o gerenciador de layout mais simples, tanto pela utilização quanto pelo funcionamento. Os componentes são colocados no container da esquerda para a direita na ordem em que são adicionados ao container (DEITEL, 2010). O FlowLayout tem mudança automática de linha como padrão, ou seja, se um componente não couber horizontalmente em uma linha, ele será inserido na linha de baixo. Os componentes podem ser alinhados à esquerda, à direita ou centralizados, que é o padrão. O gerenciador FlowLayout foi o gerenciador que foi utilizado nos exemplos de tratamento de eventos em interface gráfica da seção anterior.

Quando a janela é redimensionada, os componentes gráficos são afetados e o gerenciador reorganiza os componentes. Caso a janela tenha seu tamanho diminuído, o gerenciador jogará os componentes que não couberam na primeira linha para a segunda linha e assim sucessivamente. Caso a janela tenha seu tamanho aumentado, os componentes são alinhados conforme foi informado ao gerenciador e, se houver componentes em outras linhas além da primeira, eles serão organizados para uma linha acima, à medida que couberem no novo tamanho da linha.

O alinhamento dos componentes é setado pelo método .setAlignment, passando as constantes FlowLayout.CENTER para centralizado, FlowLayout.LEFT para alinhado à esquerda ou FlowLayout.RIGHT para alinhamento à direita.

A Figura 3.15 mostra a disposição de botões dentro de uma janela utilizando o FlowLayout.



3FIGURA 15.19 - FlowLayout FONTE: ORACLE.

Os botões foram distribuídos no fluxo, um após o outro, até o tamanho da janela. Caso o botão 5 não coubesse, ele teria sido deslocado para a linha seguinte.

BorderLayout

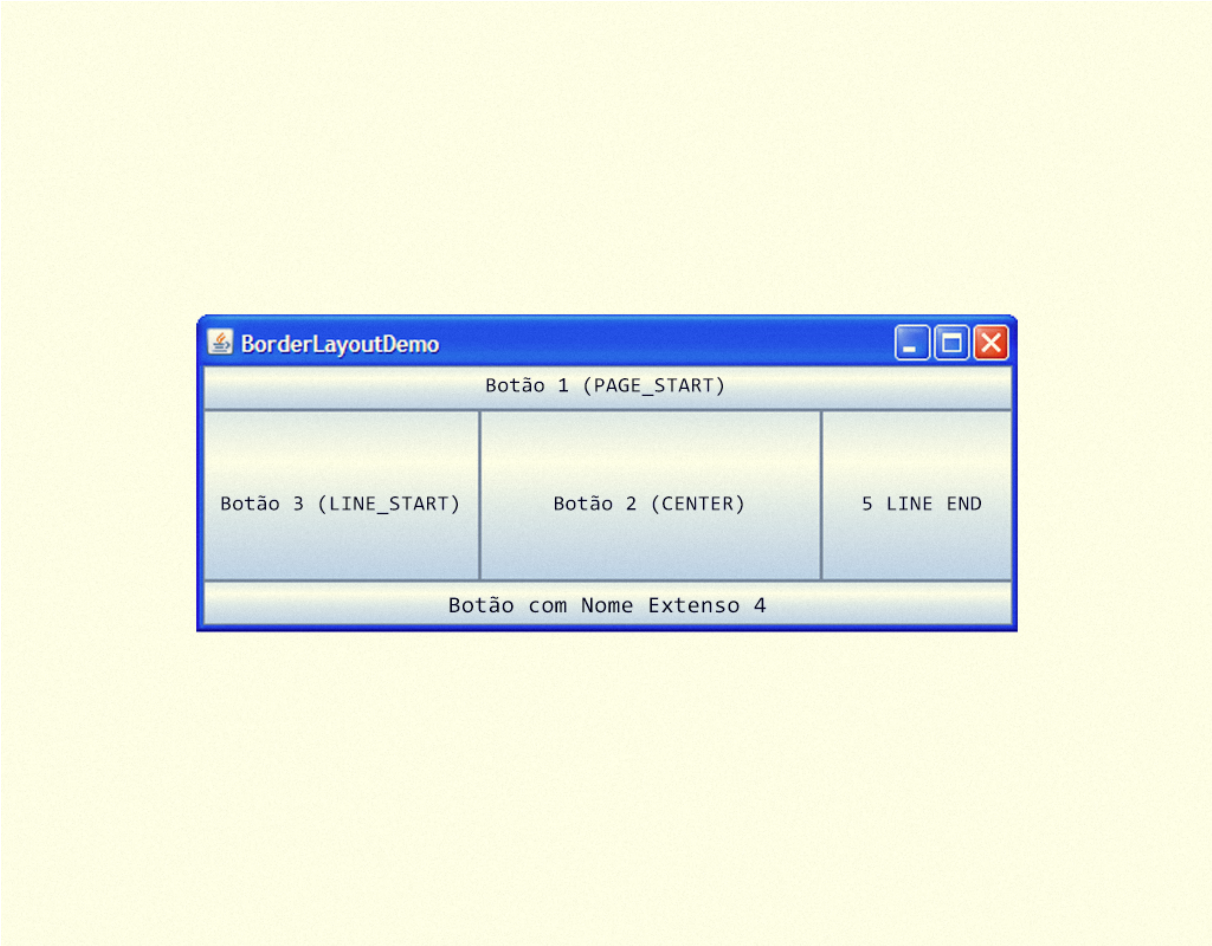
O BorderLayout divide o container em 5 regiões distintas para organizar os componentes, sendo elas: PAGE_START, PAGE_END, LINE_START, LINE_END e CENTER. A região PAGE_START corresponde à parte superior do container, PAGE_END, à parte inferior, enquanto LINE_START e LINE_END são direita e esquerda, respectivamente, e o CENTER é a parte central. Até o JDK 1.4 eram

utilizadas as direções cardinais para representar as regiões do BorderLayout (North, South, East e West), mas, após o lançamento do JDK 1.4, foram substituídas para a nomenclatura citada anteriormente.

O BorderLayout limita o container a ter, no máximo, 5 componentes, um para cada região, mas não impede que o componente de uma região seja um outro container que possa ter mais componentes anexados. Ele é o gerenciador de layout padrão do componente JFrame.

Os componentes que ocupam as regiões PAGE_START e PAGE_END têm mesmo tamanho horizontal que o container e a sua altura é a mesma do componente mais alto colocado nessas regiões. As regiões LINE_START e LINE_END preenchem o espaço vertical entre as regiões PAGE_START e PAGE_END, e a região CENTER ocupa o espaço restante (DEITEL, 2010).

Caso alguma das regiões não seja preenchida, os componentes se expandem para preencher o espaço deixado, exceto quando a região CENTER não tem componentes, ou seja, se as regiões PAGE_START ou PAGE_END não tiverem componentes, os componentes de LINE_START, CENTER e LINE_END são aumentados verticalmente para ocupar o espaço; caso LINE_START ou LINE_END não forem ocupados, o espaço é preenchido pelo componente de CENTER, mas, quando CENTER não é ocupado, o espaço fica vazio (DEITEL, 2010). Caso as cinco regiões estejam ocupadas, o container é tomado por cinco componentes de interface gráfica, como mostra a Figura 3.16.



3FIGURA 16.19 - BorderLayout FONTE: ORACLE.

O construtor do BorderLayout recebe dois parâmetros inteiros que representam o espaço em pixels entre os componentes do layout, que são os espaçamentos horizontais e verticais. O padrão é 1 pixel de espaçamento (DEITEL, 2010).

No BorderLayout, o método add, que adiciona os componentes ao container, aceita dois parâmetros, sendo o primeiro o componente a ser adicionado, e o segundo, em qual região ele será adicionado. Caso dois componentes sejam adicionados na mesma região, eles ficarão sobrepostos pois, a região deve contar apenas com um componente.

GridLayout

O GridLayout trata o container como uma grade, dividindo-o em linhas e colunas que conterão os componentes que serão adicionados. Cada componente terá as mesmas largura e altura (DEITEL, 2010).

Os componentes são adicionados em um GridLayout, começando a partir da primeira célula, que é localizada na parte superior esquerda da grade, prosseguindo para a direita. Quando a linha atual estiver completa com componentes, então, o gerenciador passa a adicionar componentes na próxima linha a partir da primeira coluna, localizada mais à esquerda (DEITEL, 2010).

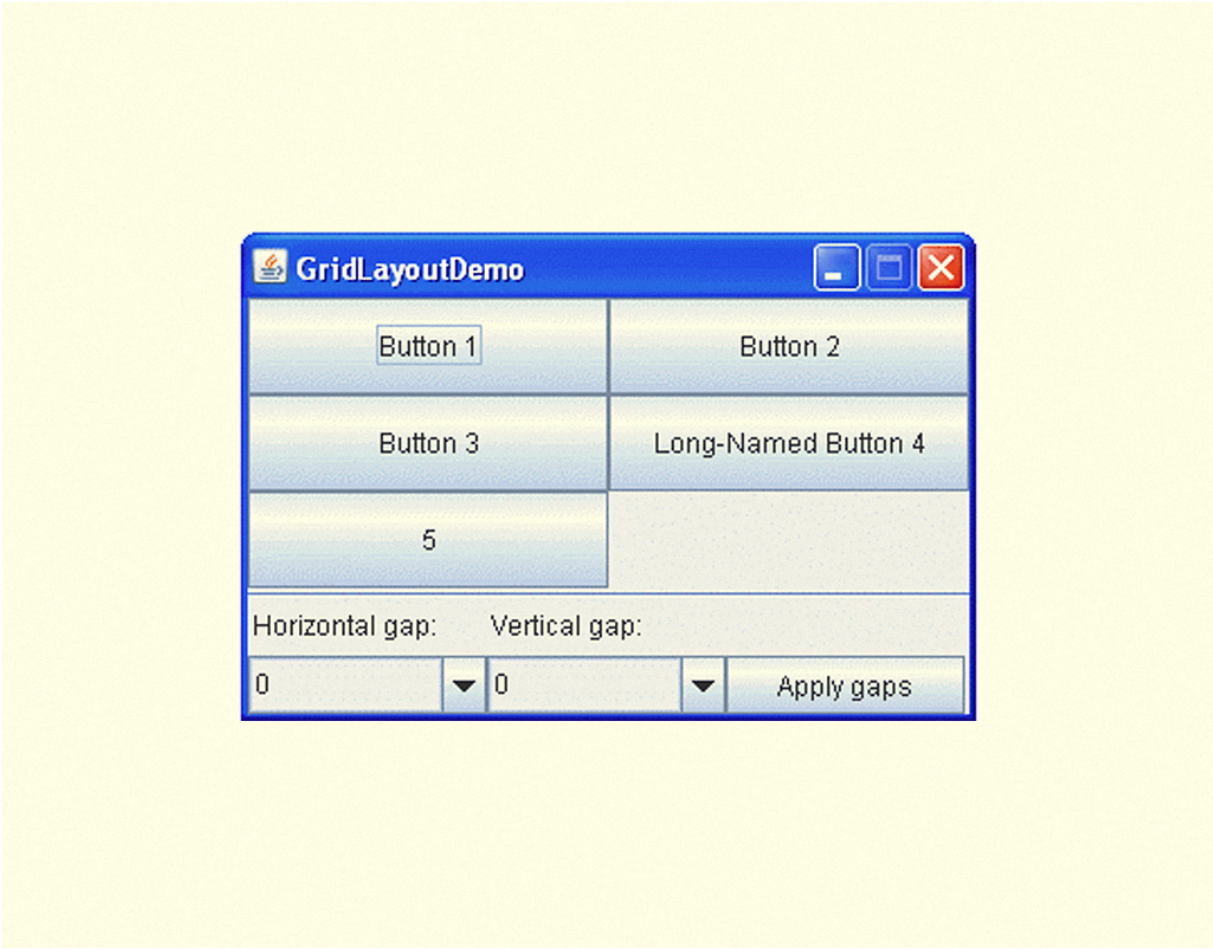
O construtor do GridLayout pode receber nenhum, dois ou quatro parâmetros. Caso nenhum parâmetro seja informado, é utilizado o *layout* padrão, com somente uma coluna, ou seja, um componente por linha. Quando são informados dois parâmetros, eles representam o tamanho da grade em que o GridLayout trabalhará, sendo o primeiro parâmetro o número de linhas e o segundo, o número de colunas. Um dos parâmetros pode ser 0, porém apenas um, e não os dois ao mesmo tempo, o que indica que qualquer número de objetos pode ser posicionado em uma linha ou coluna (no parâmetro que estiver com 0). Por exemplo, se um GridLayout for instanciado com o seguinte construtor:

```
GridLayout gridLayout = new GridLayout(0, 2);
```

significa que teremos uma grade com 2 colunas, porém com quantas linhas forem necessárias para colocar os componentes.

Quando o GridLayout é construído com quatro parâmetros, os dois primeiros representam o tamanho da grade (e seguem o mesmo princípio de um parâmetro com valor 0 acima), e o terceiro e quarto parâmetros representam o espaçamento horizontal e vertical, respectivamente, entre os componentes que serão adicionados.

A Figura 3.17 mostra uma janela com GridLayout, na qual são adicionados componentes em uma grade construída com valor 0 nas linhas e 2 colunas.



3FIGURA 17.19 - GridLayout FONTE: ORACLE.

Os botões de 1 a 5 são adicionados em painel que é gerenciado por meio do GridLayout. Os componentes abaixo (a linha separadora - JSeparator, os comboboxes - JComboBox e o botão "Apply gaps") e o painel do GridLayout são distribuídos em um outro painel gerenciado por um BorderLayout. O painel do GridLayout foi adicionado à região BorderLayout.PAGE_START, o JSeparator, na região BorderLayout.CENTER, e os ComboBoxes e o botão, na região BorderLayout.PAGE_END.

GridBagLayout

O gerenciador de layout GridBagLayout é bastante semelhante ao GridLayout, pois ambos utilizam uma grade para organizar os componentes, mas, ao contrário do GridLayout, o GridBagLayout permite que os componentes variem de tamanho, podendo ocupar múltiplas linhas e colunas, tornando-o um gerenciador de layout bastante poderoso e complexo (DEITEL, 2010).

O GridBagLayout utiliza o objeto **GridBagConstraints** para definir o comportamento de cada componente que será incluído na interface. Essas *constraints* informarão ao gerenciador em qual linha e coluna o componente estará localizado, o número de linhas e colunas que ele ocupará, dentre outras características. As principais estão exibidas na Figura 3.18.

Campo	Descrição
anchor	Especifica a posição relativa (NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST, CENTER) do componente em uma área que ele não preenche.
fill	Redimensiona o componente na direção especificada (NONE, HORIZONTAL, VERTICAL, BOTH) quando a área de exibição for maior que o componente.
gridx	A coluna em que o componente será colocado.
gridy	A linha em que o componente será colocado.
gridwidth	O número de colunas que componente ocupa.
gridheight	O número de linhas que o componente ocupa.
weightx	A quantidade de espaço extra a alocar horizontalmente. O componente na grade pode tornar-se mais largo se houver espaço extra disponível.
weighty	A quantidade de espaço extra a alocar verticalmente. O componente na grade pode tornar-se mais alto se houver espaço extra disponível.

3FIGURA 18.19 - Constraints do GridBagLayout FONTE: Deitel (2010, p. 791).

O campo **anchor** é utilizado quando o tamanho do componente é menor que a área que ele ocupará. O valor atribuído a essa propriedade indicará em qual local da célula o componente estará localizado. As constantes válidas são: CENTER (valor padrão), PAGE_START, PAGE_END, LINE_START, LINE_END, FIRST_LINE_START, FIRST_LINE_END, LAST_LINE_END, e LAST_LINE_START.

O campo **fill** determina como o componente será redimensionado. Os valores possíveis são: NONE (padrão) - não redimensiona o componente; HORIZONTAL - redimensiona horizontalmente, fazendo com que ele ocupe toda a área horizontal na célula; VERTICAL - redimensiona verticalmente, fazendo com que o componente ocupe toda a área vertical na célula; BOTH - redimensiona para ambas as direções.

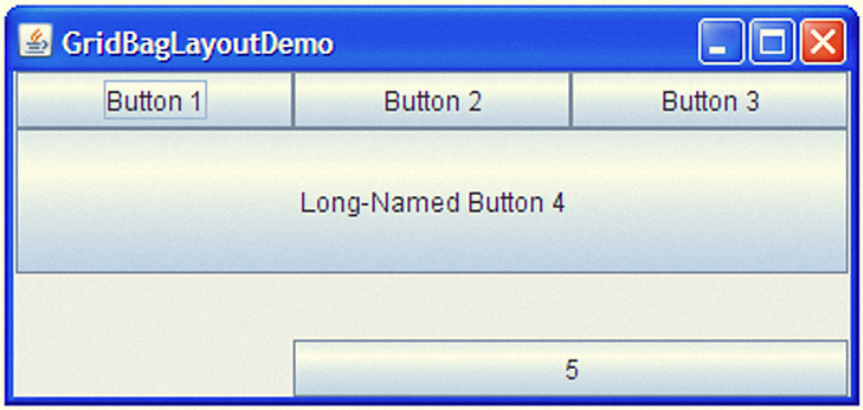
Os campos **gridx** e **gridy** especificam as posições dos eixos X (horizontal) e Y (vertical), respectivamente, para adicionar o componente. Utilizam como padrão a constante RELATIVE, e o próximo componente é adicionado após o último componente que foi inserido. Quando valores forem especificados, eles informarão em qual posição absoluta o componente será adicionado.

Os campos **gridheight** e **gridwidth** especificam o tamanho do componente com relação às células que ele ocupará. O gridheight especifica quantas linhas o componentes ocupará e o gridwidth quantas colunas. As constantes utilizadas são REMAINDER e RELATIVE, além dos valores que podem ser informados, sendo que o padrão é 1. A constante Remainder esticará o componente até a última célula, enquanto o Relative fará que o componente ocupe todas as células, exceto a última.

Os campos ***weightx*** e ***weighty*** especificam o percentual de crescimento que a célula terá quando a janela for redimensionada. O ***weightx*** especifica o quanto as células redimensionarão horizontalmente, enquanto ***weighty*** especifica o quanto as células redimensionarão verticalmente. O valor padrão para ambos é 0.

Os campos ***ipadx*** e ***ipady*** indicam o quanto as bordas do componente aumentarão. Ao serem alteradas, essas propriedades modificam o tamanho do componente, já que as bordas são somadas ao seu tamanho; o valor padrão é 0.

A Figura 3.19 mostra uma janela construída com o GridBagLayout. O site da Oracle, que exemplifica gerenciadores de layout, tem essa aplicação implementada para verificar como a janela se comporta ao redimensioná-la e como são especificadasas constantes. O acesso pode ser realizado pelo endereço: **https://docs.oracle.com <https://docs.oracle.com/javase/tutorial/uiswing/layout/gridbag.html>** .



3FIGURA 19.19 - GridBagLayout FONTE: ORACLE, *on-line*.

¶ Fique por dentro

O site de tutoriais da Oracle referente ao Java tem uma seção específica de Gerenciadores de Layout, explicando os funcionamentos e os exemplos de implementação de cada um dos gerenciadores de layout fornecidos pelo Java.

Disponível em: **https://docs.oracle.com <https://docs.oracle.com/javase/tutorial/uiswing/layout/index.html>** .

AWT x Swing

Ambos são classes que têm objetos para implementar interfaces gráficas em Java, porém AWT parou de ser utilizado, aumentando o uso de seu substituto Swing.

AWT

O AWT (Abstract Window Toolkit) foi o primeiro conjunto de ferramentas para criação de interfaces gráficas em Java. Ele foi criado com componentes simples que permitem criar interfaces gráficas básicas, com seu *look-and-feel* fixo em qualquer plataforma que seja executado. Em outras palavras, o mesmo código produzirá a mesma interface, caso ela seja executada no Windows, MacOS, Solaris, dentre outros (STELTING, 2002).

Os componentes de uma aplicação escrita em AWT são nativos à plataforma. A explicação disso é que a funcionalidade base do AWT está vinculada ao conceito de *peer* (tradução livre: ponta). Todo componente AWT tem uma classe *peer* que

está associada ao sistema operacional. As classes utilizadas pelo desenvolvedor para criar os componentes gráficos em AWT são classes *wrappers* (tradução livre: invólucro), ou seja, são classes que fazem a interação entre o código escrito pelo desenvolvedor e as classes *peers*, que se comunicam diretamente com o sistema operacional (STELTING, 2002).

