

Programação Orientada a Objetos

Victorio Albani de Carvalho

Giovany Frossard Teixeira

Curso Técnico em Informática





·rede
e-Tec
Brasil

Programação Orientada a Objetos

Victorio Albani de Carvalho

Giovanly Frossard Teixeira



INSTITUTO FEDERAL
ESPÍRITO SANTO

Colatina - ES
2012

© Instituto Federal de Educação, Ciência e Tecnologia do Espírito Santo
Este Caderno foi elaborado em parceria entre o Instituto Federal de Educação,
Ciência e Tecnologia do Espírito Santo e a Universidade Federal de Santa Catarina
para a Rede – e-Tec Brasil.

Equipe de Elaboração

Instituto Federal de Educação, Ciência e Tecnologia do Espírito Santo – IFES

Coordenação Institucional

Guilherme Augusto de Moraes Pinto/IFES
João Henrique Caminhas Ferreira/IFES

Coordenação Curso

Allan Francisco Forzza Amaral/IFES

Professor-autor

Victorio Albani de Carvalho/IFES
Giovany Frossard Teixeira/IFES

Comissão de Acompanhamento e Validação

Universidade Federal de Santa Catarina – UFSC

Coordenação Institucional

Araci Hack Catapan/UFSC

Coordenação do Projeto

Sílvia Modesto Nassar/UFSC

Coordenação de Design Instrucional

Beatriz Helena Dal Molin/UNIOESTE e UFSC

Coordenação de Design Gráfico

André Rodrigues/UFSC

Design Instrucional

Gustavo Pereira Mateus/UFSC

Web Master

Rafaela Lunardi Comarella/UFSC

Web Design

Beatriz Wilges/UFSC
Mônica Nassar Machuca/UFSC

Diagramação

André Rodrigues da Silva/UFSC
Bárbara Zardo/UFSC
Juliana Tonietto/UFSC
Marília C. Hermoso/USFC
Nathalia Takeuchi/UFSC

Revisão

Júlio César Ramos/UFSC

Projeto Gráfico

e-Tec/MEC

C331p Carvalho, Victorio Albani de

Programação orientada a objetos : Curso técnico de informática / Victorio Albani de Carvalho, Giovany Frossard Teixeira. – Colatina: IFES, 2012. 134 p. : il.

**Inclui bibliografia
ISBN: 978-85-62934-38-4**

**1. Java (Linguagem de programação de computador).
2. Programação orientada a objetos (Computação).
3. Banco de dados. I. Carvalho, Victorio Albani de.
II. Teixeira, Giovany Frossard . III. Instituto Federal do Espírito Santo. IV. Título.**

CDD: 005.133

Apresentação e-Tec Brasil

Prezado estudante,

Bem-vindo ao e-Tec Brasil!

Você faz parte de uma rede nacional pública de ensino, a Escola Técnica Aberta do Brasil, instituída pelo Decreto nº 6.301, de 12 de dezembro 2007, com o objetivo de democratizar o acesso ao ensino técnico público, na modalidade a distância. O programa é resultado de uma parceria entre o Ministério da Educação, por meio das Secretarias de Educação a Distância (SEED) e de Educação Profissional e Tecnológica (SETEC), as universidades e escolas técnicas estaduais e federais.

A educação a distância no nosso país, de dimensões continentais e grande diversidade regional e cultural, longe de distanciar, aproxima as pessoas ao garantir acesso à educação de qualidade, e promover o fortalecimento da formação de jovens moradores de regiões distantes, geograficamente ou economicamente, dos grandes centros.

O e-Tec Brasil leva os cursos técnicos a locais distantes das instituições de ensino e para a periferia das grandes cidades, incentivando os jovens a concluir o ensino médio. Os cursos são ofertados pelas instituições públicas de ensino e o atendimento ao estudante é realizado em escolas-polo integrantes das redes públicas municipais e estaduais.

O Ministério da Educação, as instituições públicas de ensino técnico, seus servidores técnicos e professores acreditam que uma educação profissional qualificada – integradora do ensino médio e educação técnica, – é capaz de promover o cidadão com capacidades para produzir, mas também com autonomia diante das diferentes dimensões da realidade: cultural, social, familiar, esportiva, política e ética.

Nós acreditamos em você!

Desejamos sucesso na sua formação profissional!

Ministério da Educação
Janeiro de 2010

Nosso contato
etecbrasil@mec.gov.br

Indicação de ícones

Os ícones são elementos gráficos utilizados para ampliar as formas de linguagem e facilitar a organização e a leitura hipertextual.



Atenção: indica pontos de maior relevância no texto.



Saiba mais: oferece novas informações que enriquecem o assunto ou “curiosidades” e notícias recentes relacionadas ao tema estudado.



Glossário: indica a definição de um termo, palavra ou expressão utilizada no texto.



Mídias integradas: sempre que se desejar que os estudantes desenvolvam atividades empregando diferentes mídias: vídeos, filmes, jornais, ambiente AVEA e outras.