Para criar um botão em AWT, é utilizada a classe java.awt.Button, que tem um implementador para interface java.awt.peer.ButtonPeer, que faz as tarefas de desenho do componente na tela (STELTING, 2002).

Com isso, dentro do AWT, existe a possibilidade de, de alguma maneira, ter informações acerca de quais são os *peers* específicos da plataforma. Essas informações são disponibilizadas na classe *Toolkit*, que contém o código para se relacionar com o sistema operacional subjacente. Essa classe raramente é utilizada por desenvolvedores, mas é essencial no funcionamento do AWT, pois é por meio dela que são carregados os objetos gráficos e criados os *peers* (STELTING, 2002).

Os componentes da GUI do AWT são considerados pesados (*heavyweights*) em relação aos componentes do Swing. Isso se dá pois eles estão vinculados diretamente ao sistema operacional por meio dos *peers*. Essa arquitetura do AWT teve alguns benefícios e consequências para o seu desenvolvimento (STELTING, 2002), como veremos a seguir.

Benefícios

- Menos código para ser escrito na API, pois a plataforma subjacente dos *peers* faz a maior parte do trabalho.
- As interfaces se comportam e aparentam como foram desenvolvidas no sistema operacional no qual são executados.

Desvantagens

- Se existe algum "problema" no componente gráfico do sistema operacional, a aplicação AWT herdará esses "problemas".
- A utilização dos *peers* para as operações de componentes da interface gráfica pode deixar a aplicação mais lenta e apresentar erros no redimensionamento.

Para evitar possíveis problemas da arquitetura baseada em *peers*, os desenvolvedores podem estender diretamente alguma das classes bases do modelo AWT: Component ou Container. Como essas classes não têm classes *peers*, qualquer subclasse delas herdará a funcionalidade do núcleo do AWT sem sofrer as limitações dos *peers*. Será necessário, porém, escrever os códigos referentes ao desenho dos componentes (STELTING, 2002).

Swing

A arquitetura do Swing (também chamado de Java Foundation Classes - JFC) é baseada no conceito de extensão de funcionalidades da classe Container do AWT. É a partir da sua subclasse JComponent que é gerada a maioria dos componentes gráficos do Swing (STELTING, 2002).

O Swing é construído em cima das classes do núcleo do AWT, herdando o comportamento básico desse AWT. Isso significa que as aplicações Swing utilizam a mesma abordagens que o AWT, para gerenciadores de *layout*, containers, alguns componentes e tratamento de eventos (STELTING, 2002). Para utilizar componentes do Swing, é necessário importar os pacotes javax.swing e java.awt (ficando clara a dependência de algumas classes bases do AWT).

Contudo o Swing é bem mais flexível, pois é escrito totalmente em Java, possibilitando criar uma grande quantidade de componentes gráficos com a vantagem de deixá-los mais customizáveis, uma vez que não necessitam do código

se comunicando diretamente com o sistema operacional, os *peers* (STELTING, 2002).

A classe Container é a superclasse dos componentes do Swing, o que significa que esses componentes podem conter outros componentes dentro, o que é o grande diferencial do Swing para o AWT, pois neste apenas alguns componentes tinham a capacidade de servir como container (STELTING, 2002).

O grande problema de construir uma arquitetura inteira baseada em uma outra arquitetura já existente é que não é possível garantir que a nova arquitetura não tenha desvantagens. A hierarquia de heranças das classes do Swing pode chegar a um grande nível de complexidade e, muitas vezes, tornar difícil a visualização do comportamento de algum objeto, por exemplo, o JButton segue esta hierarquia de classes: Object > Component > Container > JComponent > AbstractButton > Button (STELTING, 2002).

Como o Swing não utiliza as classes *peers* para fazer a comunicação direta com o sistema operacional, e é codificado diretamente em Java, ele é considerado leve em relação ao AWT, ou seja, uma interface gráfica construída em Swing demorará menos para carregar e consumirá menos recursos.

Contudo, como o Swing é baseado no AWT, existem alguns componentes pesados no Swing, pois são eles que fazem a comunicação com o sistema operacional, como apresentado no Quadro 3.1 (STELTING, 2002).

CLASSE AWT	EQUIVALENTE NO SWING
Applet	JApplet
Dialog	JDialog
Frame	JFrame
Window	JWindow

3FIGURA 1.2 - Classes AWT e Swing FONTE: Stelting (2002, p. 189).

Essas classes exibidas no Quadro 3.1 são as classes que se comunicam diretamente com o sistema operacional, ou seja, elas terão o *look-and-feel* do sistema operacional em execução, em compensação, os outros componentes poderão ter sua aparência alterada pela configuração de *look-and-feel* da GUI.

Principais diferenças

O Quadro 3.2 lista as principais diferenças e características entre o AWT e o Swing.

AWT	SWING
Componentes AWT são chamados de "componentes pesados"	Componentes Swing são chamados de "componentes leves"
Necessita do pacote java.awt	Necessita dos pacotes java.awt e javax.swing
Componentes são dependentes da plataforma (sistema operacional)	Componentes escritos diretamente em Java e independentes de plataforma
Look-and-feel fixo na plataforma em que a aplicação está rodando	Pode ter diversos look-and-feel configuráveis por aplicação
Existem classes peers que fazem a interação com o sistema operacional, tornando todos os componentes pesados e lentos.	Os peers existem somente nas classes bases, todos os outros componentes são independentes. Contudo há mais código escrito para tratar comportamento e desenho dos componentes, mas é menos oneroso para a aplicação.
AWT é considerado apenas uma camada de código visual em cima do sistema operacional.	Swing é muito mais complexo, com mais funcionalidades e flexibilidade.

3FIGURA 2.2 - Comparativo AWT x Swing FONTE: adaptado de Stelting (2002).

¶ Fique por dentro

COMPONENTIZAÇÃO

A componentização tem como objetivo transformar pedaços do *software* em componentes que podem ser reutilizados em diversas partes do desenvolvimento. A vantagem de utilizar a componentização é aumentar a produtividade, que é um assunto discutido amplamente no processo de desenvolvimento de software. Com o desenvolvimento de componentes, a ideia de "crie uma vez, utilize onde quiser" surge para ser aplicada onde for cabível, seja no mesmo sistema ou em sistemas diferentes, mas que possuam algumas funcionalidades similares.

Um exemplo simples de componentização pode ser aplicado em interfaces de login, que é uma funcionalidade utilizada por vários sistemas. Uma interface de login projetada corretamente para ser desenvolvida e funcionar como um componente pode ser reutilizada em diversas aplicações que precisam de uma autenticação para acessá-la.

Para a componentização ser realizada corretamente, ela deve passar por processos de estruturação, com base na decomposição do sistema em componentes funcionais, para isso, ela depende de vários princípios da Engenharia de Software e é tratada mais especificamente na Engenharia de Componentes.

O link a seguir realiza uma abordagem bastante completa da componentização, e o livro "Engenharia de Software - Uma Abordagem Profissional", de Roger S. Pressman, também aborda o assunto.

Disponível em: **http://www.linhadecodigo.com.br <http://www.linhadecodigo.com.br/artigo/3119/engenharia-de-componentes-parte-1.aspx>** .

Diversos artigos podem ser encontrados nos meios acadêmicos por meio das palavras-chave Desenvolvimento Baseado em Componentes - DBC (Component Based Development - CBD) ou Engenharia de Software Baseada em Componentes - ESBC (Component-based Software Engineering - CBSE).

Indicação de leitura

Nome do livro:: Java Como Programar

Editora:: Pearson Prentice Hall

Autor:: Harvey Deitel e Paul Deitel

ISBN:: 9788576055631

O livro é muito completo, abordando praticamente tudo que é disponível em Java, desde a sintaxe básica até implementações de interface gráficas, banco de dados e aplicações para Web. A leitura desse material é fundamental.

UNIDADE IV

Introdução ao SGBD utilizando Java

Ricardo de Almeida Rocha

Muitos sistemas precisam guardar as informações que são passadas a ele, para que sejam recuperadas em um momento futuro para trabalhar com elas, seja por meio de geração de relatórios, por consultas futuras ou, inclusive, por alteração nessas informações. Dessa forma, são utilizados bancos de dados para guardar as informações até que os sistemas as utilizem novamente.

Os banco de dados utilizam uma linguagem própria para a manipulação de seus dados, o SQL (Structured Query Language), e é por meio dessa linguagem que os dados são consultados, alterados e excluídos.

Esta unidade introduzirá os comandos básicos do SQL, a instalação e a configuração de um SGBD, que realizará o gerenciamento e a administração de um banco de dados. Com esse banco pronto para uso, utilizaremos a API do JDBC para implementar o acesso ao banco de dados por meio de códigos implementados em Java, utilizando os padrões MVC e DAO para estruturar o código de forma correta e flexível.

SGBD

Deitel (2010, p. 899) define banco de dados como "uma coleção organizada de dados", esses dados serão utilizados pelos sistemas para exibir informações ao usuário. Essas informações, os dados, podem ser manipuladas pelo usuário, para, então, voltarem a ser gravadas no banco de dados para uma próxima utilização.

Essas operações realizadas nos dados (inserir, modificar, incluir, excluir) só podem ser realizadas mediante um SGBD - Sistema de Gerenciamento de Banco de Dados. Um SGBD fornece mecanismos para armazenar, organizar, recuperar e modificar os dados (DEITEL, 2010).

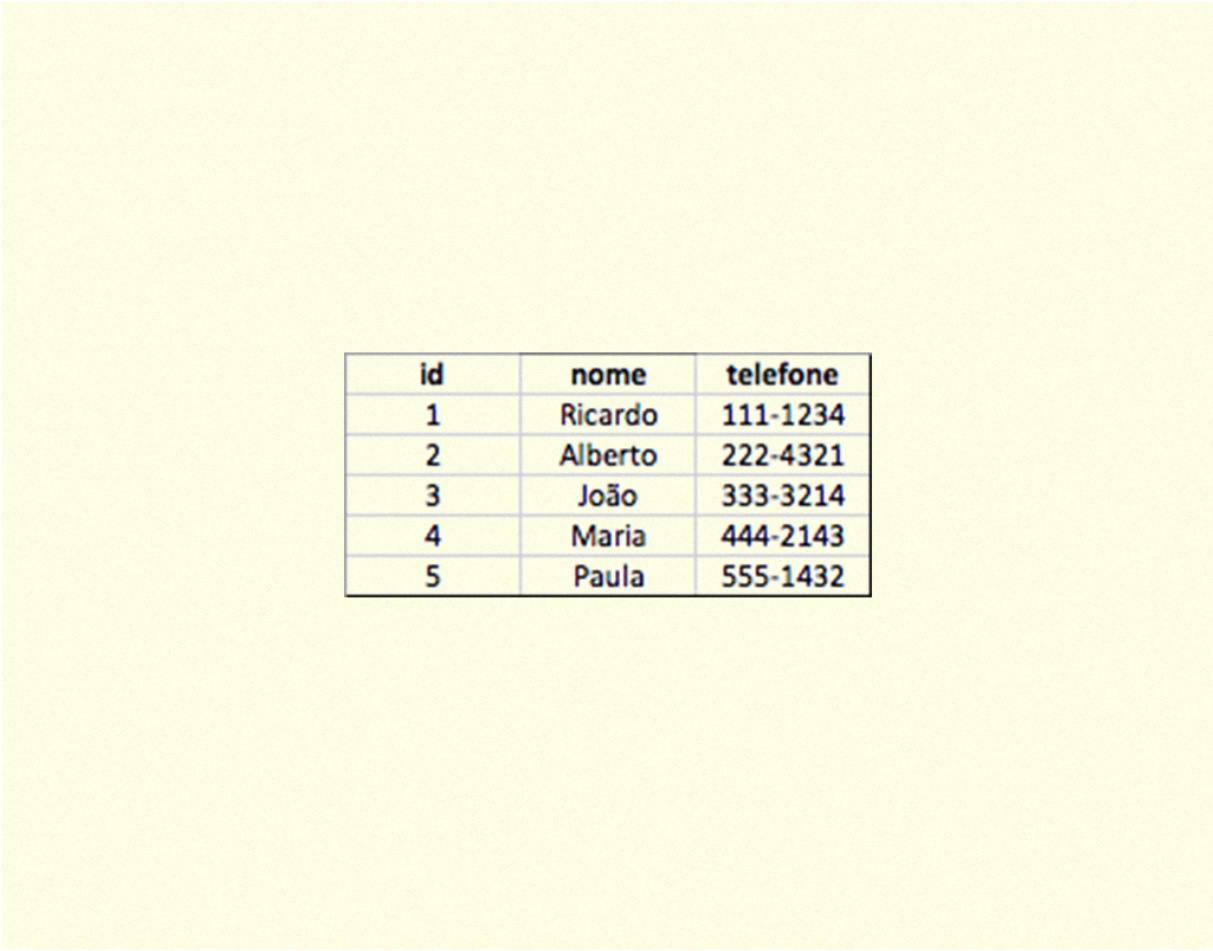
Os bancos de dados utilizados hoje são os relacionais, que armazenam os dados sem levar em conta sua estrutura física (DEITEL, 2010). Os bancos de dados mais utilizados no mundo são do tipo relacional, como Microsoft SQL Server, Oracle, DB2, PostgreSQL, MySQL e JavaDB.

A comunicação entre um sistema Java e o banco de dados é feita por meio de um JDBC (*Java Database Connectivity*). O driver JDBC permite aos aplicativos conectarem-se a um banco de dados relacional e manipularem os dados por meio da API do JDBC (DEITEL, 2010).

Banco de dados relacionais

Como citado anteriormente, um banco de dados relacional é aquele que não considera a estrutura física do dado para acessá-lo. O banco de dados relacional armazena os dados em tabelas, cada uma com um nome único. Essas tabelas são

divididas em linhas e colunas. As linhas representam os registros de cada tabela, ou seja, se uma tabela conter 10 linhas, ela terá 10 registros referentes ao dado que a tabela guarda, por exemplo, se a tabela deve guardar dados sobre Clientes, cada linha da tabela será um cliente. As colunas são os valores de um determinado registro, por exemplo, um cliente terá nome, telefone e identificador, cada um desses atributos será uma coluna de valor de um cliente, como mostra a Figura 4.1.



id	nome	telefone
1	Ricardo	111-1234
2	Alberto	222-4321
3	João	333-3214
4	Maria	444-2143
5	Paula	555-1432

4FIGURA 1.26 - Dados de uma tabela cliente FONTE: adaptada de Deitel (2010, p. 900).

Na imagem, temos 3 colunas, a primeira é o **id** que se refere ao identificador, que é uma chave primária para essa tabela. A chave primária é uma coluna (ou um grupo de colunas) com um valor que não pode ser duplicado em outras linhas, ou

seja, um valor único (DEITEL, 2010). Caso a chave primária seja um grupo de colunas, a combinação desses valores não pode se repetir, por exemplo, se as chaves primárias fossem id e nome, poderíamos ter duas linhas com os dados: 1, Ricardo e 2, Ricardo. Observamos que o nome se repetiu, porém o id não. Quando uma chave primária é um grupo de colunas, ela é chamada de chave primária composta.

As colunas seguintes são **nome** e **telefone** de cada cliente. Uma tabela pode ter diversas colunas para modelar os dados necessários para o sistema.

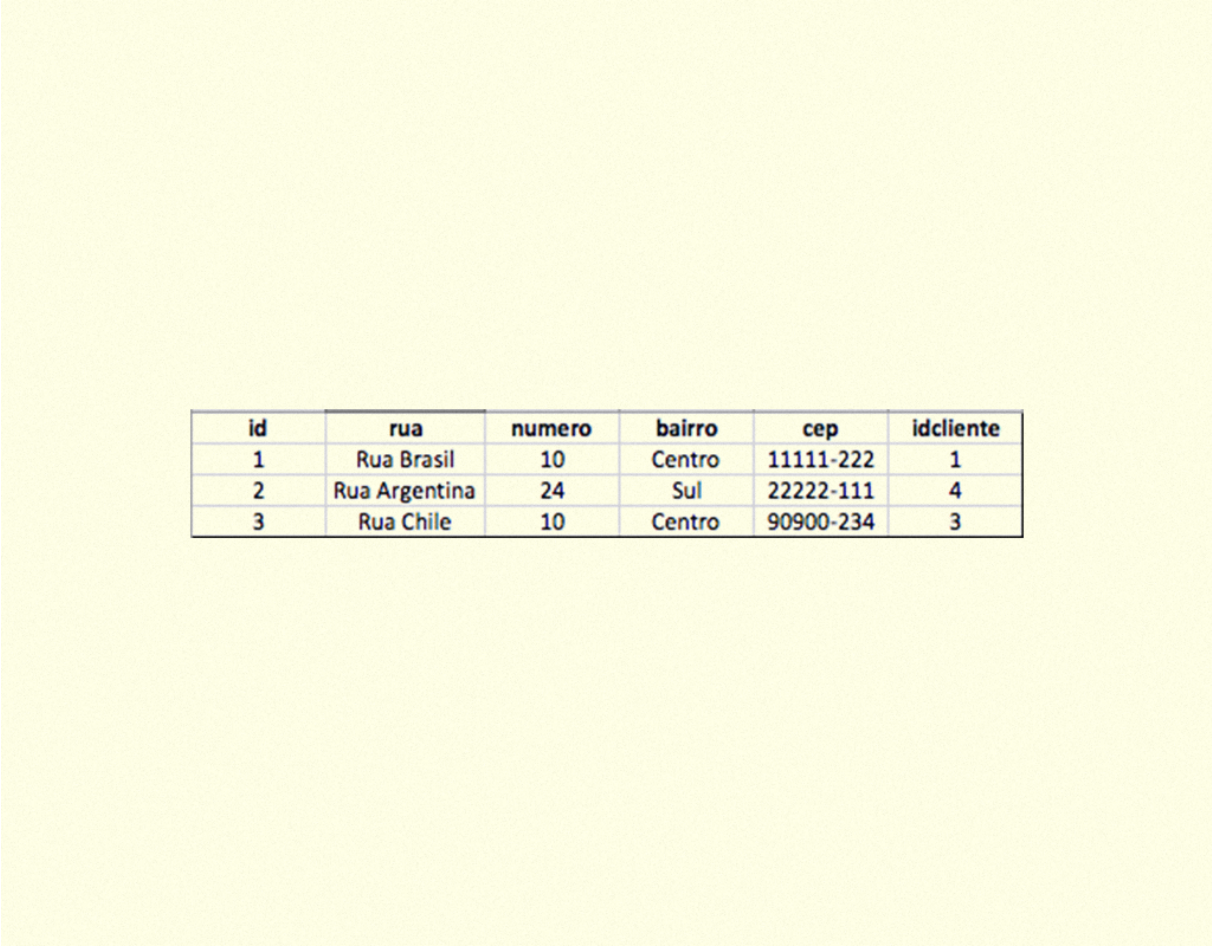
As linhas na tabela não são armazenadas em uma ordem específica, porém, quando o desenvolvedor for resgatar os dados de uma tabela, pode ser especificado um critério de ordenação para exibir os dados. Na Figura 4.1, coincidentemente, as linhas estão ordenadas pelo id, porém, se for realizada uma consulta na tabela e solicitar uma ordenação pelo nome, o primeiro registro seria o da segunda linha, com nome Alberto e id 2.

Os dados são retornados do banco de dados por meio de comandos SQL, que serão vistos na seção 1.2.

Em um banco de dados de um sistema, existem diversas tabelas para guardar as informações utilizadas por esse sistema. Para o sistema funcionar corretamente, exibindo diversos tipos de dados e realizando as tarefas requisitadas, essas tabelas deverão relacionar-se. Utilizando a tabela de Cliente, poderíamos estender o exemplo criando uma tabela de Endereço, que seria referente ao endereço completo de cada cliente.

Como a tabela Endereço guardará os dados de um endereço, por exemplo, rua, numero, bairro, cep, esses dados deverão ser identificados, mostrando para qual cliente eles pertencem. Para isso, é utilizada uma "chave estrangeira", que é a replicação da chave primária de uma tabela para a outra tabela, relacionando as duas tabelas por meio dessa chave.

A Figura 4.2 exibe a tabela Endereço com as colunas id (chave primária da tabela Endereço), rua, numero, bairro, cep e idCliente (que é a chave estrangeira da tabela Cliente).



id	rua	numero	bairro	cep	idcliente
1	Rua Brasil	10	Centro	11111-222	1
2	Rua Argentina	24	Sul	22222-111	4
3	Rua Chile	10	Centro	90900-234	3

4FIGURA 2.26 - Tabela Endereço FONTE: Microsoft Windows.

A coluna **idcliente** é a coluna que realizará o relacionamento entre a tabela Endereço e a tabela Cliente, ou seja, caso não existisse a coluna idcliente, não haveria uma forma de vincular uma linha da tabela Endereço para um Cliente.

O relacionamento entre tabelas pode ser expressado em forma de diagramas, como o Diagrama Entidade-Relacionamento (DER), que é mais focado nas matérias de Banco de Dados e de Engenharia de Software.

¶ Fique por dentro

Mais informações acerca de diagramas de relacionamento, como o DER e outros, podem ser encontradas no livro do Pressman a respeito da Engenharia de Software:

PRESSMAN, Roger S. Engenharia de Software - Uma Abordagem Profissional. 8ª ed. Editora: AMGH: Porto Alegre, 2016.

Se as especificações da regra de negócio do sistema trabalharem com um endereço por cliente e esse endereço não pode estar vinculado a outro cliente, esse relacionamento seria um relacionamento **um-para-um**, no qual um cliente pode ter um endereço, e um endereço está vinculado a apenas um cliente.

Existem relacionamentos **um-para-muitos** e **muitos-para-muitos.** Um relacionamento um-para-muitos acontece quando um cliente poderia ter diversos endereços, então, na tabela Endereço, poderia existir mais de uma linha que tenha a coluna idcliente com um valor repetido, ou seja, duas linhas com a coluna idcliente assumindo o valor 1, por exemplo. Já o relacionamento muitos-para-muitos ocorre quando as tabelas relacionadas podem ter vários registros associados. Nesse caso, é criada uma terceira tabela, em que irá conter as chaves primárias das duas tabelas e os valores que serão relacionados. No exemplo de Cliente e Endereço, seria criada a tabela ClienteXEndereco, que conteria a chave primária do Cliente e a chave primária do Endereço, assim, seria possível verificar a quais endereços o cliente está relacionado, ou a quais clientes determinado endereço está relacionado.

idcliente	idendereco
1	1
1	4
2	1
3	3

4FIGURA 3.26 - Tabela ClienteXEndereco FONTE: Microsoft Windows.

A Figura 4.3 mostra como ficaria a tabela ClienteXEndereco de um relacionamento muitos-para-muitos. Analisando essa tabela, é possível verificar que o cliente com id 1 tem dois endereços, o 1 e o 4, enquanto o cliente 2 tem somente um endereço, o 1, ou seja, o cliente 1 apresenta múltiplos endereços, enquanto o endereço 1 possui múltiplos clientes.

Cada relacionamento vai depender das tabelas que serão relacionadas e de como foi realizada a especificação para o desenvolvimento de software. Pode ser que os casos que o sistema trate não sejam os que tenham a melhor representação da vida real, porém é o necessário e suficiente para o sistema funcionar corretamente. Essa parte é definida na análise das necessidades para o desenvolvimento do software,

que abrange Engenharia de Software e, também, conhecimentos em Banco de Dados, para definir as tabelas que serão utilizadas da melhor forma possível, diminuindo as repetições de dados e os possíveis casos de dados inconsistentes.

SQL

SQL (*Structured Query Language* - Linguagem de Consulta Estruturada) é a linguagem padrão para realizar comandos em um banco de dados. A linguagem SQL tem diversas palavras-chave para executar comandos no banco de dados, como selecionar dados de uma ou mais tabelas, inserir dados em uma tabela, atualizar e excluir dados de uma tabela, dentre outros.

A Figura 4.4 mostra as principais palavras-chave do SQL.

SQL, palavras-chave	Descrição
SELECT	Recupera dados de uma ou mais tabelas.
FROM	Tabelas envolvidas na consulta. Requeridas em cada SELECT.
WHERE	Critérios de seleção que determinam as linhas a ser recuperadas, excluídas ou atualizadas. Opcional em uma consulta ou uma instrução de SQL.
GROUP BY	Critérios para agrupar linhas. Opcional em uma consulta SELECT.
ORDER BY	Critérios para ordenar linhas. Opcional em uma consulta SELECT.
INNER JOIN	Mescla linhas de múltiplas tabelas.
INSERT	Insere linhas em uma tabela especificada.
UPDATE	Atualiza linhas em uma tabela especificada.
DELETE	Exclui linhas de uma tabela especificada.

4FIGURA 4.26 - Palavras-chave do SQL FONTE: Deitel (2010, p. 903).

Cláusulas SELECT/FROM/WHERE

SELECT é o comando SQL mais utilizado em um banco de dados. Ele é responsável por realizar consultas em um banco de dados recuperando valores de uma ou mais tabelas.

A sintaxe principal do Select é:

*SELECT * FROM nomeDaTabela*

Sendo que o **asterisco** (*) significa que todas as colunas da tabela devem ser recuperadas. Caso seja necessário apenas algumas colunas, elas são especificadas separadas por **vírgula** (,), por exemplo:

SELECT ID, NOME FROM CLIENTE

Esse comando está recuperando os registros da tabela cliente, retornando apenas as colunas ID e NOME.

O **asterisco** (*) é utilizado em poucos casos, pois é recomendado retornar apenas os campos que serão utilizados, porque, caso a consulta seja realizada em uma tabela grande, que tenha diversas colunas, isso aumentará o tempo de busca, diminuindo o desempenho do sistema (DEITEL, 2010).

Nos comandos anteriores, as buscas que eram realizadas nas tabelas retornavam todas as linhas dessa tabela, o que pode não ser desejado em algum caso. Para isso, é utilizada a cláusula WHERE, que especifica um filtro para a consulta, ou seja, são retornadas as linhas que satisfazem um critério de seleção informado (DEITEL, 2010).

A cláusula WHERE é opcional nas consultas. A sua forma básica é exibida a seguir:

SELECT colunal, coluna2, ..., colunaN FROM nomeDaTabela WHERE critérioDeSeleção

Voltando ao exemplo da tabela de clientes, caso desejamos selecionar apenas o cliente com nome de João, escreveríamos a consulta:

SELECT id, nome, telefone FROM cliente WHERE UPPER(nome) = 'JOÃO'

Nessa consulta, temos alguns detalhes. Foram selecionadas as colunas id, nome e telefone, que são todas as colunas que a tabela Cliente contém. Nesse caso, poderíamos utilizar o asterisco (*), já que são poucas colunas, porém decidimos colocar os campos, pois, quando o resultado dessa coluna for utilizado no sistema, em algum trecho do código pode ser requisitada a primeira coluna do resultado. Especificando, da forma anterior, temos certeza de que a primeira coluna será o **id**; caso utilizássemos o asterisco, não teríamos uma garantia de que a primeira coluna seria o id, podendo gerar inconsistência na execução.

Outro detalhe é a utilização da função UPPER(campo) no critério de seleção WHERE. Essa função é utilizada para converter a *string* do campo para letras maiúsculas para, então, realizar a comparação com o critério escolhido, no caso, a *string* JOÃO. Alguns bancos de dados identificam que João é diferente de JOÃO, que é diferente de joão, por isso, é aconselhável utilizar uma função como o UPPER (para comparar utilizando letras em caixa alta) ou o LOWER (que transforma o valor do campo em letras minúsculas para a comparação).

As *strings* são comparadas entre aspas simples (' ') e podem ser utilizados os seguintes operadores de comparação: < (menor que), > (maior que), <= (menor ou igual que), >= (maior ou igual que), = (igual), <> (diferente) e LIKE. O operador LIKE é utilizado para correspondência por padrão, em que são utilizados os

caracteres % e _ como curingas (DEITEL, 2010). O curinga % procura *substrings* que contenham zero ou mais caracteres; para uma busca em que um nome comece por JO e termine com qualquer outra string, é utilizado: WHERE UPPER(nome) LIKE 'JO%'. O curinga _ indica que pode ter apenas um caracter qualquer quando utilizado.

Order By

A cláusula **Order By** tem como objetivo ordernar as linhas do resultado de uma consulta. A ordenação pode ser crescente ou decrescente, utilizando a palavra-chave ASC ou DESC, respectivamente. A cláusula Order By é opcional (DEITEL, 2010).

A consulta a seguir mostra a utilização de um ordenamento pelo id:

*SELECT * FROM Cliente ORDER BY id*

Caso não seja especificado se a ordenação será crescente ou decrescente, o padrão do SGBD é ordenar de forma crescente. Quando a coluna que será ordenada for do tipo *string*, será ordenada em ordem alfabética, caso utilizado o ASC ou não for especificado o tipo da ordenação.

A ordenação pode ser realizada utilizando duas colunas, sendo ordenada a partir da primeira coluna; quando ocorrer ambiguidade na ordenação pelo primeiro critério, essa ambiguidade será ordenada pelo segundo critério e assim por diante.

Inner Join

Quando precisamos exibir dados que estão em duas ou mais tabelas, necessitamos juntá-las em uma consulta para termos acesso aos resultados. Essa junção é feita pela cláusula **Inner Join** (ou somente Join).

O Inner Join trabalha juntando duas (ou mais) tabelas a partir da comparação de colunas em cada tabela, para determinar quais linhas serão mescladas. Normalmente, a junção é realizada pela chave primária de uma tabela com a chave estrangeira de outra tabela. Pode-se ter casos em que não é utilizada a chave primária, mas deve-se tomar cuidado para não mesclar dados inconsistentes.

O comando SQL a seguir exemplifica a utilização do Join com base nas tabelas trabalhadas.

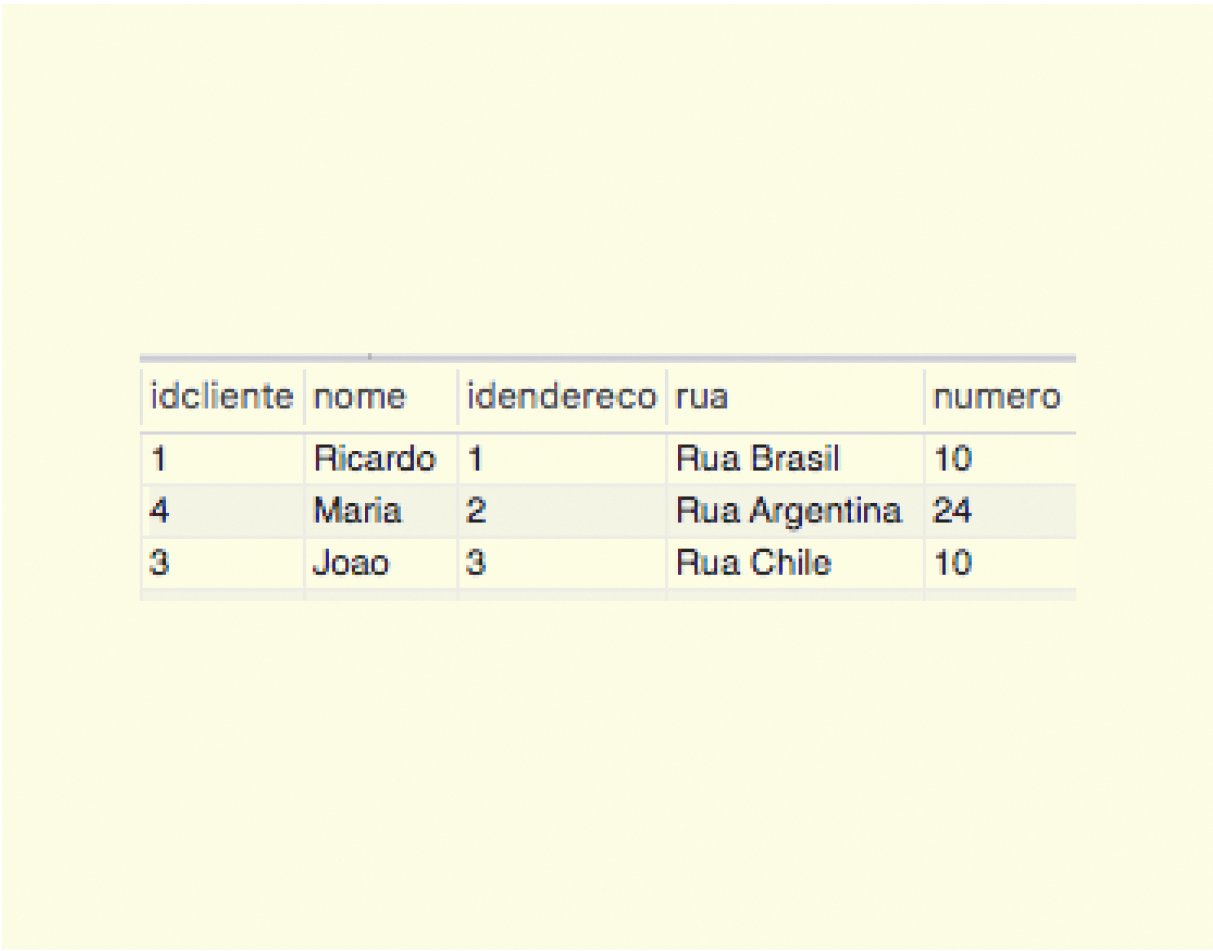
```
SELECT A.id idcliente, A.nome, B.id idendereco, B.rua, B.numero
FROM cliente A
JOIN endereco B ON (A.id = B.idcliente)
```

Nessa consulta, é exibido o id do cliente, seu nome e endereço. A junção é feita pela chave primária da tabela Cliente (id) e pela chave estrangeira na tabela Endereço (idcliente), utilizando a palavra-chave ON da cláusula JOIN. É possível utilizar cláusulas de Where, Order By e Group By em uma consulta com Join, especificando-os abaixo do Join.

É possível verificar que foram utilizados *aliases* para as tabelas e para os campos id que são retornados. *Alias* é um apelido que é informado para os campos e para as tabelas em um comando SQL. O *alias* de campo serve para organizar os resultados, como exibido na consulta anterior, temos dois campos **id** que são retornados da tabela cliente e da tabela endereço; com isso, utilizamos um *alias* para especificar que um id é da tabela cliente (idcliente) e o outro id é da tabela endereço (idendereco). O *alias* de campo é especificado após o nome do campo na cláusula SELECT e, quando especificado, na tabela de resultado, será exibido o *alias*, e não mais o nome original da coluna. O *alias* de tabela é utilizado quando serão feitas junções para especificar quais campos das tabelas serão utilizados para a junção. O *alias* de tabela é especificado após o nome da tabela na cláusula FROM. Na consulta exibida anteriormente, o *alias* A foi utilizado na tabela cliente

e o *alias* B na tabela endereço; ambos foram utilizados para especificar os campos das tabelas na cláusula do JOIN e as colunas que foram selecionadas na cláusula do SELECT.

A consulta retornará os resultados da Figura 4.5.



idcliente	nome	idendereco	rua	numero
1	Ricardo	1	Rua Brasil	10
4	Maria	2	Rua Argentina	24
3	Joao	3	Rua Chile	10

4FIGURA 5.26 - Resultado Inner Join FONTE: MySQL Workbench.

Os dados da tabela Cliente que não tinham um endereço vinculado na tabela Endereço não foram exibidos. Podemos ver que o campo idcliente é referente ao campo **id** da tabela Cliente, e o campo idendereco é referente ao campo **id** da

tabela Endereco. No comando SQL executado, foram utilizados *labels* para os campo, já que ambos têm o mesmo nome, para ficar mais clara a identificação de cada campo.

Nos dados retornados, podemos observar que o idcliente é diferente do idendereco, pois o campo idendereco é a chave primária que identificará cada registro da tabela Endereço.

A cláusula INNER JOIN pode ser escrita somente como JOIN, que resultará nos mesmos dados selecionados.

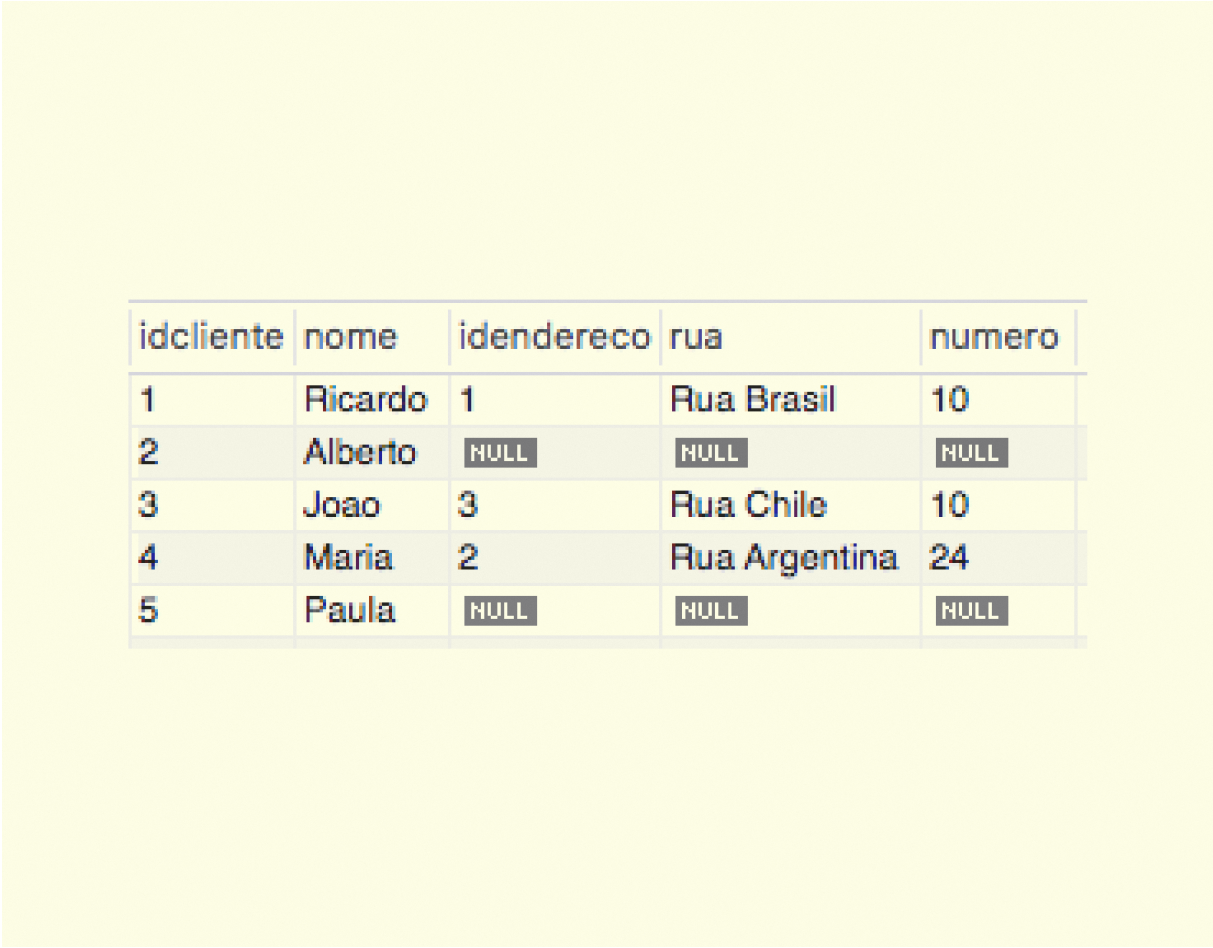
Left Join e Right Join

Além do Inner Join, existem outros tipos de junções, como Left Join, Right Join, Full Outer Join e Cross Join, mas veremos o Left Join e Right Join, que são mais utilizados dentre os citados.

Ao contrário do Inner Join, que resulta apenas nos dados correspondentes de uma tabela em outra, o Left Join retorna a tabela A inteira e os dados correspondentes (com a cláusula especificada no ON) da tabela B (ou campos nulos para os que não possuírem correspondência).