Atividades de aprendizagem: apresenta atividades em diferentes níveis de aprendizagem para que o estudante possa realizá-las e conferir o seu domínio do tema estudado.

Sumário

| | |
|--|-----------|
| Apresentação da disciplina | 13 |
| Projeto instrucional | 15 |
| Aula 1 – Introdução à plataforma Java | 17 |
| 1.1 Introdução..... | 17 |
| 1.2 A plataforma Java..... | 18 |
| 1.3 Ambientes de desenvolvimento Java..... | 19 |
| 1.4 Primeiro exemplo de programa em Java..... | 20 |
| 1.5 Variáveis e constantes..... | 22 |
| 1.6 Conversões entre tipos primitivos..... | 23 |
| 1.7 Comentários..... | 25 |
| 1.8 Atribuição, entrada e saída de dados..... | 26 |
| 1.9 Operadores..... | 27 |
| 1.10 Comandos de decisão ou seleção..... | 28 |
| 1.11 Comandos de repetição..... | 30 |
| 1.12 Escopo de variáveis..... | 31 |
| 1.13 Vetores e matrizes..... | 31 |
| Aula 2 – Introdução à Orientação a Objetos | 35 |
| 2.1 Conceitos básicos..... | 35 |
| 2.2 Classes em Java..... | 36 |
| 2.3 Declaração de atributos e métodos..... | 38 |
| 2.4 Utilização de objetos..... | 41 |
| 2.5 Atributos e métodos estáticos..... | 43 |
| 2.6 A classe <i>String</i> | 44 |
| 2.7 Listas..... | 45 |
| Aula 3 – Construtores, destrutores e encapsulamento | 49 |
| 3.1 Construtores..... | 49 |
| 3.2 Destrutores..... | 50 |
| 3.3 Encapsulamento..... | 51 |

| | |
|--|-----------|
| Aula 4 – Herança e polimorfismo | 55 |
| 4.1 Herança..... | 55 |
| 4.2 Utilização de atributos <i>protected</i> | 57 |
| 4.3 Polimorfismo..... | 58 |
| 4.4 Sobrescrita..... | 59 |
| 4.5 Sobrecarga..... | 62 |
| 4.6 Classe <i>Object</i> | 64 |
| Aula 5 – Classes abstratas e associações | 69 |
| 5.1 Classes abstratas..... | 69 |
| 5.2 Métodos abstratos..... | 72 |
| 5.3 Associações..... | 73 |
| Aula 6 – Herança múltipla e interfaces | 77 |
| 6.1 Herança múltipla..... | 77 |
| 6.2 Interfaces..... | 78 |
| Aula 7 – Interfaces gráficas em Java – Parte I | 83 |
| 7.1 Java Swing..... | 83 |
| 7.2 JFrame..... | 84 |
| 7.3 JLabel e ImageIcon..... | 86 |
| 7.4 <i>JOptionPane</i> | 88 |
| 7.5 Tratamento de eventos e <i>JButton</i> | 91 |
| 7.6 <i>JTextField</i> e <i>JPasswordField</i> | 93 |
| Aula 8 – Interfaces gráficas em Java – parte II | 99 |
| 8.1 Padrões de <i>layout</i> | 99 |
| 8.2 <i>JComboBox</i> e tratamento de eventos..... | 102 |
| 8.3 <i>JCheckBox</i> | 105 |
| 8.4 <i>JRadioButton</i> e <i>ButtonGroup</i> | 106 |
| 8.5 <i>JMenuBar</i> , <i>JMenu</i> e <i>JMenuItem</i> | 109 |

| | |
|---|------------|
| Aula 9 – Integração com Banco de Dados – parte I | 115 |
| 9.1 Programação cliente-servidor..... | 115 |
| 9.2 Acesso a Bancos de Dados em Java..... | 116 |
| 9.3 Fontes de dados ODBC..... | 117 |
| 9.4 Conectando com o Banco de Dados..... | 119 |
| 9.5 Consultando dados no banco..... | 122 |
| Aula 10 – Integração com Banco de Dados – parte II | 127 |
| 10.1 Introdução..... | 127 |
| 10.2 Inserção de dados..... | 129 |
| 10.3 Atualização de dados..... | 130 |
| 10.4 Exclusão de dados..... | 131 |
| Referências | 133 |
| Currículo dos professores-autores | 134 |

Palavra do professor-autor

Caro estudante!

Somos Victorio Albani de Carvalho e Giovany Frossard Teixeira, responsáveis pela elaboração deste material para a disciplina de Programação Orientada a Objetos.

Ao se deparar com mais uma disciplina voltada à programação, você pode se questionar sobre o motivo de termos mais uma disciplina com esse foco. De fato, nesta disciplina daremos continuidade aos assuntos das outras disciplinas, mas estudaremos a programação sob um novo paradigma: a orientação a objetos.

Nessa nova forma de programar, modelaremos o universo do nosso problema como um conjunto de objetos que têm determinadas características e agruparemos esses objetos em categorias chamadas classes. Então, em última instância, nossos programas serão um conjunto de classes.