A consulta a seguir retorna os dados da Figura 4.6.

```
SELECT A.id idcliente, A.nome, B.id idendereco, B.rua, B.numero
FROM cliente A
LEFT JOIN endereco B ON (A.id = B.idcliente)
ORDER BY idcliente
```



idcliente	nome	idendereco	rua	numero
1	Ricardo	1	Rua Brasil	10
2	Alberto	NULL	NULL	NULL
3	Joao	3	Rua Chile	10
4	Maria	2	Rua Argentina	24
5	Paula	NULL	NULL	NULL

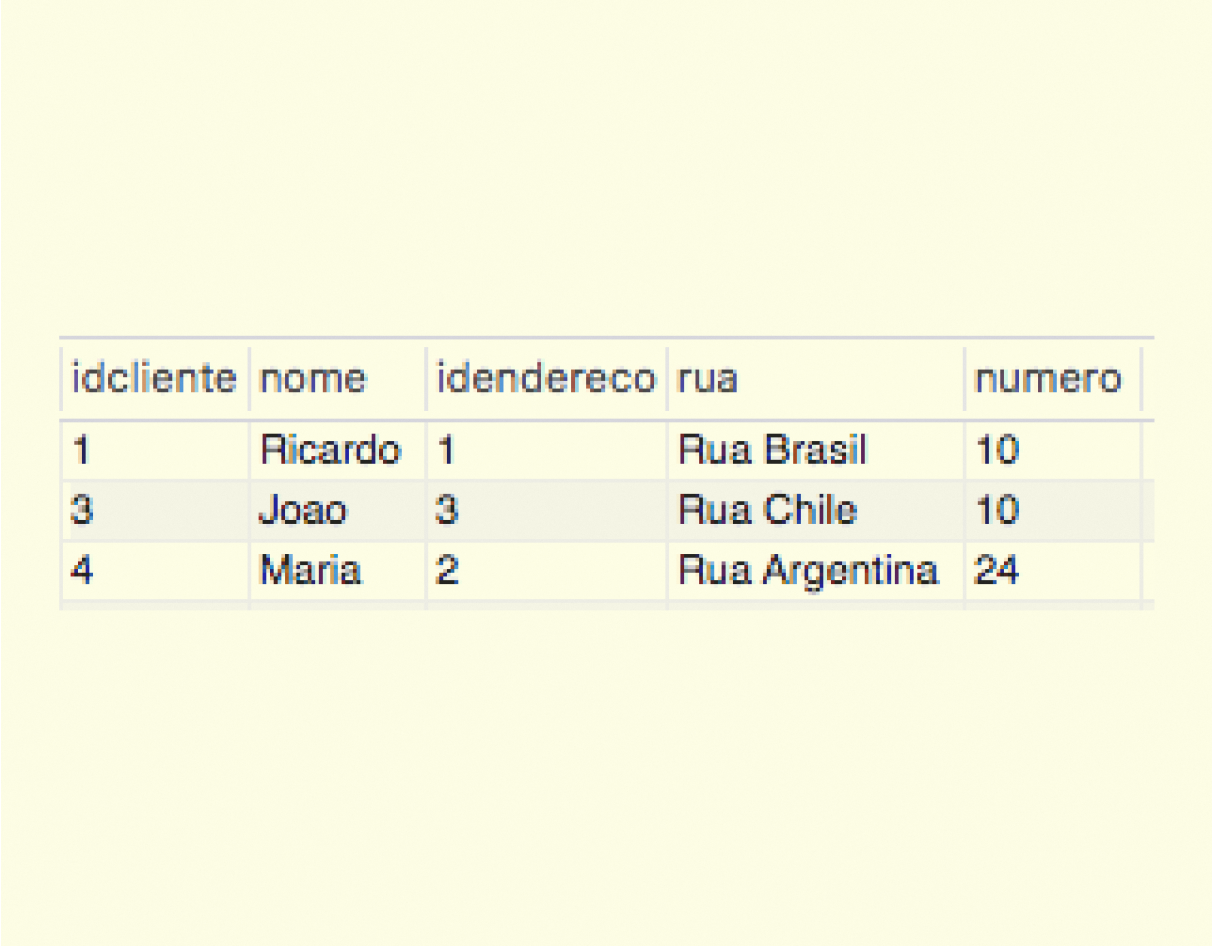
4FIGURA 6.26 - Resultado Left Join FONTE: MySQL Workbench.

Podemos ver que todos os dados da tabela Cliente foram exibidos, e os que não tinham uma linha referenciada pela cláusula do ON na tabela Endereco foram listados com dados nulos nos campos da tabela Endereco.

Caso alterássemos a cláusula do ON para (A.id = B.id) os dados retornados seriam outros, porém se deve verificar como o sistema está estruturado, para não resultar em dados inconsistentes.

O Right Join é o inverso do Left Join. Ele exibe todos os dados da tabela B e os dados correspondentes ao ON da tabela A (ou nulos para os que não possuírem correspondentes).

A consulta anterior com Right Join resulta nos dados da Figura 4.7.



idcliente	nome	idendereco	rua	numero
1	Ricardo	1	Rua Brasil	10
3	Joao	3	Rua Chile	10
4	Maria	2	Rua Argentina	24

4FIGURA 7.26 - Resultado Right Join FONTE: MySQL Workbench

¶ Reflita

O que aconteceria se invertêssemos a ordem das tabelas especificadas no Right Join? E por que o resultado da Figura 4.6 não tem nenhum valor nulo?

Insert

A instrução INSERT é a responsável por inserir dados nas tabelas do banco de dados. A sua inserção ocorre por meio de um comando que especifica os valores dos campos de determinada tabela.

A forma básica do Insert é (DEITEL, 2010):

INSERT INTO nomeDaTabela (nomeColunal, nomeColuna2, ..., nomeColunaN) VALUES (valorl, valor2, ..., valorN)

Em que nomeDaTabela especifica qual tabela será populada com os valores informados. Os campos nomeColunal, nomeColuna2, nomeColunaN representam o nome da coluna que foi especificado na criação da tabela. Caso todas as colunas da tabela receberem um valor, os nomes das colunas são opcionais, ou seja, se uma instrução Insert for inserir valores em todas as colunas da tabela, é possível escrever somente INSERT INTO nomeDaTabela VALUES (...), em que os valores serão separados entre vírgula (,) e na ordem das colunas que a tabela contém.

Caso não sejam todas as colunas que receberão valores, o nome das colunas são especificados e, após a palavra-chave VALUES, são informados os valores das colunas acima, na ordem que foram especificadas.

Para inserirmos um novo registro na tabela Cliente, seria executada a seguinte instrução:

INSERT INTO cliente (nome, telefone) VALUES ('Alfredo', '555-5566')

Como estamos informando valores para todas as colunas, poderíamos escrever a instrução da seguinte forma:

INSERT INTO cliente VALUES ('Alfredo', '555-5566')

Não informamos um valor para o campo id, pois, como ele é chave primária, é marcado com autoincremento, ou seja, sempre que for inserido um novo registro na tabela, o próprio SGBD verificará qual é o último valor de chave primária e irá incrementá-lo para atribuir ao novo registro (DEITEL, 2010).

Update

A instrução Update tem como objetivo modificar dados já existentes em uma tabela, como seu próprio nome diz, ele atualiza os dados. A sintaxe do Update tem, primeiramente, a tabela que será modificada; após, as colunas e os valores que serão atribuídos, no Update, apenas as colunas que terão seus valores modificados é que são especificadas; por fim, a cláusula WHERE para identificar qual linha da tabela será modificada. Caso não seja informada a cláusula WHERE, todas as linhas da tabela serão modificadas, o que pode causar inconsistência dependendo da estruturação do banco de dados e de quais colunas serão modificadas (DEITEL, 2010).

UPDATE nomeDaTabela

SET nomeColuna1 = valor1, nomeColuna2 = valor2, ..., nomeColunaN = valorN

WHERE critérios

Se existir mais de uma linha selecionada no critério especificado na cláusula WHERE, todas essas linhas serão modificadas.

DELETE

A instrução DELETE apaga as linhas da tabela. Assim como na instrução Update, a instrução DELETE tem a cláusula opcional WHERE, que especifica os critérios para uma linha ser removida. Caso a cláusula não seja especificada, todas as linhas da tabela serão apagadas (DEITEL, 2010).

DELETE FROM nomeTabela

WHERE critérios

Similar ao Update, se existir mais de uma linha no critério especificado na cláusula WHERE, todas as linhas serão excluídas.

Instalação e utilização de um SGBD

Nesta seção, veremos a utilização das principais características de um SGBD. Existem diversos tipos de SGBD disponíveis, tanto comerciais quanto versões com licenças gratuitas (GPL e/ou BSD).

Alguns SGBDs comerciais tem sua versão livre, não sendo necessário pagar para utilizá-la, porém, normalmente, essas versões grátis de um SGBD comercial têm menos ferramentas e algumas limitações, como quantidade de dados, memória utilizada, quantidade de banco de dados para gerenciar em um SGBD, dentre outras.

O SGBD Oracle tem a versão paga, que é a comercial, e a versão Express Edition (XE), que é a versão gratuita, com algumas limitações citadas anteriormente. No caso do MySQL, a versão comercial é chamada de MySQL Enterprise Edition, enquanto a versão gratuita é a MySQL Community Edition.

O SGBD MySQL Community Edition será utilizado nos exemplos desse material.

¶ Fique por dentro

O instalador do SGBD MySql está disponível no site: **dev.mysql.com <https://dev.mysql.com/downloads/mysql/>** .

Nesse site, estão disponíveis, também, para download as versões Enterprise e Community (que é a versão que utilizaremos). A instalação é oferecida pelo arquivo MSI, que tem o assistente de instalação, ou um arquivo zip.

As informações para instalação nos ambientes que são contemplados pelo MySQL podem ser encontradas no link a seguir, que é a documentação oficial acerca da instalação e da atualização do MySQL:

dev.mysql.com <https://dev.mysql.com/doc/refman/5.7/en/installing.html> .

O link faz menção, diretamente, à versão 5.7 (que foi utilizada nos exemplos), porém a documentação de outras versões pode ser encontrada na parte principal da página de documentação (**dev.mysql.com <https://dev.mysql.com/doc/>**).

Alguns instaladores de SGBD disponibilizam apenas o banco de dados, sem interface gráfica, então, todas as atividades a serem realizadas em um banco de dados devem ser feitas pela linha de comando com instruções SQL, o que pode ser um pouco complicado e oneroso para quem não é um DBA; portanto, é mais simples utilizar um SGBD com interface gráfica.

Gerenciamento básico do banco de dados com MySQL Workbench

O gerenciamento do banco de dados pode ser feito por meio do MySQL Workbench, que é a ferramenta de gerenciamento com interface gráfica. Caso queira se aventurar por linhas de comandos, é possível utilizar o MySQL Shell. O MySQL Workbench permite criar um novo banco de dados no servidor em execução, criar tabelas, campos, inserir dados, dentre outras configurações nos bancos de dados.

Para criarmos um novo banco de dados, é necessário criar um novo *schema* (que é como o MySQL chama um banco de dados). Para a criação desse *schema*, é necessário informar um nome único para o banco de dados e uma collation, que é a configuração dos caracteres que será utilizado em um banco de dados; normalmente, são utilizadas as *collations* UTF-8 ou Latin1, mas se deve ficar atento, pois determinadas *collations* não aceitam caracteres acentuados ou caracteres especiais de determinados idiomas, deixando o conteúdo em uma coluna do banco com caracteres ilegíveis. Após a criação, o ideal é definir esse banco como padrão, para que as ações sejam feitas nele, e não em outro banco já existente.

Um banco de dados necessita de tabelas para guardar os dados que serão utilizados em um sistema. As tabelas podem ser criadas com comandos SQL de CREATE TABLE, ou pelo Assistente de Criação de Tabelas do MySQL Workbench, que gera menos trabalho.

A vantagem do assistente é que as tabelas são criadas por meio de interface gráfica e, antes da criação, ele mostra o comando SQL que será utilizado para executar a operação.

Para a criação de tabelas, é necessário ter especificado quais colunas farão parte da tabela, quais serão seus índices (que são atributos para melhorar o desempenho de consultas na tabela) e as chaves estrangeiras. Essas características das tabelas devem ser especificadas no momento de realizar o projeto do sistema, para que não

ocorram problemas durante a codificação por falta de colunas em uma tabela, por colunas com tipagem incorreta, dentre outros problemas que podem ocorrer durante o desenvolvimento.

Para a criação de colunas, é necessário informar o nome da coluna (*Column Name*), o tipo de dados que a coluna receberá (*Datatype*) e alguns atributos (*flags*) para essa coluna. O atributo PK (*primary key*) indica que a coluna será uma chave primária, o atributo NN (*not null*) indica que essa coluna não aceita valor nulo, ou seja, é obrigatório informar valor, o atributo UQ (*unique*) cria um índice único para essa coluna, com isso, não é admitido valor repetido para essa coluna em outro registro da tabela. O atributo BIN (*binary*) indica que a coluna aceita *strings* binárias, em que as comparações são feitas pelos *bytes* da *string*. O atributo UN significa que o valor da coluna é "*unsigned*", que indica que a coluna aceitará somente valores não negativos, começando em 0. O flag ZF (*zero-fill*) completará o valor com zeros à esquerda, com isso, se o tamanho de um INT é 5 e o valor para aquela coluna é 10, ele será inserido na coluna como 00010, e o último flag é o AI (*auto-increment*), que indica que o campo terá autoincremento, que é bastante utilizado para chaves primárias (dependendo de como for estruturada essa chave primária).

Na criação de uma tabela, é possível especificar quais chaves estrangeiras (*Foreign Keys* - FK) essa tabela terá. No nosso exemplo, a tabela Endereco tem a coluna idcliente, que mantém o id do cliente que possui esse endereço. Para chaves estrangeiras, é possível determinar o comportamento do banco de dados no momento de excluir um registro que tenha referência em outras tabelas. Portanto, se um registro da tabela cliente for excluído, é possível solicitar ao banco de dados que exclua os registros que têm uma chave estrangeira vinculada à tabela cliente. Esse comportamento é definido por meio de ações de Update ou Delete na definição das chaves estrangeiras.

As ações podem ser **Restrict**, que impedirá a exclusão do registro caso haja um "filho", **Cascade**, em que o registro filho será deletado/atualizado também, **Set Null**, que irá setar o registro filho como nulo na coluna que se refere à FK, quando ela for

excluída, e **No Action**, em que não será repassada nenhuma ação ao registro filho.

Ao escolher algumas dessas ações para o Update ou Delete de registros com chave estrangeira, deve-se ter muito cuidado para não gerar dados inconsistentes no banco de dados, por exemplo, no caso das tabelas Cliente e Endereco; caso excluirmos um cliente, devemos verificar o que será feito com o endereço que pertence a esse cliente. Caso não fosse feita nenhuma ação, a tabela Endereco iria ter uma linha que possuiria um idcliente que não existe mais, gerando inconsistência. Esse é outro detalhe para tomar cuidado no momento de criação do projeto de banco de dados.

JDBC

O JDBC é um mecanismo implementado no Java para realizar a conectividade entre a aplicação e o banco de dados. O JDBC Java DataBase Connectivity é uma API que realiza a conexão e a comunicação diretamente com o banco de dados.

A API reúne diversas classes e interfaces que possibilitam a conexão com do banco de dados por meio de um driver específico do SGBD que está sendo utilizado. Esse driver realiza a interface de comunicação entre a aplicação e o SGBD, ou seja, ele realiza a tradução das mensagens trocadas com o protocolo SGBD.

Utilização do JDBC

O primeiro passo para utilizar o JDBC em um projeto Java é importar o driver do JDBC referente ao SGBD que está sendo utilizado para o projeto. Para adicionar o arquivo JAR no projeto, é necessário realizar o download do arquivo JAR do JDBC e descompactar o arquivo zip do driver em alguma pasta.

Fique por dentro

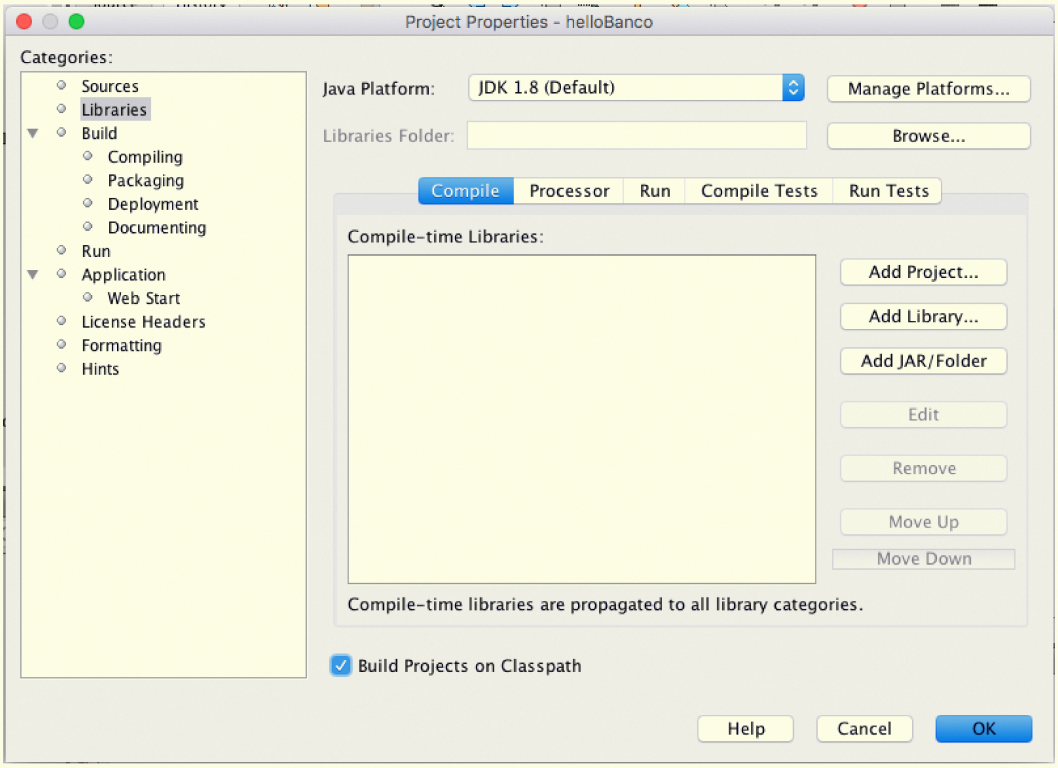
Os arquivos de driver do JDBC para MySQL, que foram utilizados nos exemplos, estão disponíveis para download no site: **<dev.mysql.com <https://dev.mysql.com/downloads/connector/j/> >**.

Além do MySQL, o Firebird é bastante utilizado, a seguir, está o endereço para realizar o download.

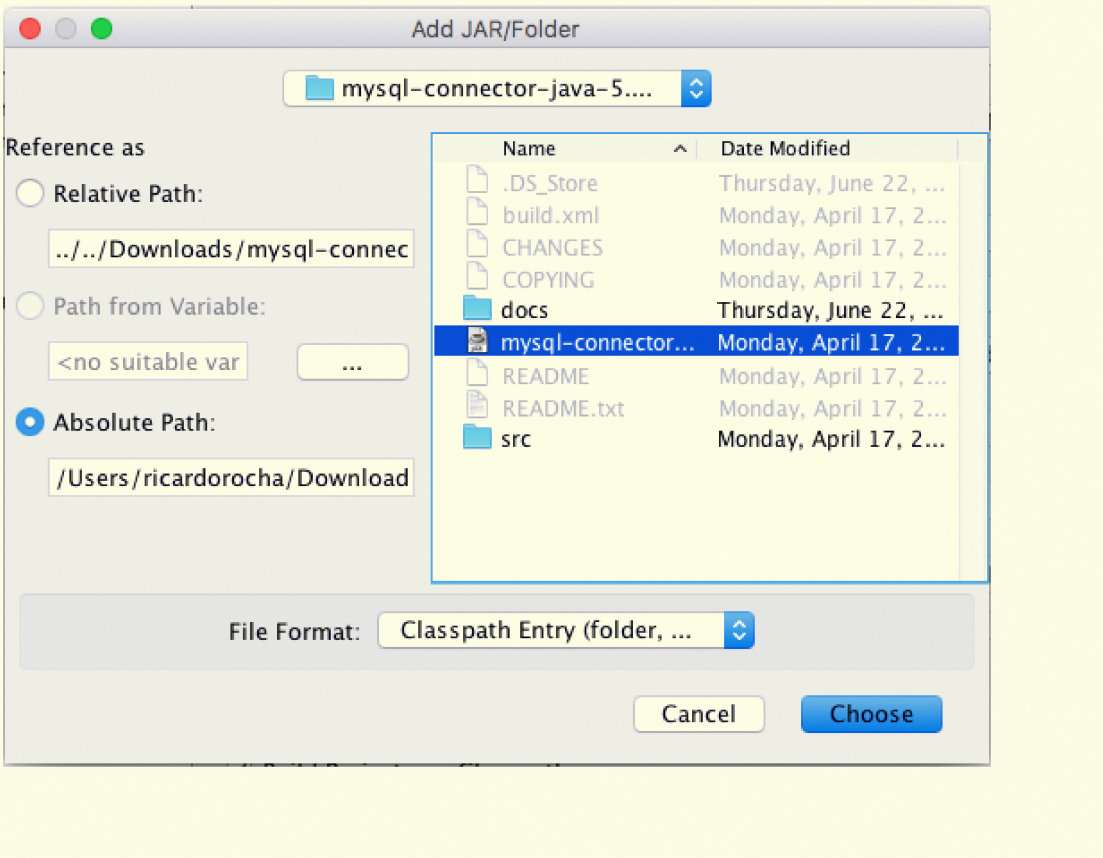
Disponível em: **www.firebirdsql.org <https://www.firebirdsql.org/en/jdbc-driver/>** .

Em um projeto Java no NetBeans é necessário adicionar o arquivo JAR do driver do MySQL. Isso é realizado nas propriedades do projeto, que é acessado por meio do clique com botão direito no projeto -> Propriedades.

A Figura 4.8 mostra a tela das propriedades do projeto. Nela, deverá ser acessado o item Bibliotecas (Libraries) e, então, clicar no botão Adicionar JAR/Pasta (Add JAR/Folder). Ao clicar nesse botão, a tela da Figura 4.9 será exibida. Nessa tela, deverá ser escolhido o arquivo JAR, que estará localizado na pasta em que o arquivo zip do driver foi descompactado. Após selecioná-lo, basta confirmar as telas das Figuras 4.9 e 4.8, respectivamente, e, então, o driver será exibido na pasta Bibliotecas (Libraries), que é exibida na árvore de visualização do projeto.



4FIGURA 8.26 - Tela de propriedades do projeto no NetBeans FONTE: NetBeans IDE.



4FIGURA 9.26 - Adicionar arquivo JAR FONTE: NetBeans IDE.

Pacote java.sql

Para o sistema realizar operações em um banco de dados, é necessário conectar-se ao banco. A conexão é realizada pelas interfaces do JDBC disponibilizadas no pacote *java.sql*. Nesse pacote, está a classe *DriverManager* e as interfaces *Driver, Connection, Statement, ResultSet e PreparedStatement*.

Classe DriverManager

A classe DriverManager é a responsável por registrar o driver que será utilizado com o banco de dados. O driver de um banco de dados é um arquivo com a extensão .jar que, normalmente, é encontrado nos sites dos SGBDs (VINICIUS1, 2013, on-

line).

Dentro da classe DriverManager existe o método getConnection, que retorna um objeto da interface Connection. O método tem três sobrecargas, exibidas na Figura 4.10. A primeira é somente o endereço do banco de dados, na segunda, além do endereço, é informada uma lista de propriedades para o banco (nessa lista, ao menos as propriedades *user* e *password* devem ser informadas), e, na terceira sobrecarga, os parâmetros são o endereço do banco de dados, o usuário e a senha para conexão.

```
public static Connection getConnection(String url)
public static Connection getConnection(String url, Properties info)
public static Connection getConnection(String url, String user, String password)
```

4FIGURA 10.26 - Sobrecargas do método getConnection FONTE: Vinicius1 (2013, *on-line*).

Interface Connection

O objeto da interface Connection que é retornado pelo método getConnection representa a conexão em si com o banco de dados. Esse objeto possui diversos métodos que interagem com a conexão, como o método close, que termina a conexão com o banco e libera um objeto Connection, normalmente utilizando dentro de um bloco *finally* (*try* ... *catch* ... *finally* ...). O método isClosed verifica se a conexão está fechada.

As instruções SQL são utilizadas por meio de dois métodos, createStatement e prepareStatement, ambos presentes na interface Connection.

O método createStatement implementa instruções simples de SQL em que não seja necessário realizar a passada de parâmetro. Esse método retorna um objeto do tipo Statement, que executa a instrução SQL.

O método prepareStatement cria um objeto que representará a instrução SQL a ser executada no banco de dados, porém essa instrução SQL pode receber parâmetros para ser executada. Esse método retorna um objeto do tipo PreparedStatement, que é a instrução SQL pré-compilada para ser executada no banco de dados. A execução ocorre pelo método execute da interface PreparedStatement. A classe PreparedStatement é uma subclasse (estende) da classe Statement (VINICIUS1, 2013).

Além dos métodos que criam objetos de instruções SQL, um objeto Connection tem métodos para realizar o *commit* ou o *rollback* das transações em um banco de dados.

Quando uma tabela for alterada por meio de Update, Insert ou Delete, é iniciada uma transação que irá bloquear as outras operações na tabela, para que não seja criada inconsistência no banco de dados (para mais informações, verificar o assunto Controle de Transações em livros acerca de Banco de Dados). Após realizar uma das operações de alteração de uma tabela, é necessário realizar o comando de *commit* no banco de dados, para que as alterações sejam gravadas (persistidas) e estejam disponíveis para os outros usuários. Caso ocorra um erro na alteração da tabela, é realizado um *rollback* e a tabela voltará ao estado que estava antes das alterações.

O método setAutoCommit faz com que todas as instruções que serão executadas sejam gravadas (comumente chamadas de comutadas). O padrão é que, quando as instruções de alteração (Update, Insert ou Delete) são finalizadas, a operação de *commit* é realizada automaticamente. Em uma operação de Select, o *commit* é realizado quando o conjunto de dados associados à consulta é fechado pelo método close (VINICIUS1, 2013, on-line).

Enquanto o *commit* realiza a gravação da alteração, o método *rollback* desfaz as alterações. A chamada do método *rollback* normalmente é realizada dentro de um bloco *catch* (*try* ... *catch* ...), pois, se um bloco *catch* foi executado, algum erro ocorreu, portanto, o mais prudente é desfazer as alterações.

A Figura 4.11 mostra um trecho do código da Figura 4.15 em que é utilizada a interface Connection. O objeto de conexão, do tipo Connection, é declarado na linha 20 e recebe a conexão com o banco de dados na linha 28, pelo método getConnection da classe DriverManager.

Com esse objeto de conexão, é possível criar os objetos que realizarão a interação com o banco de dados; neste exemplo, é criado um objeto do tipo Statement na linha 30 pelo método createStatement(), que está definido na interface Connection.

```
16      static final String DATABASE_URL = "jdbc:mysql://localhost/loja";
17      static final String QUERYCLIENTES = "SELECT id, nome FROM cliente";
18
19      public static void main(String args[]) {
20          Connection conn = null;
21          Statement statement = null;
22          ResultSet rs = null;
23
24          String nome;
25          int id;
26
27          try {
28              conn = DriverManager.getConnection(DATABASE_URL, "root", "admin");
29
30              statement = conn.createStatement();
```

4FIGURA 11.26 - Exemplo de utilização da interface Connection FONTE: NetBeans IDE.

Interface Statement

É a partir dessa interface que as instruções SQL são executadas, pois os métodos createStatement e prepareStatement, da interface Connection, retornam objetos do tipo Statement e PreparedStatement, respectivamente. A execução da instrução SQL é realizada por três métodos, execute, executeQuery e executeUpdate. O primeiro método executa qualquer instrução SQL, retornando um *booleano* que indica verdadeiro (*true*), se a instrução foi uma consulta e existe um ResultSet com os valores da consulta. Para retornar esse ResultSet, é utilizado o método .getResultSet(). Caso o retorno seja falso (*false*), o primeiro resultado é um contador de registros modificados, então, a instrução SQL foi Update/Insert/Delete, e é retornado pelo método .getUpdateCount(). O segundo método executa uma instrução SQL que retorna um objeto ResultSet com os dados retornados da consulta.

O terceiro método executa instruções SQL referentes aos comandos Insert, Update e Delete, retornando o número de registro afetado/alterado pela execução da instrução (VINICIUS1, 2013, on-line).

A Figura 4.12 mostra um exemplo utilizando o método executeUpdate.

```
String sql = "UPDATE funcionario SET nome=?, sobrenome=? WHERE codigo=?";
ps = conn.prepareStatement(sql);
ps.setString(1, nome);
ps.setString(2, sobrenome);
ps.setInt(3, codigo);
//Executa a instrução
int retorno = ps.executeUpdate();
if(retorno > 0){
        System.out.printf("Novo registro alterado: %d: %s - %s",codigo, nome,sobrenome);
}else{
        System.out.println("Não foi possível alterar os registros!");
}
```

4FIGURA 12.26 - Exemplo da utilização do método executeUpdate FONTE: Vinicius1 (2013, *on-line*).

Na primeira linha, é atribuído à *string* sql o texto da instrução SQL de Update da tabela Funcionario. Os caracteres ? são parâmetros que receberão valores para o comando SQL ser executado. Da linha 2 a 5, é criado um PreparedStatement por meio da instrução SQL e são setados os parâmetros pelos métodos referentes ao tipo do parâmetro (setString, para um parâmetro string, e setInt, para um parâmetro

numeral inteiro, lembrando de que os tipos dos dados dos parâmetros deverão ser compatíveis com os tipos dos dados que foram definidos para as colunas no banco de dados), informando para qual parâmetro será o valor; no exemplo, a terceira linha (ps.setString(1, nome)) indica que a variável nome será atribuída ao parâmetro 1 da instrução SQL.

A instrução é executada mediante o método executeUpdate(), que é invocado pelo objeto do tipo PreparedStatement, retornando um valor inteiro. Se o retorno for maior que zero, registros foram alterados, caso contrário, nenhum foi alterado.

Na Figura 4.13, é exibido um trecho do código que realiza uma consulta simples no banco de dados sem utilizar parâmetros. Na linha 17, é definida uma constante, que é a *query* escrita em linguagem SQL. Essa *query* é utilizada na linha 31 por meio do método executeQuery de um objeto do tipo Statement. Como não há parâmetros na *query* executada, o método executeQuery recebe apenas a variável que contém a *query*.

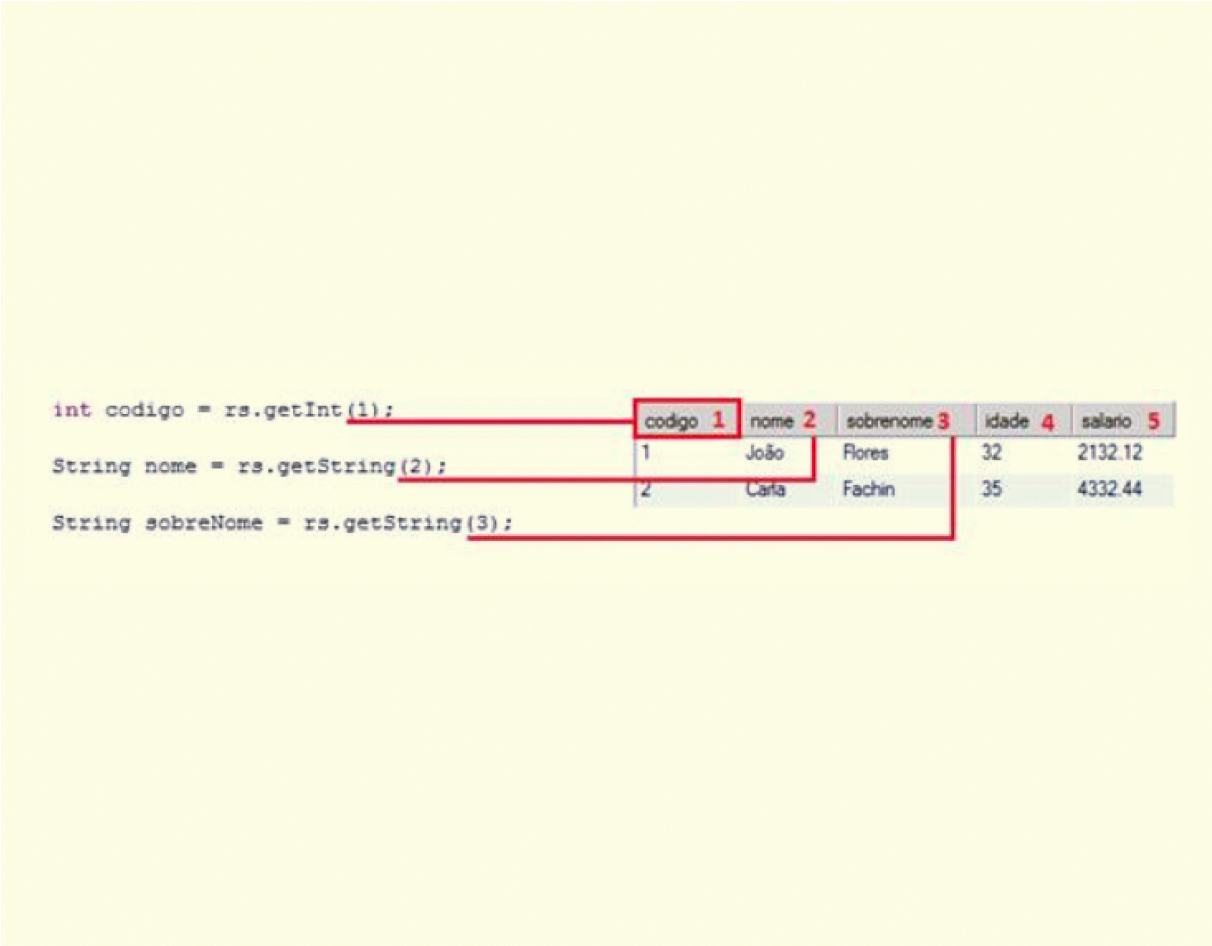
```
17      static final String QUERYCLIENTES = "SELECT id, nome FROM cliente";
31              rs = statement.executeQuery(QUERYCLIENTES);
32
33              while (rs.next()) {
34                  nome = rs.getString("nome");
35                  id = rs.getInt("id");
36
37                  System.out.printf("Cliente: id: %d Nome: %s \n", id, nome);
38              }
```

4FIGURA 13.26 - Exemplo de execução de query com o método executeQuery FONTE: NetBeans IDE.

Interface ResultSet

Um ResultSet é um conjunto com os registros que são resultantes de uma consulta em uma tabela no banco de dados. É um objeto que representa uma tabela, com seu cursor posicionado antes do primeiro registro no início; quando utilizado o método next(), o cursor apontará para a próxima linha de dados (VINICIUS1, 2013, on-line).

O objeto ResultSet tem métodos assessores (getters) para retornar os valores referentes aos tipos de dados da coluna, que devem ser utilizados em correspondência com os tipos de dados da coluna no banco de dados. A Figura 4.14 mostra a utilização dos métodos *getters* em um objeto ResultSet rs.



4FIGURA 14.26 - Recuperação de valores em um ResultSet FONTE: Vinicius1 (2013, *on-line*).

Para recuperar um valor de uma coluna que é do tipo inteiro, é utilizado o método getInt(1), no qual 1 indica que é a primeira coluna, porém se pode utilizar o nome da coluna ao invés de seu índice, por exemplo, rs.getInt("codigo").

Para retornarmos o resultado da coluna nome que é *string*, é utilizado o método getString(2) ou getString("nome"), atribuindo o retorno em uma *string*.

Na Figura 4.15, é exibido outro exemplo da utilização de um ResultSet. Esse ResultSet é obtido por meio da execução do método executeQuery (linha 31) e, na linha 33, ele é posicionado no primeiro registro pela execução do método next() da classe ResultSet. O laço de iteração da linha 33-38 será executado apenas enquanto

o objeto ResultSet conter resultados, ou seja, a cada iteração o ponteiro do ResultSet será apontado para a próxima linha. Quando não houver mais resultados, o método next() retornará false e a iteração será interrompida.

As linhas 34 e 35 da Figura 4.15 realizaram a atribuição dos valores da consulta para as variáveis nome e id, respectivamente. É importante observar que, para atribuir o nome, que é uma String, para a variável nome, é utilizado o método getString("nome"), respeitando o tipo do dado no banco de dados e o tipo de dado declarado na variável. Para atribuir o id, é utilizado o método getInt("id"), pois o id foi definido no banco de dados como um inteiro.

Exemplo de consulta ao banco de dados

A Figura 4.15 mostra o código de uma classe de teste de conexão e consulta básica no banco de dados que foi criado na seção anterior. Dessa figura, foram retirados trechos de código para explanação de métodos importantes para a interação com banco de dados no Java.

No exemplo da Figura 4.15, é realizada uma conexão com o banco de dados e, após conectado, é executada uma consulta SQL no banco. Nas linhas 16 e 17, são definidas duas constantes, uma com o endereço do banco de dados e a outra constante com a *query* de consulta que será executada.

As linhas 20 a 22 declaram os objetos de Connection, Statement e ResultSet. O objeto Connection realizará a conexão com o banco de dados pela interface DriverManager.getConnection, passando a *string* do endereço do banco de dados, usuário e senha de conexão (linha 28). Após ser realizada a conexão, é criado um Statement para realizar as consultas no banco de dados por meio do método executeQuery, retornando um ResultSet com os dados (linha 31).

Como o ResultSet é um conjunto de registros, os dados são consultados no ResultSet dentro de um loop para movimentar o cursor. O método next() do ResultSet posiciona o cursor no próximo registro (o cursor de um ResultSet é posicionado

inicialmente antes do primeiro registro) e é realizada a iteração nos registros por meio de um *while* (linha 33).

TesteConexao.java

Source History

```
14  public class TesteConexao {
15
16      static final String DATABASE_URL = "jdbc:mysql://localhost/loja";
        static final String queryClientes = "SELECT id, nome FROM cliente";
18
19      public static void main(String args[]) {
20          Connection conn = null;
21          Statement statement = null;
22          ResultSet rs = null;
23
24          String nome;
25          int id;
26
27          try {
28              conn = DriverManager.getConnection(DATABASE_URL, "root", "admin");
29
30              statement = conn.createStatement();
31              rs = statement.executeQuery(queryClientes);
32
33              while (rs.next()) {
34                  nome = rs.getString("nome");
35                  id = rs.getInt("id");
36
37                  System.out.printf("Cliente: id: %d Nome: %s \n", id, nome);
38              }
39          } catch (SQLException ex) {
              ex.printStackTrace();
41          } finally {
42              try {
                    rs.close();
                    statement.close();
                    conn.close();
                } catch (Exception ex) {
                    ex.printStackTrace();
48              }
49          }
50
51      }
52  }
```

4FIGURA 15.26 - Exemplo de interação com banco de dados FONTE: NetBeans IDE.

Quando não houver mais registros para iterar, o método *next* retorna false e o *loop* é interrompido. Os dados do ResultSet são obtidos pelos métodos *getters*. O método para retornar o id é getInt("id"), ou poderíamos utilizar getInt(1), já que a coluna id é a coluna 1 do conjunto e, para retornar nome, é utilizado getString("nome") (linhas 34-35).

Todo o processo de conexão e execução da instrução SQL é envolvido em um bloco *try ... catch ... finally*, pois pode ocorrer alguma exceção do tipo SQLException (a própria IDE exibe um *warning* para envolvermos o trecho de código referente ao banco de dados em um *catch* com SQLException).

No bloco *finally*, temos a liberação da conexão e dos objetos envolvidos diretamente com acesso ao banco de dados. É necessário verificar se os objetos foram instanciados corretamente antes de chamar o método close(), pois, caso não tenham sido instanciados, esse comando poderá lançar um NullPointerException.

Como esse exemplo é em um projeto sem interface gráfica, o resultado da consulta será exibido por meio de linhas impressas no console. Para imprimirmos o resultado, foi utilizado o método System.out.printf, que exibe uma *string* formatada. Foram utilizados os curingas %d e %s na *string* de impressão e, então, passadas por parâmetros as variáveis que seriam impressas, id e nome, respectivamente.