Além disso, teremos outras duas novidades nos programas desenvolvidos nesta disciplina: interfaces gráficas com tratamento de eventos e acesso a banco de dados. Esse último assunto vem para fazer a ligação entre os programas que desenvolvemos nas disciplinas de programação e os conceitos estudados na disciplina de banco de dados.

Ou seja, será necessário relembrar diversos conceitos estudados em várias outras disciplinas e ainda estudar vários conceitos totalmente novos. Assim, trata-se de uma disciplina com conteúdo muito denso e que demanda muita dedicação.

Lembre-se: a melhor forma de aprender é praticando! Essa frase é ainda mais verdadeira para a programação. Assim, organize seu tempo, dedique-se ao curso e aproveite: você deverá divertir-se durante o processo de aprendizagem.

Um abraço!

Prof. Victorio Albani de Carvalho e Prof. Giovany Frossard Teixeira

Apresentação da disciplina

Nesta disciplina estudaremos os conceitos do paradigma de orientação a objetos utilizando a linguagem de programação Java para aplicar tais conceitos. Além disso, desenvolveremos programas com interface gráfica e com acesso a bancos de dados.

Ao programar seguindo o paradigma de orientação a objetos, usamos uma maneira de pensar diferente do que fazíamos nas disciplinas de programação anteriores, mas todos os conceitos aprendidos nas outras disciplinas serão utilizados nesta. Assim, é fundamental revisar conceitos como: variáveis, operadores, estruturas de laço e de decisão, vetores etc. A primeira aula desta disciplina visa revisar esses conceitos e apresentar a sintaxe de Java. De fato fazemos uma revisão geral da linguagem, pois consideramos que o aluno já utilizou a linguagem Java em alguma disciplina anterior sem se aprofundar nos conceitos de orientação a objetos ou vem estudando linguagem C, cuja sintaxe é muito parecida com a sintaxe de Java.

Após essa revisão inicial, focaremos nos conceitos de orientação a objetos e em sua aplicação na programação em Java. Em seguida focaremos na construção de programas com interface gráficas e no tratamento de eventos. Nesse contexto estudaremos as classes de Java voltadas à construção de interfaces gráficas. Por fim, aprenderemos a construir programas Java que acessem bancos de dados com capacidade para inserir, consultar, excluir e alterar dados. Assim, para essa última parte da disciplina serão fundamentais os conhecimentos assimilados na disciplina de Banco de Dados.

Este material se propõe a ser um guia didático dos conceitos a serem estudados. Entretanto, é impossível esgotar tantos assuntos em um único material. Assim, é importante ressaltar que estudos das bibliografias recomendadas podem ser necessários para um melhor aprendizado e desenvolvimento de capacitações complementares imprescindíveis à formação direcionada para este curso.

Bons estudos e sucesso!

Prof. Victorio Albani de Carvalho e Prof. Giovany Frossard Teixeira

Projeto instrucional

Disciplina: Programação Orientada a Objetos (carga horária: 90 hs).

Ementa: Parte 1: Paradigmas de programação orientada a objetos. Linguagem de programação orientada a objetos.

Parte 2: Programação de interfaces gráficas e tratamento de eventos. Programação cliente-servidor, compilação em separado. Bibliotecas dinâmicas (DLL). Integração com Banco de Dados.

| AULA | OBJETIVOS DE APRENDIZAGEM | MATERIAIS | CARGA HORÁRIA (horas) |
|---|--|---|-----------------------|
| 1. Introdução à plataforma Java | <p>Entender o funcionamento da plataforma Java.</p> <p>Entender o que são os ambientes de desenvolvimento Java.</p> <p>Conhecer ou revisar as principais características e recursos da linguagem Java: variáveis, comentários, operadores, comandos de decisão e de repetição, vetores e matrizes.</p> | <p><i>Softwares:</i> DK e NetBeans.</p> <p>Artigo sobre o que é Java em: http://javafree.uol.com.br/artigo/871498/Tutorial-Java-O-que-e-Java.html</p> <p>Artigo sobre as características básicas de Java em http://javafree.uol.com.br/artigo/871496/Tutorial-Java-2-Caracteristicas-Basicasindex</p> | 10 |
| 2. Introdução a orientação a objetos | <p>Conhecer os conceitos fundamentais de orientação a objetos: objetos, classes, métodos, atributos e pacotes. Aprender a criar e a utilizar classes e objetos em Java.</p> <p>Conhecer e aprender a utilizar a classe <i>String</i> e a classe <i>ArrayList</i>.</p> | <p><i>Softwares:</i> JDK e NetBeans.</p> <p>Documentação oficial da Oracle sobre a classe <i>ArrayList</i> em http://download.oracle.com/javase/6/docs/api/java/util/ArrayList.html</p> <p>Documentação oficial da Oracle sobre a classe <i>LinkedList</i> em http://download.oracle.com/javase/6/docs/api/java/util/LinkedList.html</p> | 10 |
| 3. Construtores, destrutores e encapsulamento | <p>Entender os conceitos de construtores e destrutores. Entender o conceito de encapsulamento e sua importância.</p> <p>Criar programas com maior manutenibilidade e extensibilidade pela utilização do conceito de encapsulamento.</p> | <p><i>Softwares:</i> JDK e NetBeans.</p> | 08 |
| (continua) | | | |