No site da API do Java, são informados quais os conversores possíveis para os tipos de dados (**docs.oracle.com <https://docs.oracle.com/javase/tutorial/java/data/numberformat.html>**).

DAO e MVC

Diversas vezes nos deparamos com situações em que operações específicas, por exemplo, acesso a banco de dados relacionais (CRUD: *Create, Retrieve, Update, Delete* - Criar, Recuperar, Atualizar, Excluir), estão embutidas em locais que não são adequados para essas operações.

O caso das operações relacionadas ao CRUD foi utilizado como exemplo, pois é muito comum vermos uma interface gráfica com botões para inserir um novo registro no banco de dados, atualizá-lo ou removê-lo, com a codificação de acesso ao

banco de dados sendo feita diretamente no código do formulário (ou da página de aplicação, caso seja um ambiente web), em outras palavras, o acesso ao banco está diretamente (ou quase) atrelado ao evento de clique do botão.

O programa funcionará normalmente, porém poderá gerar vários problemas à frente, quando o sistema crescer e demandar manutenção das classes. Além disso, não existe nenhuma separação de responsabilidades, fazendo com que diversas operações sejam executadas somente em um local (objeto) (MACORATTI, [2017], on-line).

Esses problemas não ocorrem somente com acesso a banco de dados, mas podem acontecer em diversas situações, principalmente na programação orientada a objetos. Muitos desses problemas têm soluções que estão ligadas diretamente à estruturação do programa, à análise e à solução proposta ao problema.

Os padrões de projeto (*design patterns* em inglês) são formas para solucionar problemas que ocorrem frequentemente dentro de um projeto de *software*. Com foco em tornar mais flexível e diminuir a complexidade de manutenção, os padrões de projeto podem ser vistos como um mapa de como melhorar a implementação quando uma determinada característica recorrente acontece no sistema (FERREIRA, 2005).

Existem padrões de projeto para diversos problemas que podem ocorrer, como situação na criação de um objeto, problemas comportamentais, por exemplo, adicionar funcionalidades dinamicamente, problemas de acesso a objetos, isolando a responsabilidade de cada objeto, dentre outros.

Estudaremos o padrão de projeto DAO, que tem como objetivo isolar as operações relacionadas ao banco de dados dos outros objetos.

Na parte de indicação de leitura, há um livro que aborda diversos padrões de projeto que podem ser úteis em diversas situações na estruturação de um programa. Alguns dos padrões de projeto bastante utilizados são Singleton, Factory Method, Decorator, Façade, Composite, observando que cada um tem sua aplicação distinta.

POJO

POJOs (*Plain Old Java Objects*), em tradução livre, seria "Simples e Velhos Objetos Java", ou seja, eles são classes Java do modo mais simples possível. As classes de domínio que são utilizadas no padrão DAO são POJOs. Elas teriam apenas seu construtor, os atributos e os métodos acessores (*getters* e *setters*), que são responsáveis por atribuir ou retornar os valores dos atributos. É importante ressaltar que o Java tem um padrão de nomenclatura para os métodos *getters* e *setters*, no qual os getters começam sempre com a palavra get e, então, o nome do atributo que retornará (getId, getNome, getCodigo), enquanto os setters começam com set e o nome do atributo (setId, setNome, setCodigo). A Figura 4.16 mostra o POJO, ou a classe de domínio, Cliente que será utilizado em nosso exemplo.

Na maioria dos casos, os métodos acessores dos POJOs são todos públicos e pode não haver nenhum tratamento lógico ou condicional dentro desses métodos, o que faz com que não haja realmente a aplicação do encapsulamento.

Normalmente, os POJOs são utilizados como definidores de classes de domínio e como objetos padrões para utilização de diversos *frameworks*, como Hibernate ou Spring.

```java
package br.com.ricardor.loja.model;

public class Cliente {

    private int id;
    private String nome;
    private String telefone;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

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

    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }
}
```

4FIGURA 16.26 - POJO Cliente FONTE: NetBeans IDE.

DAO

O DAO (*Data Access Object*) é um padrão de projeto que tem o objetivo de desacoplar o código de acesso e a persistência dos dados da lógica da aplicação. Com essa separação, o código tende a ficar mais organizado e a aumentar sua reutilização em outras aplicações (MACORATTI, [2017], on-line).

A estruturação do DAO permite que classes de dados sejam criadas independente de qual seja a fonte de dados (banco de dados, arquivos XML, arquivos texto, dentre outras), encapsulando o mecanismo de acesso a dados e criando uma interface genérica que definirá o comportamento de como os dados serão acessados.

Com a interface, é possível que os mecanismos de acesso a dados sejam alterados independente de como o código utiliza os objetos que acessarão os dados (MACORATTI, [2017], on-line).

Ou seja, é possível que a aplicação seja isolada da API que está realizando a comunicação dos dados. Porventura, futuramente, pode ser necessário alterar a API utilizada para acessar os dados ou, inclusive, trocar de SGBD, sem ser necessário interferir no restante do código da aplicação.

As principais características do padrão DAO são (MACORATTI, [2017], on-line):

- toda a comunicação com o mecanismo de persistência (ou seja, todo acesso aos dados) deverá ser realizada por classes DAO.
- cada instância da DAO é responsável por um objeto de domínio, ou seja, existirá um objeto DAO para cada domínio (tabela) necessário para ser realizado o acesso dos dados.
- as operações relacionadas ao acesso aos dados (CRUD) é de responsabilidade total do DAO, ou seja, é no objeto DAO que estará a implementação de acesso aos dados no banco.

A Figura 4.17 mostra a definição de como é estruturado um DAO para o domínio Cliente. Primeiro, há a definição da classe Cliente, com seus atributos e os acessores. É essa classe que terá os valores que serão acessados no banco de dados, a sua implementação está exibida na Figura 4.16 (nessa figura, a implementação da classe de domínio foi suprimida para a figura não ficar muito extensa).

A segunda classe (ClienteDAO) é a interface que definirá como a classe DAO para o Cliente se comportará. Ela terá métodos que realizarão o acesso ao banco de dados. Os métodos de busca, atualização e exclusão recebem um objeto Cliente como parâmetro, que conterá os dados que serão utilizados para o acesso ao banco. Os

métodos de busca (findBy) recebem por parâmetro o critério que será utilizado na busca, como o método findById, que realizará a busca por meio do Id de um Cliente, seu parâmetro é o id do tipo inteiro.

A terceira classe (JDBCClienteDAO) é a implementação da interface definida anteriormente. É nela que serão escritos os comandos SQL que serão utilizados via JDBC para realizar o acesso ao banco de dados. Caso fosse utilizada outra forma de acessar os dados, por exemplo, por meio de um arquivo XML, o código de leitura desse arquivo XML seria escrito nessa classe.

Em alguns casos, pode ser interessante alterar a implementação do DAO e, ao invés de utilizar uma interface para definir o comportamento dos objetos DAO, utilizar uma classe abstrata, porém com implementações que serão padrões para todo DAO, como requerer uma conexão, preparar uma lista de parâmetros para executar determinado comando SQL. Essas implementações podem ser realizadas na classe abstrata e, então, as classes DAO referentes às classes de domínio estenderiam a classe abstrata. Utilizaremos esse modelo no exemplo para facilitarmos a implementação.

```java
public class Cliente {
    //atributos
    //acessores
}

public interface ClienteDAO {
    Cliente create();
    void insert(Cliente c);
    void update(Cliente c);
    void delete(Cliente c);
    Cliente findById(Integer id);
    Cliente findByNome(String nome);
}


public class JDBCClienteDAO implements ClienteDAO {
    @Override
    public Cliente create() {
        return new Cliente();
    }
    @Override
    public void insert(Cliente c) {
        //Executar comando de INSERT utilizando o JDBC
    }
    @Override
    public void update(Cliente c) {
        //Executar comando de UPDATE utilizando o JDBC
    }
    @Override
    public void delete(Cliente c) {
        //Executar comando de DELETE utilizando o JDBC
    }
    @Override
    public Cliente findById(Integer id) {
        //Executar a busca de um cliente pelo id
        //utilizando o JDBC
    }
    @Override
    public Cliente findByNome(String nome) {
        //Executar a busca de um cliente pelo nome
        //utilizando o JDBC
    }
}
```

4FIGURA 17.26 - Estruturação do DAO para o domínio Cliente FONTE: NetBeans IDE.

MVC

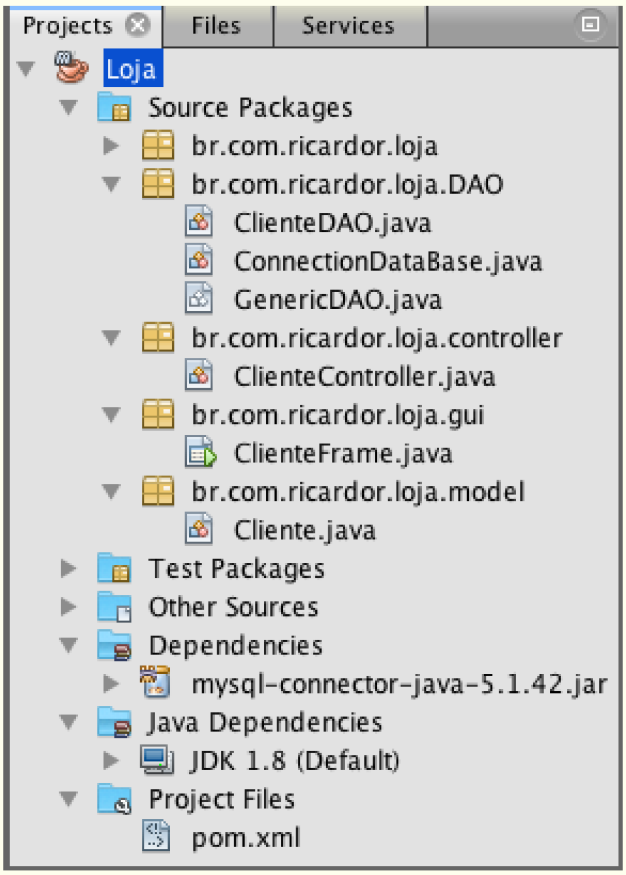
O MVC é um padrão que consiste em separar os dados e/ou a lógica de negócios da interface do usuário e do fluxo da aplicação. MVC é o acrônimo de *Model View Controller*, em que a aplicação é dividida em camadas. A camada Modelo (*Model*) é a que manipula as informações de forma mais detalhada, contém as regras de negócio e os cálculos que o sistema pode fazer. É ela que tem acesso a informações proveniente de alguma fonte de dados (BALLEM, 2011). A camada Visão (*View*) é responsável por tudo o que é exibido para o usuário, ou seja, tudo o que ele visualiza. É a interface gráfica do sistema (BALLEM, 2011). A camada de Controle (*Controller*), por sua vez, é a que faz o controle do fluxo da aplicação. Ela decide como o sistema se comportará, quais requisições devem ser feitas, quais regras devem ser acionadas, ou seja, toda a forma como o sistema executará as operações.

A vantagem da utilização do MVC é que as operações são desacopladas e cada camada realiza operações referentes apenas ao seu escopo. O objetivo do MVC é aumentar a reutilização do código, demandando o mínimo possível de alterações. Supomos que uma aplicação seja voltada totalmente para execução desktop, porém é necessário convertê-la para funcionamento em ambiente Web. Caso ela esteja estruturada e implementada obedecendo ao conceito de MVC, será necessário alterar a *View* (que será desenvolvida em linguagem Web) e realizar alguns ajustes na camada *Controller*, pois aplicações Web se comportam de maneira diferente de desktop. Entretanto, se o projeto foi implementado sem seguir o padrão MVC e as regras de negócio e os acessos ao banco de dados fossem implementados diretamente na interface gráfica, seria necessário reescrever a aplicação para ser executada na Web.

É comum o MVC ser confundido com o DAO, porém o MVC é um padrão arquitetural de um sistema, ou seja, é como o sistema será projetado para diminuir a acoplação da regra de negócio do sistema com sua interface gráfica, dividindo o sistema em camadas. Já o DAO é um padrão de projeto que indica como deve ser realizada a estruturação dos objetos do sistema para que o acesso aos dados fique isolado e, em uma situação de alteração da fonte de dados (SGBD, XMLs, arquivos textos etc.), não afete diretamente o resto do sistema. Muitas vezes, o DAO é implementado em cima de um sistema que foi estruturado com o MVC, realizando o acesso aos dados na camada *Model*, porém em objetos diferentes dos que detêm as regras de negócio e toda a lógica do sistema.

Exemplo utilizando DAO e MVC

As figuras a seguir (4.18 a 4.24) mostram exemplos de CRUD com a tabela Cliente do nosso banco de dados, utilizando uma interface gráfica implementada usando o padrão de projeto DAO e estruturada no padrão MVC.



4FIGURA 18.26 - Arquivos e estruturação do projeto FONTE: NetBeans IDE.

A Figura 4.18 mostra como foram feitas as divisões das classes nos pacotes do projeto. O projeto tem quatro pacotes em que estão divididas as classes por meio da sua funcionalidade. O pacote br.com.ricardor.loja.DAO abriga as classes que fazem as operações referentes ao banco de dados (GenericDAO, ClienteDAO e ConnectionDataBase). No pacote br.com.ricardor.loja.Controller, contém a classe ClienteController, que faz o relacionamento entre a interface gráfica e os objetos do banco de dados. O pacote br.com.ricardor.loja.gui abriga a classe que implementa a interface gráfica do projeto; caso o projeto possuísse mais telas de interface gráficas, elas ficariam localizadas nesse pacote. No último pacote br.com.ricardor.loja.Model, contém as classes que representarão as entidades de domínio do projeto (é importante observar que esse *model* não é o mesmo *model* do modelo MVC).

O modelo MVC, como vimos, é dividido em *Model*, *View* e *Controller*; aplicando no projeto, teríamos a seguinte classificação dos arquivos:

- ***Model***: contém as classes GenericDAO, ClienteDAO, Cliente e ConnectionDataBase, que são as responsáveis pela lógica do sistema.
- ***View***: é a interface do sistema, ou seja, o que será exibido ao usuário, nela, está a classe ClienteFrame.
- ***Controller***: faz o relacionamento entre a interface e a lógica, no nosso sistema, o responsável por isso é o ClienteController.

A Figura 4.19 mostra a implementação da classe que realiza a conexão com o banco de dados por meio do JDBC. Nas linhas 9-12, foram definidas constantes com o endereço do banco de dados, o driver que será utilizado (no caso, MySql), o usuário e a senha para conexão. No método getConnection(), é retornado um objeto do tipo Connection, que é instanciado pelo DriverManager (linha 19). A linha 18 "carrega" o driver que será utilizado.

```java
package br.com.ricardor.loja.DAO;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionDataBase {

    private static final String URL_MYSQL = "jdbc:mysql://localhost/loja";
    private static final String DRIVER_CLASS = "com.mysql.jdbc.Driver";
    private static final String USER = "root";
    private static final String PASS = "admin";

    public static Connection getConnection() {
        System.out.println("Conectando ao Banco de Dados");

        try {
            Class.forName(DRIVER_CLASS);
            return DriverManager.getConnection(URL_MYSQL, USER, PASS);
        } catch (ClassNotFoundException | SQLException ex) {
            ex.printStackTrace();
        }

        return null;
    }
}
```

4FIGURA 19.26 - Classe ConnectionDataBase FONTE: adaptada de Ballem (2011).

Fique por dentro

A partir do Java 6, não é mais necessário carregar explicitamente o driver por meio do Class.forName() para registrar um driver JDBC. O código que carrega explicitamente o driver foi colocado para conhecimento, caso esse trecho de código seja encontrado em algum projeto mais antigo.

Mais informações em: **http://www.onjava.com <http://www.onjava.com/pub/a/onjava/2006/08/02/jjdbc-4-enhancements-in-java-se-6.html>** .

A classe GenericDAO, exibida na Figura 4.20, é uma classe genérica para realizar operações em banco de dados (BALLEM, 2011). Todos os métodos são *protected*, fazendo com que apenas as classes que sejam derivadas dessa classe poderão acessá-los. A conexão com o banco de dados é realizada no construtor da classe e atribuída ao atributo Connection da classe. Os métodos *save*, *update* e *delete* recebem um comando SQL por parâmetro que conterá a instrução SQL que será executada pelo método. O outro parâmetro é uma lista de objetos de tamanho variável (Object... parametros), ou seja, os três pontos (...) após o Object indicam que poderá ser passado quantos objetos sejam desejados, desde 0 até n objetos. Essa funcionalidade é chamada de ***varargs*** e mais informações podem ser encontradas no seguinte endereço: **http://docs.oracle.com <http://docs.oracle.com/javase/7/docs/technotes/guides/language/varargs.html>** .

O método *update* recebe um parâmetro a mais, que é o id do item que será atualizado.

```java
package br.com.ricardor.loja.DAO;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public abstract class GenericDAO {

    private Connection connection;

    protected GenericDAO() {
        this.connection = ConnectionDataBase.getConnection();
    }

    protected Connection getConnection() {
        return connection;
    }

    protected void save(String insertSql, Object... parametros) throws SQLException {
        PreparedStatement pstmt = getConnection().prepareStatement(insertSql);

        for (int i = 0; i < parametros.length; i++) {
            pstmt.setObject(i+1, parametros[i]);
        }

        pstmt.execute();
        pstmt.close();
    }

    protected void update(String updateSql, Object id, Object... parametros) throws SQLException {
        PreparedStatement pstmt = getConnection().prepareStatement(updateSql);

        for (int i = 0; i < parametros.length; i++) {
            pstmt.setObject(i+1, parametros[i]);
        }

        pstmt.setObject(parametros.length + 1, id);
        pstmt.execute();
        pstmt.close();
    }

    protected void delete(String deleteSql, Object... parametros) throws SQLException {
        PreparedStatement pstmt = getConnection().prepareStatement(deleteSql);

        for (int i = 0; i < parametros.length; i++) {
            pstmt.setObject(i+1, parametros[i]);
        }

        pstmt.execute();
        pstmt.close();
    }
}
```

4FIGURA 20.26 - Classe GenericDAO FONTE: adaptada de Ballem (2011).

A Figura 4.21 mostra a classe ClienteDAO que é a implementação específica do acesso aos dados; essa classe estende da classe genérica de dados GenericDAO. Nos métodos de acesso aos dados é escrita a instrução SQL que será executada e, então, é invocado o método implementado na superclasse (GenericDAO) com a *string* referente ao SQL e os parâmetros necessários. Caso não tivéssemos a classe GenericDAO, as operações executadas nela poderiam ser implementadas na classe ClienteDAO, porém, se nosso projeto tivesse mais uma classe DAO, como EnderecoDAO, essas operações teriam que ser reescritas, com isso, o objetivo da classe genérica é diminuir o retrabalho das operações comuns entre diversas classes.