| AULA | OBJETIVOS DE APRENDIZAGEM | MATERIAIS | CARGA HORÁRIA (horas) |
|---|--|--|-----------------------|
| 4. Herança e polimorfismo | <p>Conhecer o conceito de herança e aprender a implementar esse conceito em Java. Compreender o conceito de polimorfismo.</p> <p>Conhecer os conceitos de sobrecarga e sobrescrita de métodos.</p> <p>Aprender a utilizar polimorfismo pela aplicação dos conceitos de herança, sobrecarga e sobrescrita de métodos.</p> | <p>Softwares: JDK e NetBeans.</p> <p>Documentação oficial da Oracle sobre a classe <i>Object</i> em http://download.oracle.com/javase/6/docs/api/java/lang/Object.html.</p> | 10 |
| 5. Classes abstratas e associações | <p>Conhecer os conceitos de classes e métodos abstratos.</p> <p>Aprender a utilizar classes abstratas em hierarquias de classes.</p> <p>Conhecer o conceito de associação entre classes.</p> <p>Aprender a implementar associações entre classes em Java.</p> | <p>Softwares: JDK e NetBeans.</p> | 08 |
| 6. Herança múltipla e interfaces | <p>Entender o conceito de herança múltipla.</p> <p>Conhecer o conceito de interface.</p> <p>Aprender a aplicar o conceito de interface em Java.</p> <p>Entender como o conceito de interface pode ser utilizado para simular uma herança múltipla em Java.</p> | <p>Softwares: JDK e NetBeans.</p> | 04 |
| 7. Interfaces gráficas em Java – parte I | <p>Construir as primeiras interfaces gráficas em Java.</p> <p>Conhecer algumas classes para construção de interfaces gráficas em Java: <i>JFrame</i>, <i>JLabel</i>, <i>ImageIcon</i>, <i>JOptionPane</i>, <i>JTextField</i> e <i>JPasswordField</i>.</p> <p>Aprender a fazer tratamento de eventos sobre interface gráfica em Java.</p> | <p>Softwares: JDK e NetBeans.</p> <p>Documentação oficial da Oracle sobre a classe <i>JTextArea</i> em http://download.oracle.com/javase/6/docs/api/javax.swing/JTextArea.html</p> | 10 |
| 8. Interfaces gráficas em Java – parte II | <p>Aprender o conceito de padrões de <i>layout</i>.</p> <p>Conhecer alguns padrões de <i>layout</i> de Java.</p> <p>Conhecer mais classes para construção de interfaces gráficas em Java: <i>JComboBox</i>, <i>JCheckBox</i>, <i>JRadioButton</i>, <i>ButtonGroup</i>, <i>JMenuBar</i>, <i>JMenu</i> e <i>JMenuItem</i>.</p> <p>Desenvolver novos exemplos de tratamento de eventos sobre interface gráfica em Java.</p> | <p>Softwares: JDK e NetBeans.</p> | 10 |
| 9. Integração com Bancos de Dados – parte I | <p>Compreender o conceito de Programação cliente-servidor.</p> <p>Aprender a acessar bancos de dados em Java utilizando <i>drivers</i> nativos e ODBC.</p> <p>Aprender a construir aplicações em Java capazes de consultar dados de bancos de dados.</p> | <p>Softwares: JDK, NetBeans, MySQL Server e <i>driver</i> nativo do MySQL.</p> | 10 |
| 10. Integração com Bancos de Dados – parte II | <p>Aprender a construir aplicações em Java capazes de inserir, alterar e excluir dados de bancos de dados.</p> | <p>Softwares: JDK, NetBeans, MySQL Server e <i>driver</i> nativo do MySQL.</p> | 10 |

(conclusão)

Aula 1 – Introdução à plataforma Java

Objetivos

Entender o funcionamento da plataforma Java.

Entender o que são os ambientes de desenvolvimento Java.

Conhecer ou revisar as principais características e recursos da linguagem Java: variáveis, comentários, operadores, comandos de decisão e de repetição, vetores e matrizes.

1.1 Introdução

Nesta disciplina estudaremos os conceitos de programação orientada a objetos. Para isso, precisaremos aprender a programar em uma linguagem de programação que siga esses conceitos: a linguagem Java.

A linguagem Java apresenta uma sintaxe muito semelhante às linguagens C e C++.

Nesta disciplina consideramos que o aluno já utilizou a linguagem Java no semestre anterior sem se aprofundar nos conceitos de orientação a objetos, ou o aluno vem estudando linguagem C nos períodos anteriores, já tendo assim familiaridade com a sintaxe dessa linguagem. Nesse contexto, essa primeira aula traz uma visão geral da plataforma e da linguagem Java apenas para solidificar conhecimentos prévios do aluno.



Dentre as características da linguagem Java destacam-se:

- Orientação a objetos: suporte ao paradigma de programação orientada a objetos.
- Portabilidade: é possível rodar um *software* feito em Java em qualquer máquina que disponha de máquina virtual implementada para ela.

- *Multithreading*: possibilidade de desenvolvimento utilizando *threads*.
- Suporte à programação para internet: Java foi concebida originalmente para ser usada no ambiente da *World Wide Web*, diferentemente de outras linguagens que foram adaptadas para o desenvolvimento *web*.
- Suporte à comunicação: classes para programação em rede.
- Acesso remoto a banco de dados – dados recuperados e/ou armazenados de qualquer ponto da internet.
- Segurança: mecanismos de segurança que a linguagem oferece para realização de processos pela internet.
- Sintaxe baseada na sintaxe da linguagem C.

1.2 A plataforma Java

Plataformas podem ser descritas como a combinação do sistema operacional e o *hardware* em que rodam. Nesse contexto, a maioria das plataformas de desenvolvimento existentes possui uma restrição marcante: cada programa é produzido para uma plataforma (Sistema Operacional + *hardware*) específica. A *plataforma* Java difere dessas plataformas pelo fato de desagregar o *hardware* de si, ou seja, trata-se de uma **plataforma** de *software* que roda em cima de outras plataformas baseadas em *hardware*.

Essa independência de *hardware* obtida pela plataforma Java deve-se à utilização do conceito de máquina virtual: a *Java Virtual Machine* (JVM). A JVM é um *software* que funciona sobre o sistema operacional, sendo responsável pelo processo de tradução de um programa Java para uma plataforma específica. Assim, um programa feito em Java pode rodar em qualquer SO de qualquer arquitetura, desde que exista uma JVM implementada para ele. A Figura 1.1 ilustra o processo de execução de um aplicativo Java.

A-Z

Plataforma

É o ambiente de *software* ou *hardware* no qual um programa é executado.

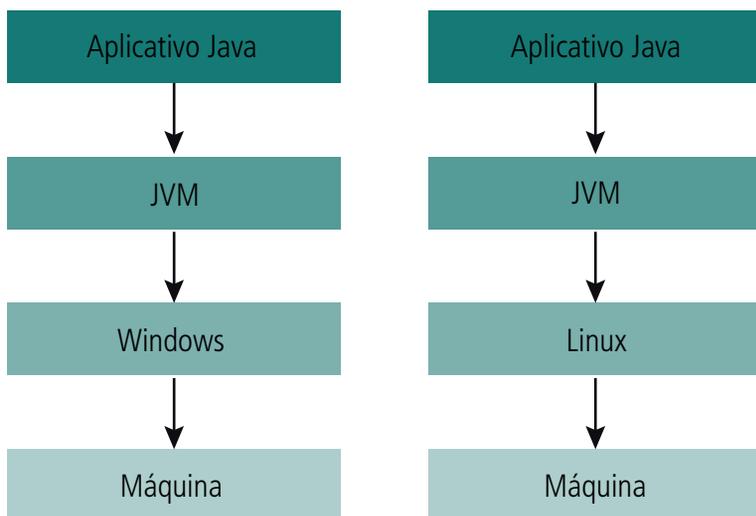


Figura 1.1: Execução de um aplicativo Java

Fonte: Elaborada pelos autores

1.3 Ambientes de desenvolvimento Java

Um programa Java precisa passar por um processo de compilação para ser analisada a existência de erros de sintaxe. Esse processo de compilação traduz o código-fonte escrito pelo programador para uma linguagem intermediária chamada *Java bytecodes*. Esse processo de tradução dos códigos fontes para *Java bytecodes* é feito por um programa chamado **compilador**. Então, é necessário que outra ferramenta chamada **interpretador** se responsabilize por interpretar esses *bytecodes* para o sistema operacional. Essa ferramenta que interpreta *bytecodes* é a máquina virtual Java (JVM).

O conjunto de ferramentas necessárias para desenvolver, compilar e rodar aplicativos Java é disponibilizado em um *kit* conhecido como *Java Development Kit* (JDK). Assim, para começar a programar em Java você deve realizar o *download* do JDK e instalá-lo.

Ao realizar o *download* do JDK, escolha a versão correta para seu sistema operacional.

Com o JDK instalado, você pode começar a programar em Java utilizando um simples editor de texto para editar seus programas, como, por exemplo, o bloco de notas. Assim, você teria de editar seus programas, salvá-los com extensão *.Java*, compilá-los e então executá-los.



A plataforma Java consiste em três partes principais: a linguagem de programação Java, a máquina virtual Java (JVM) e diversas APIs Java. O projeto da plataforma Java é controlado pela *Sun* e por sua comunidade de usuários.