Além dos métodos de salvar, alterar e excluir, existem os métodos de buscas findByName, findById e findClientes. Os métodos findByName e findById realizam a busca utilizando parâmetro, nome e id, respectivamente, retornando um objeto Cliente, e o método findClientes busca todos clientes da tabela, retornando uma lista de Clientes. Nessa classe, poderiam ter mais métodos implementados, como buscas por critérios específicos.

```java
package br.com.ricardor.loja.DAO;

import br.com.ricardor.loja.model.Cliente;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

public class ClienteDAO extends GenericDAO {

    public void salvar(Cliente cliente) throws SQLException {
        String insert = "INSERT INTO CLIENTE(NOME, TELEFONE) "
                + "VALUES (?,?)";

        save(insert, cliente.getNome(), cliente.getTelefone());
    }

    public void excluir(int id) throws SQLException {
        String delete = "DELETE FROM CLIENTE WHERE id = ?";

        delete(delete, id);
    }

    public void alterar(Cliente cliente) throws SQLException {
        String update = "UPDATE CLIENTE "
                + "SET NOME = ?, TELEFONE = ? "
                + "WHERE id = ?";
        update(update, cliente.getId(), cliente.getNome(), cliente.getTelefone());
    }

    public List<Cliente> findClientes() throws SQLException {
        List<Cliente> clientes = new ArrayList<Cliente>();

        String select = "SELECT * FROM CLIENTE";

        PreparedStatement stmt =
                        getConnection().prepareStatement(select);

        ResultSet rs = stmt.executeQuery();

        while (rs.next()) {
            Cliente cliente = new Cliente();
            cliente.setId(rs.getInt("id"));
            cliente.setNome(rs.getString("nome"));
            cliente.setTelefone(rs.getString("telefone"));
            clientes.add(cliente);
        }

        rs.close();
        stmt.close();

        return clientes;
    }

    public Cliente findByName(String nome) throws SQLException {
        String select = "SELECT * FROM CLIENTE WHERE nome = ?";
        Cliente cliente = null;
        PreparedStatement stmt =
                        getConnection().prepareStatement(select);

        stmt.setString(1, nome);
        ResultSet rs = stmt.executeQuery();

        while (rs.next()) {
            cliente = new Cliente();
            cliente.setId(rs.getInt("id"));
            cliente.setNome(rs.getString("nome"));
            cliente.setTelefone(rs.getString("telefone"));
        }

        rs.close();
        stmt.close();
        return cliente;
    }

    public Cliente findById(int id) throws SQLException {
        String select = "SELECT * FROM CLIENTE WHERE id = ?";
        Cliente cliente = null;
        PreparedStatement stmt =
                        getConnection().prepareStatement(select);

        stmt.setInt(1, id);
        ResultSet rs = stmt.executeQuery();

        while (rs.next()) {
            cliente = new Cliente();
            cliente.setId(rs.getInt("id"));
            cliente.setNome(rs.getString("nome"));
            cliente.setTelefone(rs.getString("telefone"));
        }

        rs.close();
        stmt.close();
        return cliente;
    }
}
```

4FIGURA 21.26 - Classe ClienteDAO FONTE: adaptada de Ballem (2011).

A classe de controle, ClienteController, é exibida na Figura 4.22. Assim como a classe ClienteDAO, ela terá um método para cada método de acesso ao banco de dados implementado na classe ClienteDAO, mas fazendo o relacionamento com a interface. Caso, no futuro, seja alterado algo no sistema, como a interface gráfica, e o projeto seja portado para a Web, as classes DAO não precisarão ser alteradas, apenas a interface e algumas alterações nas classes Controllers. Os métodos da classe ClienteController recebem os parâmetros referentes ao cliente que serão necessários para as instruções SQL desejadas (nome, id e/ou telefone) e, então, realiza a chamada dos métodos por meio da classe ClienteDAO.

```java
package controller;

import br.com.ricardor.loja.DAO.ClienteDAO;
import br.com.ricardor.loja.model.Cliente;
import java.sql.SQLException;
import java.text.ParseException;
import java.util.List;
import javax.swing.JOptionPane;

public class ClienteController {

    public void salvar(String nome, String telefone)
            throws SQLException, ParseException
    {
        Cliente cliente = new Cliente();
        cliente.setNome(nome);
        cliente.setTelefone(telefone);

        new ClienteDAO().salvar(cliente);
    }

    public void alterar(int id, String nome, String telefone)
            throws ParseException, SQLException
    {

        Cliente cliente = new Cliente();
        cliente.setId(id);
        cliente.setNome(nome);
        cliente.setTelefone(telefone);

        new ClienteDAO().alterar(cliente);
    }

    public List<Cliente> listarClientes() {
        ClienteDAO dao = new ClienteDAO();
        try {
            return dao.findClientes();
        } catch (SQLException e) {
            JOptionPane.showMessageDialog(null,
                                "Problemas ao localizar cliente\n" +
                                e.getLocalizedMessage()
                            );
        }
        return null;
    }

    public void excluir(int id) throws SQLException {
        new ClienteDAO().excluir(id);
    }

    public Cliente buscaClientePorNome(String nome) throws SQLException {
        ClienteDAO dao = new ClienteDAO();
        return dao.findByName(nome);
    }

    public Cliente buscaClientePorId(int id) throws SQLException {
        ClienteDAO dao = new ClienteDAO();
        return dao.findById(id);
    }
}
```

4FIGURA 22.26 - Classe ClienteController FONTE: adaptada de Ballem (2011).

As Figuras 4.23, 4.24 e 4.25 compõem a classe da interface gráfica construída por meio do Swing e da ferramenta de desenho de interface do NetBeans. Os códigos autogerados do NetBeans foram escondidos para diminuir o tamanho da classe e, como eles são gerados automaticamente pela IDE quando o componente é posicionado e suas propriedades são setadas, não é necessária sua exibição **github.com <https://github.com/oricardorocha/CrudEAD>** . A Figura 4.26 mostra como ficou a interface gráfica implementada, e a implementação do desenho da interface (seja pela ferramenta de desenho do NetBeans ou por linhas de código) fica como exercício para praticar.

As linhas 201, 205, 209, 213, 217 e 221 são os métodos vinculados aos eventos de clique dos botões e da tabela que exibe a lista dos clientes.

Na classe da GUI, temos como atributos para o funcionamento correto da interface uma lista dos clientes da tabela e um objeto inteiro *key*, que terá o id do registro atual que está sendo consultado/modificado. Os métodos para salvar, excluir e localizar utilizam as informações dos objetos TextField da interface gráfica e passam por parâmetro para os métodos do objeto ClienteController, que é instanciado no momento do clique do botão. Para salvar e atualizar, é utilizado o mesmo método (onClickSalvar), mas é verificado se o atributo *key* tem valor, caso tenha, é para atualizar o registro atual, se o valor for 0, um novo registro será inserido.

O método localizarClientes() é chamado em qualquer operação que realize alteração no banco de dados (salvar, excluir) e, também, ao iniciar a interface, para recarregar os dados que são exibidos na tabela. Esse método remove todos os dados que estão exibidos na tabela, para garantir que não fique nenhum dado incorreto, e realiza a busca de todos os clientes no banco de dados. Os dados na tabela são controlados pelo TableModel, que é definido nas propriedades da tabela. Para o exemplo, utilizamos o DefaultTableModel, que é o modelo padrão, porém não é o mais fácil e nem o mais completo para executar diversas operações. Para utilização de JTable em aplicações mais complexas, o ideal é definir um TableModel próprio, com características próprias dos dados que serão exibidos, por

exemplo, tivemos que exibir o id do registro na tabela, pois precisamos do id para a atualização e a exclusão dos registros. No DefaultTableModel, tornar-se-ia muito complexo ocultar a coluna id, mas utilizar seu valor.

¶ Fique por dentro

Uma solução para não utilizar o DefaultTableModel é um projeto criado por Mark Vasconcelos que implementa uma ObjectTableModel para ser utilizada em diversos casos. Para mais informações, os links a seguir explicam a utilização e disponibilizam os arquivos necessários:

markytechs.wordpress.com <https://markytechs.wordpress.com/2009/05/29/objecttablemodel/>.

code.google.com <https://code.google.com/archive/p/towel/wikis/ObjectTableModel.wiki>.

```
1   package br.com.ricardor.loja.gui;
2
3   import br.com.ricardor.loja.model.Cliente;
4   import controller.ClienteController;
5   import java.sql.SQLException;
6   import java.text.ParseException;
7   import java.util.List;
8   import java.util.logging.Level;
9   import java.util.logging.Logger;
10  import javax.swing.JOptionPane;
11  import javax.swing.table.DefaultTableModel;
12
13  public class ClienteFrame extends javax.swing.JFrame {
14
15      /** Creates new form ClienteFrame ...3 lines */
18      public ClienteFrame() {
19          initComponents();
20          localizarClientes();
21          tblClientes.setRowSelectionAllowed(true);
22      }
23
24      /** This method is called from within the constructor to initialize the form ...5 lines */
29      @SuppressWarnings("unchecked")
30      Generated Code
199
200     private void btnSalvarActionPerformed(java.awt.event.ActionEvent evt) {
201         onClickSalvar();
202     }
203
204     private void btnExcluirActionPerformed(java.awt.event.ActionEvent evt) {
205         onClickExcluir();
206     }
207
208     private void btnLimparActionPerformed(java.awt.event.ActionEvent evt) {
209         onClickLimpar();
210     }
211
212     private void btnBuscarActionPerformed(java.awt.event.ActionEvent evt) {
213         onClickLocalizar();
214     }
215
216     private void btnBuscarTodosActionPerformed(java.awt.event.ActionEvent evt) {
217         onClickLocalizarTodos();
218     }
219
220     private void tblClientesMouseClicked(java.awt.event.MouseEvent evt) {
221         onMouseClickedTblClientes(evt);
222     }
223
```

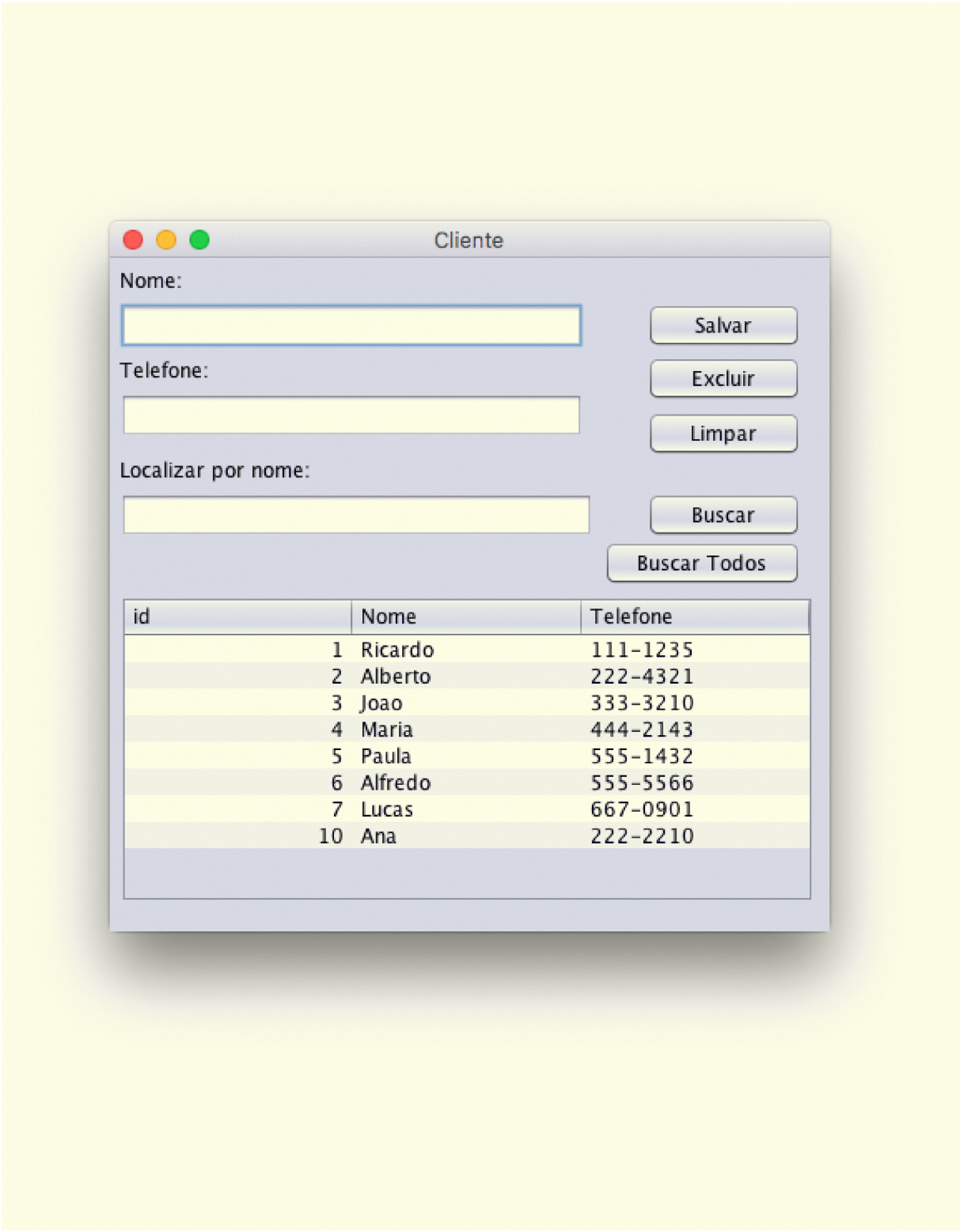
4FIGURA 23.26 - Classe ClienteFrame 1 FONTE: adaptada de Ballem (2011).

```
224     /**...3 lines */
227     public static void main(String args[]) {
228         /* Set the Nimbus look and feel */
229         Look and feel setting code (optional)
250
251         /* Create and display the form */
            java.awt.EventQueue.invokeLater(new Runnable() {
                public void run() {
254                 new ClienteFrame().setVisible(true);
255             }
256         });
257     }
258
259     // Variables declaration - do not modify
260     private javax.swing.JButton btnBuscar;
261     private javax.swing.JButton btnBuscarTodos;
262     private javax.swing.JButton btnExcluir;
263     private javax.swing.JButton btnLimpar;
264     private javax.swing.JButton btnSalvar;
265     private javax.swing.JLabel jLabel1;
266     private javax.swing.JLabel jLabel2;
267     private javax.swing.JLabel jLabel3;
268     private javax.swing.JScrollPane jScrollPane1;
269     private javax.swing.JTextField jTextField1;
270     private javax.swing.JTable tblClientes;
271     private javax.swing.JTextField txtLocalizar;
272     private javax.swing.JTextField txtNome;
273     private javax.swing.JTextField txtTelefone;
274     // End of variables declaration
275
276     private List<Cliente> clienteList = new ClienteController().listarClientes();
277     private int key;
278
279     private void onClickSalvar() {
280         ClienteController cc = new ClienteController();
281
282         try {
283             if (key == 0) {
284                 cc.salvar(txtNome.getText(), txtTelefone.getText());
285                 JOptionPane.showMessageDialog(this, "Contato salvo com sucesso!");
286             } else {
287                 cc.alterar(key, txtNome.getText(), txtTelefone.getText());
288                 JOptionPane.showMessageDialog(this, "Cliente atualizado com sucesso");
289             }
290
291             limparCampos();
292             key = 0;
293             localizarClientes();
294         } catch (SQLException ex) {
295             JOptionPane.showMessageDialog(this, "Ocorreu um erro, tente novamente!",
296                     "Erro", JOptionPane.ERROR_MESSAGE);
297         } catch (ParseException ex) {
298             JOptionPane.showMessageDialog(this, "Campos com dados incorretos",
299                     "Erro", JOptionPane.ERROR_MESSAGE);
300         }
301
302     }
```

4FIGURA 24.26 - Classe ClienteFrame 2 FONTE: adaptada de Ballem (2011).

```
303
304     private void onClickExcluir() {
305         ClienteController cc = new ClienteController();
306
307         try {
308             cc.excluir(key);
309             JOptionPane.showMessageDialog(this, "Cliente excluído com sucesso");
310             limparCampos();
311             key = 0;
312             localizarClientes();
313         } catch (SQLException ex) {
314             JOptionPane.showMessageDialog(this, "Não foi possível excluir o cliente",
315                     "Erro", JOptionPane.ERROR_MESSAGE);
316         }
317     }
318
319     private void onClickLimpar() {
320         limparCampos();
321         key = 0;
322     }
323
324     private void onClickLocalizar() {
325         ClienteController cc = new ClienteController();
326
327         try {
328             Cliente cliente = cc.buscaClientePorNome(txtLocalizar.getText());
329             txtNome.setText(cliente.getNome());
330             txtTelefone.setText(cliente.getTelefone());
331             key = cliente.getId();
333         } catch (SQLException ex) {
334             JOptionPane.showMessageDialog(this, "Cliente não localizado ou não existente",
335                     "Erro", JOptionPane.ERROR_MESSAGE);
336             limparCampos();
337         } catch (NullPointerException ex) {
338             JOptionPane.showMessageDialog(this, "Cliente não localizado ou não existente",
339                     "Erro", JOptionPane.ERROR_MESSAGE);
340             limparCampos();
341         }
342     }
343
344     private void limparCampos() {
345         txtNome.setText("");
346         txtTelefone.setText("");
347         txtLocalizar.setText("");
348     }
349
350     private void onClickLocalizarTodos() {
351         localizarClientes();
352     }
353
354     private void onMouseClickedTblClientes(java.awt.event.MouseEvent evt) {
355         if (evt.getClickCount() > 0) {
356             int row = tblClientes.getSelectedRow();
357             key = (int) tblClientes.getValueAt(row, 0);
358         }
359
360         ClienteController cc = new ClienteController();
361         try {
362             Cliente cliente = cc.buscaClientePorId(key);
363             txtNome.setText(cliente.getNome());
364             txtTelefone.setText(cliente.getTelefone());
365         } catch (SQLException ex) {
366             Logger.getLogger(ClienteFrame.class.getName()).log(Level.SEVERE, null, ex);
367         }
368     }
369
370     private void localizarClientes() {
371         ClienteController cc = new ClienteController();
372
373         DefaultTableModel tabelaModelo = (DefaultTableModel) tblClientes.getModel();
374
375         try {
376             while (tabelaModelo.getRowCount() > 0) {
377                 tabelaModelo.removeRow(0);
378             }
379
380             clienteList = cc.listarClientes();
382             for (Cliente c : clienteList) {
383                 tabelaModelo.addRow(new Object[] { c.getId(),
384                     c.getNome(), c.getTelefone()});
385             }
386
387         } catch (Exception ex) {
388             JOptionPane.showMessageDialog(this, "Deu Erro!", "Erro", JOptionPane.ERROR_MESSAGE);
389         }
390     }
}
```

4FIGURA 25.26 - Classe ClienteFrame 3 FONTE: adaptada de Ballem (2011).



4FIGURA 26.26 - Interface gráfica FONTE: adaptada de Ballem (2011).

Indicação de leitura

Nome do livro:: Padrões de Projeto em Java

Editora:: Bookman

Autor:: Steven John Metsker

ISBN:: 8536304111

Comentário: O livro aborda a utilização de diversos padrões de projeto utilizando Java. É interessante pois, em um projeto, podem surgir diversas situações que seriam solucionadas mais facilmente com a utilização de algum padrão de projeto. Como o livro é de 2004, algum código específico em Java pode estar desatualizado, porém o foco do livro, que são os padrões de projetos, tem aplicabilidade total com a utilização da versão atual do Java.

Conclusão

Este livro teve como objetivo introduzir as características da programação orientada a objetos, utilizando como base a linguagem Java, desde a evolução histórica das linguagens de programação, a instalação do ambiente de desenvolvimento Java pelo download do JDK, até a configuração dos caminhos necessários para a compilação e a execução correta de um projeto Java.

O ambiente de desenvolvimento utilizado foi o NetBeans IDE, que oferece um ambiente completo, tanto para desenvolvimento de aplicações Web quanto para aplicações desktop, provendo uma ferramenta de desenvolvimento de interfaces gráficas muito interessante.

Na programação orientada a objetos, foram abordados os conceitos-chave, como herança, que faz com que uma subclasse herde os atributos e os métodos de sua superclasse, a sobrecarga de métodos, possibilitando que um método de mesmo nome tenha formas diferentes de ser declarado, por meio de mudança da assinatura. Juntamente com a herança, outro conceito base da orientação a objetos foi introduzido, o polimorfismo, utilizando, também, sobreposição de métodos e implementação de interfaces, que definem o comportamento dos objetos, indicando quais métodos devem ser implementados.

No conteúdo voltado para interfaces gráficas (GUI), foi abordada a construção de interfaces utilizando os principais componentes do Swing, obedecendo ao princípio da orientação a objetos, no qual todos os componentes da interface gráfica são tratados como objetos, inclusive os eventos que adicionam funcionalidades à interface, pois uma interface gráfica sem tratamento de eventos não teria funcionalidade.

A última unidade abordou a utilização de padrões de projeto para implementar um projeto com menos dependência no acesso ao banco de dados, possibilitando futuras alterações no sistema, como troca do sistema gerenciador de banco de dados ou a alteração do ambiente de execução. Além dos conceitos, o uso de MVC e DAO foram exemplificados por meio de códigos, sugerindo a leitura de outros padrões de projeto que podem auxiliar a resolução de situações que podem ocorrer ao longo do desenvolvimento da melhor forma possível.