Acesse o site http://www.netbeans.org/index_pt_BR.html e baixe a versão mais recente do NetBeans (a versão utilizada neste material é a 7.0). Note que você pode baixar diretamente o JDK com o NetBeans IDE Java SE já embutido. Nesse caso, não será necessário executar o *download* e instalação do JDK. Baixe a versão mais recente do NetBeans com JDK embutido e instale em seu computador. A instalação é muito simples! Basta seguir as mensagens informadas nas janelas exibidas pelo assistente de instalação.

Para facilitar e agilizar esse processo, existem disponíveis vários Ambientes de Desenvolvimento – *Integrated Development Environment* (IDE), que dão suporte à linguagem Java. Um IDE é um programa de computador que reúne ferramentas de apoio ao desenvolvimento de *software* com o objetivo principal de agilizar o processo de codificação.

Há vários IDEs para programação Java. Os dois mais amplamente utilizados são o NetBeans e o Eclipse. Nessa disciplina utilizaremos o NetBeans.

O NetBeans IDE é um ambiente de desenvolvimento integrado gratuito e de código aberto. Esse IDE é executado em muitas plataformas, como Windows, Linux, Solaris e MacOS, sendo fácil de instalar e usar. O NetBeans IDE oferece aos desenvolvedores todas as ferramentas necessárias para criar aplicativos profissionais de *desktop*, empresariais, *web* e móveis multiplataformas. Assim, todo o processo de edição, compilação e execução dos programas será feito dentro do NetBeans.

1.4 Primeiro exemplo de programa em Java

Nesta aula ainda não estudaremos os conceitos de orientação a objeto, que é uma das principais características da linguagem Java. Por enquanto, vamos nos focar na criação de programas em Java semelhantes aos que criávamos em linguagem C.

Para ilustrar as semelhanças entre a linguagem C e Java, utilizaremos um exemplo simples de programa que apenas imprime na tela a mensagem “*Primeiro Exemplo!*”. A Figura 1.2 exibe no lado esquerdo o código do programa em linguagem C e, no lado direito, o código equivalente em Java.

| Exemplo em JAVA | Exemplo em C |
|--|--|
| <pre>public class Exemplo01 { public static void main(String[] args) { System.out.println("Primeiro Exemplo!"); } }</pre> | <pre>int main(){ printf ("Primeiro Exemplo!\n"); }</pre> |

Figura 1.2: Programa simples em Java e em C

Fonte: Elaborada pelos autores



Assim como a linguagem C, Java é *case sensitive*, ou seja, o compilador diferencia letras minúsculas de maiúsculas. Logo, tenha muita atenção ao digitar!

A principal diferença que podemos notar já de início é que todo programa em Java inicia-se com a definição de uma classe. Uma classe é definida pela palavra reservada **class**, seguida pelo nome da classe (neste caso, o nome da classe é "Exemplo01"). Por convenção, todo nome de classe inicia-se com uma letra maiúscula. Um programa Java é formado por uma ou mais classes. Não se preocupe agora com a definição de classe. Estudaremos esse conceito mais à frente.

Assim como em C todo programa Java tem sua execução iniciada pelo método **main**, alguma das classes do programa Java deverá conter um método **main**. Essa classe é chamada de classe principal.

As palavras "public" e "static" que aparecem na definição do método **main** serão explicadas em aulas futuras. Já a palavra "void", assim como em C, indica que o método não possui retorno. O argumento "String args[]" é um vetor de Strings formado por todos os argumentos passados ao programa na linha de comando quando o programa é invocado. O comando "System.out.println ()" é utilizado para imprimir algo na tela. Ele é equivalente ao "printf()" da linguagem C.

Você poderia editar esse programa no bloco de notas e salvá-lo com o nome "Exemplo01.java" e, em uma máquina com o Java Development Kit (JDK) instalado, compilá-lo digitando no *prompt* comando: "javac Exemplo01.java". Esse comando geraria o arquivo *bytecode* com o nome "Exemplo01.class". Então, para executar o programa, deveria ser digitado no *prompt* comando: "java Exemplo01.class".

Em Java, o nome do arquivo deve ser sempre igual ao nome da classe, seguida da extensão ".java". No caso do nosso exemplo, o nome do arquivo precisa ser "Exemplo01.java" porque o nome da classe é "Exemplo01".



Mas, vamos utilizar o NetBeans, pois nos poupa bastante trabalho e seu editor oferece alguns recursos que facilitam a edição do código. O primeiro passo é abrirmos o NetBeans e criarmos um novo projeto. Para isso, realize os passos a seguir:

1. Clique em "**Arquivo**" > "**Novo Projeto**".
2. Dentre as opções "Categorias" da janela aberta, escolha "**Java**". Dentro de "Projetos", escolha "**Aplicativo Java**". Clique em "**Próximo**".

3. Escolha um nome para o projeto (por exemplo, “**Exemplos**”) e deixe marcadas as opções “Criar classe principal” (defina o nome da classe principal como “**Exemplo01**”) e “Definir como projeto principal”. Clique em “**Finalizar**”.