O desenvolvimento deste livro foi uma tarefa muito prazerosa, buscando passar o conteúdo referente ao Java e a orientação a objetos da forma mais clara possível, tendo em vista sempre utilizar exemplos práticos e situações que podem ocorrer no dia a dia do desenvolvimento de um sistema. Algumas boas práticas de programação e desenvolvimento foram citadas, principalmente o estudo de padrões de projetos para construir um sistema que gere a menor quantidade possível de retrabalho.

Muito obrigado pela oportunidade e espero que o conteúdo abordado seja útil para o seu futuro, caro(a) aluno(a)! Até a próxima!

Referências

ALVES, Luan. Sobrecarga e sobreposição de métodos em orientação a objetos. DEVMEDIA. <http://www.devmedia.com.br/sobrecarga-e-sobreposicao-de-metodos-em-orientacao-a-objetos/33066>

ANNALENI. Linha de habilidades de programa. 123RF. <https://br.123rf.com/photo_67657607_line-programming-skills-icons-circle.-illustration-of-coding-outline-objects-over-blurred-background.html?term=programming+objects&vti=ma3rhy7ctngjqphu1q>

BAKHTIAR ZEIN. Dados de banco de dados no armazenamento de rede multim. 123RF. <https://br.123rf.com/photo_46569990_dados-de-banco-de-dados-no-armazenamento-de-rede-multim.html?term=database&vti=nwx7bqdgnhpgli606z>

BAKHTIAR ZEIN. Programação Orientada a Objetos código OOP ilustração vetorial. 123RF. <https://br.123rf.com/photo_39638794_programação-orientada-a-objetos-código-oop-ilustração-vetorial.html?term=concept+java&vti=mg3vt482fto0qcttkf>

BALBO, Wellington. Conceitos e Exemplos – Polimorfismo: Programação Orientada a Objetos. DEVMEDIA. <http://www.devmedia.com.br/conceitos-e-exemplos-polimorfismo-programacao-orientada-a-objetos/18701>

BALLEM, Marcio. Utilizando Swing com Banco de Dados. MBallem. <http://www.mballem.com/post/utilizando-swing-com-banco-de-dados/>

BARNES, David J.; KÖLLING, Michael. Programação Orientada a Objetos com Java: Uma introdução prática utilizando o Blue J. 1. ed. São Paulo: Prentice-Hall, 2004.

BATES, Bert; SIERRA, Kathy. Use a cabeça! Java. 1. ed. Rio de Janeiro: Alta Books, 2005.

CAELUM. Classes Abstratas. Caelum. <https://www.caelum.com.br/apostila-java-orientacao-objetos/classes-abstratas/#mtodos-abstratos>

CAELUM 1. Interfaces gráficas em Java. Caelum. <http://www.caelum.com.br/apostila-java-testes-xml-design-patterns/interfaces-graficas-com-swing/#5-1-interfaces-graficas-em-java>

DANTAS, Rhawi. NetBeans IDE 7 Cookbook. Birmingham, UK: Packt Publishing, 2011.

DAVID, Hailton. Encapsulamento, Polimorfismo, Herança em Java. DEVMEDIA. <http://www.devmedia.com.br/encapsulamento-polimorfismo-heranca-em-java/12991>

DEITEL, Harvey M.; DEITEL, Paul J. Java: Como Programar. Trad. Edson Furmankiewicz. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

DEVMEDIA. Comparando as IDEs NetBeans e Eclipse. Revista Easy Java Magazine 22. <http://www.devmedia.com.br/comparando-as-ides-netbeans-e-eclipse-revista-easy-java-magazine-22/25767>>

FERREIRA, Alessandro. Conheça os Padrões de Projeto. DEVMEDIA. <http://www.devmedia.com.br/conheca-os-padroes-de-projeto/957>

FILGUEIRAS, Fellipe. <https://tableless.com.br/java-estruturas-de-repeticao/>

FREEMAN, Elisabeth; FREEMAN, Eric. Use a Cabeça! Padrão de Projetos. 2. ed. Rio de Janeiro: Alta Books, 2007.

GUDWIN, Ricardo. Linguagens de Programação. 1997. (ftp://ftp.dca.fee.unicamp.br/pub/docs/ea877/lingpro.pdf)

HORSTMANN, Cay. Big Java. Trad. Edson Furmankiewicz. Porto Alegre: Bookman, 2004.

JAVA. <https://www.java.com/pt_BR/download/faq/techinfo.xml>

JAVA 2. <http://docs.oracle.com/javase/8/docs/technotes/guides/install/windows_jdk_install.html#A1097272>

JUNGTHON, Gustavo; GOULART, Cristian M. Paradigmas de Programação. 2010. <https://fit.faccat.br/~guto/artigos/Artigo_Paradigmas_de_Programacao.pdf>

MACORATTI, José Carlos. Apresentando o padrão DAO - Data Access Object. Macoratti.net.

Microsoft Windows. <https://support.microsoft.com/pt-br>

NETBEANS. <https://netbeans.org/community/releases/80/install_pt_BR.html>

ORACLE. <https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html>

RICARTE, Ivan M. Programação Orientada a Objetos: Uma Abordagem com Java. Campinas: UNICAMP, 2007. <http://www.dca.fee.unicamp.br/cursos/PooJava/Aulas/poojava.pdf>

SOUZA, Vitor E. S. O Paradigma Orientado a Objetos. 2013. <http://www.inf.ufes.br/~vitorsouza/wp-content/uploads/academia-br-paradigmaoo.pdf>

STAFUSA, Victor. O que são os pacotes? Stack over flow. <http://pt.stackoverflow.com/questions/67112/o-que-são-os-pacotes>

STELTING, Stephem; MAASSEN, Olav. Applied Java Patterns. Palo Alto, USA: Prentice-Hall, 2002.

VINICIUS, Thiago. Trabalhando com Exceções em Java. DEVMEDIA. <http://www.devmedia.com.br/trabalhando-com-excecoes-em-java/27601>

VINICIUS 1, Thiago. Aprendendo Java com JDBC. DEVMEDIA. <http://www.devmedia.com.br/aprendendo-java-com-jdbc/29116>

WEXBRIDGE, Jason; NYLAND, Walter. NetBeans Platform for Beginners. Leanpub, 2014.]

Atividades

Atividades - Unidade I

Acerca das linguagens de programação, assinale a alternativa correta.

A) As primeiras linguagens eram simples e escritas em linguagem comum ao ser humano.

B) A linguagem Assembly é classificada como de baixo nível, demandando bastante trabalho para ser utilizada em aplicações.

C) COBOL é uma linguagem bastante conhecida, porém não tem mais utilização, pois era aplicada em dispositivos que não são mais usados.

D) Dentre as linguagens procedurais, destacam-se o Pascal e o C, porém ambos com princípios educacionais não utilizados comercialmente e ficam restritos apenas a características procedurais.

E) O Java é a linguagem orientada a objetos mais conhecida atualmente, porém não tem suporte para aplicações móveis e Web.

A programação utilizando uma linguagem orientada a objetos pode ser realizada diretamente em um editor de texto ou usar as funcionalidades de uma IDE de desenvolvimento. Acerca da IDE NetBeans, é correto afirmar que:

A) o NetBeans pode ser utilizado somente com a linguagem Java e não tem opções para implementar aplicações Web, somente desktop.

B) um projeto criado no NetBeans não pode ser utilizado como uma biblioteca para realizar tarefas de outro projeto.

C) o NetBeans tem integração com o repositório Maven, que analisa as dependências da aplicação e as mantêm atualizada.

D) a divisão de código-fonte não é importante no Java, com os arquivos podendo ser alocados em qualquer lugar, sem apoio algum da IDE.

E) o NetBeans não é multiplataforma e pode ser instalado apenas no Windows.

A orientação a objetos é uma forma de programação que surgiu recentemente (em relação à história da programação) e muda a visão de como o problema deve ser ambientado para implementar sua solução. Acerca da orientação a objetos, classes e métodos, qual a alternativa correta?

A) A orientação a objetos não tem diferença em relação ao paradigma estruturado, ambos têm processamento e saída, sem saber ao certo como é feita a relação entre o processamento e a saída.

B) A Engenharia de Software não é aplicada para auxiliar a planejar aplicações que serão desenvolvidas a partir da orientação a objetos.

C) As classes representam somente objetos inanimados do mundo real e que não possam realizar nenhuma tarefa.

D) Caso um objeto que seja representado por uma classe tenha que se comunicar com outra classe, essa comunicação é realizada pela chamada de seus métodos.

E) Um objeto representado em uma classe tem um número limitado de atributos que compõem suas características.

A respeito da sintaxe do Java, é correto afirmar que:

A) a utilização da palavra reservada new cria um novo objeto e, se esse objeto for da mesma classe de um objeto já existente, eles serão iguais.

B) o laço de repetição for pode realizar a iteração com valor decrescente para suas iterações.

C) o for é uma estrutura de repetição com intuito de realizar um bloco de comando diversas vezes. Durante as repetições, não há nenhuma chance de o fluxo de operações ficar executado infinitamente e sempre sairá do loop.

D) a vantagem da utilização do condicional switch é que diversos tipos de variáveis podem ser comparados, agilizando a programação e diminuindo o tamanho do código.

E) a comparação de dois objetos da classe Gato, gato1 == gato2, que foram somente instanciados pela palavra-chave new, tem como resultado verdadeiro.

Atividades - Unidade II

A respeito das classes em orientação a objeto, assinale a alternativa correta.

A) Construtor é o método responsável por inicializar o objeto no momento em que é criada uma nova referência a esse objeto.

B) No Java, não há uma maneira padrão para divisão dos pacotes e, muitas vezes, não é necessário utilizá-los.

C) Uma classe sempre deve ter um construtor declarado, caso contrário, não será criada corretamente no momento de criar uma nova instância.

D) Um método pode ser sobrecarregado de qualquer forma, trocando seus argumentos, trocando seu retorno ou trocando seu nome.

E) O encapsulamento não é uma boa prática de programação, pois devem ser escritos mais códigos ao invés de setar os valores diretamente nas variáveis de instância.

A herança é uma funcionalidade muito importante na Orientação a Objetos, facilitando a especificação do mundo real. Acerca da herança, assinale a alternativa correta.

A) Uma subclasse herda apenas as variáveis de instância, sendo necessário redefinir todos os métodos.

B) Uma classe só poderá ser estendida se ela satisfizer o teste É-UM, que indica que a subclasse complementa a superclasse.

C) O teste TEM-UM é similar ao teste É-UM e pode ser usado para representar herança.

D) As classes abstratas têm relação com a herança, pois uma classe abstrata pode ser herdada por outras classes e, quando é instanciada, pode invocar métodos da sua subclasse.

E) Uma classe abstrata não pode ter implementação em nenhum método, o que faz com que as classes que são herdadas dessa classe abstrata implementem esses métodos.

A respeito das características do Polimorfismo, assinale a alternativa correta.

A) O Polimorfismo é a característica da orientação a objetos que permite que um objeto estenda outro, fazendo com que suas características sejam compartilhadas.

B) Assim como na herança, uma classe que implementa uma interface pode implementar apenas uma interface.

C) Um objeto declarado do tipo X pode ser instanciado para a classe Y, desde que Y seja subclasse de X. Com isso, qualquer método definido em Y pode ser invocado.

D) A única diferença de uma classe abstrata para uma interface é a forma como é declarada, pois ambas não podem ter implementação.

E) Um método recebe como argumento um objeto do tipo X, e esse objeto X é uma superclasse de Y, portanto, é possível passar um objeto do tipo Y por parâmetro.

Exceções são erros que podem ocorrer nos programas por diversos motivos, muitas vezes, levando até a interrupção do programa. Acerca das exceções, assinale a alternativa correta.

A) A exceção ocorre somente por causa dos dados que o usuário insere no programa, não ocorrendo de outra forma.

B) As exceções são exclusivamente do método e da classe em que ocorreram, não sendo possível tratá-las em outro lugar.

C) Os blocos try/catch servem para capturar exceções e tratá-las, porém podem capturar apenas as exceções genéricas (classe Exception) e não podem ser utilizados em exceções específicas.

D) As exceções de tempo de execução (Runtime Exception) são classificadas como não verificadas e, caso não forem tratadas, podem ocasionar falha de execução do programa.

E) As informações geradas pelas exceções são somente as informações genéricas do Java, não sendo possível especificar os detalhes do erro.

Atividades - Unidade III

As interfaces gráficas são úteis para facilitar a interação entre usuário e sistema. A respeito das interfaces gráficas em orientação a objetos, assinale a alternativa correta.

A) A interface gráfica fornecida pelo Java não pode ser trocada de aparência (look-and-feel), utilizando as formas padrões da linguagem.

B) As caixas de diálogos são importante em uma GUI, pois enviam mensagens ao usuário, porém essas mensagens podem ser somente do tipo de Atenção e não podem ser customizadas pelo desenvolvedor.

C) Um componente JTextField utilizado para a inserção de números impede automaticamente que o usuário escreva caracteres não numéricos.

D) O componente JCheckBox é uma forma de botão que permite que apenas uma opção seja selecionada.

E) O Swing tem diversos componentes gráficos, desde componentes simples, como botões, até mais complexos, como listas, tabelas e containers que fazem o agrupamento e o gerenciamento desses componentes.

O tratamento de eventos é uma função importantíssima na programação de uma interface gráfica, pois é por meio dele que é realizada a interação entre o usuário e o sistema. Acerca dos tratamentos de eventos, assinale a alternativa correta.

A) Um evento ocorre somente quando há um clique do mouse, ativando o listener de eventos da interface.

B) Um evento de qualquer componente pode ser tratado como uma classe anônima. A classe anônima pode ser instanciada em qualquer lugar da classe, porém não precisa ter um nome definido

C) Os listeners são objetos que ficam "escutando" determinados componentes para verificar se um evento é disparado. Caso o evento seja disparado, o controle do evento é passado para o listener relacionado.

D) O Java pode tratar eventos que ocorrem nos componentes e, também, eventos que são disparados pelo teclado e pelo mouse. Contudo, o único evento de mouse possível é o clique.

E) Um listener de um determinado evento, que implementa a interface necessária, pode ser atribuído apenas ao componente no qual ele foi especificado.

O Swing tem gerenciadores de layout que organizam os componentes dentro de uma interface gráfica. Acerca dos gerenciadores de layout, assinale a alternativa correta.

A) Um gerenciador de layout pode trabalhar com qualquer componente do Swing, controlando sua disposição na janela, o tamanho do componente e, também, suas funcionalidades e label de descrição.

B) O Swing tem diversos gerenciadores de layout nativos, cada um se comportando de maneira distinta na disposição dos componentes, porém todos gerenciadores de layout devem ser atribuídos a um container que receberá os componentes. Um desses componentes pode ser um outro container que pode ter outro tipo de gerenciador de layouts.

C) O gerenciador BorderLayout distribui os componentes, primeiramente, pelas bordas, para, então, distribuir os componentes no centro, quantos componentes couberem no

container.

D) O GridLayout tem como característica distribuir os componentes como se estivessem em uma planilha, por meio de células, seguindo a seguinte ordem: primeiro, os componentes são adicionados nas colunas e, assim que as colunas estiverem cheias, passa-se para a coluna seguinte.

E) O FlowLayout é o gerenciador com mais detalhes de configurações, sendo necessário configurar como os componentes serão distribuídos, seu alinhamento e se os componentes serão adicionados em ordem alfabética.

AWT e Swing são pacotes com classes para implementar interfaces gráficas em Java. A respeito deles, assinale a alternativa correta.

A) AWT significa Abstract Window Toolkit e tem classe para implementar uma GUI em Java. A vantagem de utilizar o AWT é que ele já tem todas as funcionalidades pré-implementadas e é independente de sistema operacional.

B) O Swing é a ferramenta de desenvolvimento de GUI mais utilizada em Java, apresentando diversas funcionalidades, além de ser flexível e independente.

C) O AWT tem diversas configurações de look-and-feel, podendo ser customizado de acordo com o desenvolvedor da aplicação.

D) Como uma evolução do AWT, o Swing tem componentes mais leves, alta personalização e é totalmente diferente do AWT, não utilizando nenhum recurso já implementado pelo AWT.

E) Ao contrário do AWT, o Swing praticamente não tem código escrito, pois sua interface pode ser construída por meio do recurso de arrastar e soltar das IDEs.

Atividades - Unidade IV

SQL é a linguagem utilizada para realizar operações em um banco de dados. Acerca da SQL e de suas principais características, assinale a alternativa correta.

A) A linguagem SQL é utilizada para realizar consultas, inserções, atualizações e remoções de dados em uma tabela. Seu funcionamento é similar ao de uma linguagem orientada a objetos, na qual é necessário instanciar um objeto do tipo da operação desejada.

B) A instrução INSERT tem como objetivo inserir dados em uma tabela. Sua sintaxe pode ser escrita declarando quais colunas receberão os valores, ou quando todas as colunas da tabela receberão valores, não é necessário definir quais colunas, porém os dados devem ser inseridos na ordem em que as colunas foram definidas na tabela.

C) O comando SELECT realiza uma consulta em uma tabela do banco de dados, mas essa consulta pode ser realizada apenas em uma tabela por vez.

D) O JOIN é utilizado para exibir dados que estão em mais de uma tabela. Para sua utilização, deve-se apenas especificar quais campos são desejados e quais tabelas têm os dados que são desejados. Seu resultado é nada mais que todos os dados das duas tabelas.

E) O UPDATE altera os valores das colunas de uma linha de uma tabela, podendo até remover essa linha.

Sistema de Gerenciamento de Banco de Dados é um sistema que oferece ferramentas para administrar determinados bancos de dados. Acerca dos SGBDs assinale a alternativa correta.

A) No momento de criar as tabelas, é possível criar as colunas com tipo de campo e, também, definir condições para as colunas, como not null, primary key, dentre outras.

B) Os programas SGBD são apenas programas para gerenciamento, e não são possíveis instalar/executar juntamente com o banco de dados, apenas isolado.

C) Ao excluir um registro que possua uma chave estrangeira em outra tabela, o banco de dados exclui o registro da tabela principal e toda a tabela da chave estrangeira.

D) O SGBD do banco de dados MySql não permite fazer operações no banco de dados por meio de assistentes, somente pela linha de código.

E) Na criação de tabelas, é necessário criar uma coluna para ser a chave primária dessa tabela, marcando o flag PK (primary key), o que significa que essa coluna pode ter valores repetidos, mas são utilizados para busca de registros na tabela.

A utilização do JDBC em uma aplicação Java permite o acesso ao banco de dados. Assinale a alternativa correta a respeito do JDBC.

A) O JDBC é uma API que contém diversas classes e métodos para acessar banco de dados. O acesso ao banco de dados é feito por meio de um driver que é identificado automaticamente pelo Java.

B) O DriverManager é uma classe que realiza o registro do driver utilizado e retorna o arquivo de conexão com o banco de dados.

C) O objeto do tipo Connection, que é retornado pelo método createConnection da classe Driver, representa a conexão com o banco de dados e é ele que cria os objetos necessários para realizar as instruções SQL.

D) As instruções SQL (Select, Insert, Update e Delete) são executadas por meio de objetos do tipo Statement, que recebem uma string que contém a instrução SQL por parâmetro.

E) Ao executar métodos que contenham comandos SQL não é necessário envolvê-los com blocos try-catch ou utilizar um throw da exceção, pois o Java controla automaticamente as operações de banco de dados.

Em um projeto que se torna grande com o passar do tempo, não é recomendável realizar as operações de banco de dados juntamente com as interfaces gráficas, pois pode acarretar problemas na manutenibilidade do sistema. A respeito das formas de programação DAO e MVC, assinale a alternativa correta.

A) O DAO (Data Access Object) é um padrão de projeto que tem como objetivo modular a codificação do projeto, separando-o em três camadas: Modelo, Visão e Controle.

B) A utilização do padrão de projetos DAO e da arquitetura MVC em um projeto faz com que esse projeto tenha mais classes, porém fique mais organizado para futuras manutenções, diminuindo o retrabalho, caso seja necessário trocar de arquitetura de banco de dados ou de interface gráfica.

C) POJOs são as classes que fazem o acesso aos dados. Elas são utilizadas no DAO para realizar os comandos SQL e estão na camada Model do MVC.

D) A interação das camadas do MVC se dá pela camada View, que faz o relacionamento da camada Model com a camada Controller.

E) A vantagem de utilizar o padrão DAO é que uma única classe pode ser utilizada para fazer o acesso aos dados de todas tabelas.