Após criarmos o projeto, ele passa a ser exibido na aba “Projetos”, que fica no canto esquerdo do NetBeans. Na área de edição principal será exibida a classe “Exemplo01” criada. Ela já irá possuir um método “**main**”. Nesse caso, apenas insira no método o comando “**System.out.println (“Primeiro exemplo!”)**”.

Para compilar e executar o programa basta acessar o menu “**Executar**” > “**Executar Projeto Principal**” ou pressione a tecla “F6” ou, ainda, clique no botão “Executar” da barra de ferramentas (um triângulo verde). Então, a saída do programa será exibida na aba “Saída”, localizada abaixo da área de exibição de código do NetBeans. A Figura 1.3 exibe a tela do NetBeans destacando a aba de saída e o botão “Executar”.

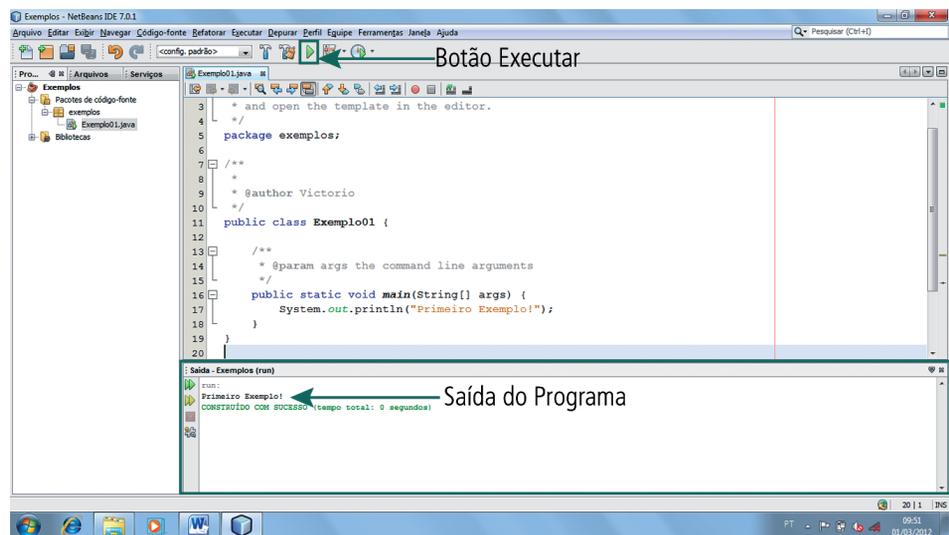


Figura 1.3: Resultado da execução do exemplo da Figura 1.2

Fonte: NetBeans IDE 7.0.1

A-Z

Variável

É um lugar onde as informações podem ser armazenadas enquanto um programa está sendo executado. O valor pode ser alterado em qualquer ponto do programa – daí o nome (CADENHEAD; LEMAY, 2005, p. 22).

1.5 Variáveis e constantes

Assim como em linguagem C, programas feitos em Java devem ter suas variáveis declaradas e inicializadas antes de serem utilizados. Em Java toda **variável** é identificada por um nome e tem um tipo definido no ato da declaração, que não pode ser alterado.

A sintaxe para declaração de variáveis em Java é idêntica à que utilizamos em C.

Sintaxe: <tipo_da_variável> <nome_da_variável>;

Exemplos: `double salario_liquido, salario_bruto;`

O tipo da variável define, além do tipo de dado que ela pode armazenar, o tamanho do espaço de memória que deve ser alocado para ela. O Quadro 1.1 apresenta os tipos primitivos da linguagem Java. São os mesmos tipos de C, com o acréscimo do tipo *boolean*.

Quadro 1.1: Tipos primitivos da linguagem Java

| Tipo primitivo | Valores a serem armazenados | Tamanho em bits |
|----------------|---|-----------------|
| <i>char</i> | Permite armazenar um caractere. Exemplo: 'a'. | 16 |
| <i>byte</i> | Permite armazenar um número inteiro de -128 até +127. Exemplo: 10. | 8 |
| <i>short</i> | Permite armazenar um número inteiro de -32.768 até +32.767. Exemplo: 10. | 16 |
| <i>int</i> | Permite armazenar um número inteiro de -2.147.483.648 até +2.147.483.647. Exemplo: 10. | 32 |
| <i>long</i> | Permite armazenar um número inteiro de -9.223.372.036.854.775.808 até +9.223.372.036.854.775.807. Exemplo: 10. | 64 |
| <i>float</i> | Permite armazenar um ponto flutuante de -3,40292347E+38 até +3,40292347E+38. Exemplo: 3.1416. | 32 |
| <i>double</i> | Permite armazenar um ponto flutuante de -1,79769313486231570E+308 até +1,79769313486231570E+308. Exemplo: 3.1416. | 64 |
| <i>boolean</i> | Permite armazenar apenas o valor <i>true</i> ou o valor <i>false</i> . | 8 |

Fonte: Elaborado pelos autores



Ponto flutuante

É um formato de representação de números reais em computadores.

Em Java não há a definição de constantes. Quando queremos definir variáveis com valores constantes em Java utilizamos a palavra reservada **final**.

Sintaxe: **final** <tipo_da_variável> <nome_da_variável> = <valor_constante>;

Exemplo: `final float PI = 3.1416;`

1.6 Conversões entre tipos primitivos

A conversão de tipos consiste em utilizar uma variável ou valor de um tipo para gerar um novo valor de outro tipo. Em alguns casos essa conversão é feita de forma direta, como, por exemplo, quando atribuímos um valor inteiro a uma variável do tipo *double*. Nesse caso a conversão é direta (também chamada de conversão implícita) porque é óbvio que um número inteiro pode ser representado como um número real.

Mas, alguns tipos de variáveis apresentam valores incompatíveis entre si. Assim, nesses casos não é possível fazer uma atribuição direta. Por exemplo, se tentarmos atribuir a uma variável inteira o valor de uma variável double, teremos um erro de compilação; veja a Figura 1.4.

```
double p = 23.3;
int i;
i=p; //não compila
i=2.3; //também não compila
```

Figura 1.4: Atribuição entre valores incompatíveis

Fonte: Elaborada pelos autores

Para que atribuições como esta sejam possíveis, precisamos solicitar explicitamente que o número real seja moldado (*casted*) como um número inteiro. Essa operação de conversão entre tipos é também chamada de *casting*. Para realizar a conversão explícita, coloca-se, antes da expressão a ser convertida, o tipo destino da transformação entre parênteses (Figura 1.5).

```
double p = 23.3;
int i;
i = (int)p; //agora compila
i = (int)2.3; //agora compila
```

Figura 1.5: Conversão correta

Fonte: Elaborada pelos autores

No primeiro caso do exemplo apresentado na Figura 1.5, o valor da variável *p* foi convertido para caber em uma variável inteira. Assim, a variável *i* assumiu o valor 23. Logo depois, a variável *i* assumiu o valor 2.

Ao fazer o **casting** para forçar a atribuição de um valor a uma variável, deve-se prestar atenção aos limites de valores de cada tipo primitivo. Esses limites foram apresentados no Quadro 1.1. Por exemplo, se atribuímos o valor 200 a uma variável inteira e depois utilizamos um **casting** para atribuir esse valor a uma variável *byte* cujos possíveis valores são de -128 a 127 não teremos erro de compilação, mas, como 200 estoura o valor máximo para uma variável *byte*, ocorrerá uma falha conhecida como *overflow* e o valor armazenado na variável *byte* será -56 (Figura 1.6).

```
int i = 200;
byte c = (byte) i; //Compila normalmente
System.out.println("c = " + c); // O valor impresso será -56
```

Figura 1.6: Conversão com overflow

Fonte: Elaborada pelos autores

As conversões entre tipos primitivos podem ainda ser necessárias em expressões aritméticas quando são utilizados operandos com tipos divergentes (Figura 1.7).

```

long l = 12;
int i1 = 15;
int i2;
❌ i2 = i1 + l; //Não compila pois, o resultado da soma será um long
i2 = i1 + (int)l; //Agora compila
byte b1, b2 = 10;
❌ b2 = b1 + (byte)i1; //Mesmo com casting não compila!!!!

```

Figura 1.7: Conversões necessárias em expressões aritméticas

Fonte: Elaborada pelos autores

Note que no exemplo da Figura 1.7 foi necessário fazer a conversão da variável *l* que é do tipo *long* para o tipo *int*, pois quando é realizada uma operação entre operadores de tipos distintos o Java faz as contas utilizando o maior tipo que aparece nas operações, no caso, *long*. No entanto, na última linha do exemplo, mesmo fazendo a conversão da variável *i1* que é do tipo *int* para o tipo *byte* continuou não compilando. Isso aconteceu porque o Java armazena o resultado de expressões em, no mínimo, um *int*. Assim, para que o trecho de código apresentado pela Figura 1.7 seja compilado sem erros, as duas linhas marcadas devem ser retiradas ou comentadas.

O Quadro 1.2 exhibe alguns exemplos de conversões entre tipos.

| Quadro 1.2: Conversão entre tipos | | |
|-----------------------------------|-----------------|---|
| Tipo de Origem | Tipo de Destino | Exemplo |
| <i>int</i> | <i>double</i> | <i>int</i> a = 20; <i>double</i> b = a; //nesse caso a conversão é implícita |
| <i>double</i> | <i>int</i> | <i>double</i> a = 20.1; <i>int</i> b = (int) a; //conversão explícita |
| <i>double</i> | <i>float</i> | <i>double</i> a = 20.1; <i>float</i> b = (float) a; //conversão explícita |
| <i>long</i> | <i>int</i> | <i>long</i> a = 20; <i>int</i> b = (long) a; //conversão explícita |

Fonte: Elaborado pelos autores

Valores do tipo *boolean* não podem ser convertidos para nenhum outro tipo.



1.7 Comentários

Comentários são indicações que colocamos no código para facilitar que outros programadores o entendam. Java aceita três tipos de comentários:

- Quando queremos fazer um comentário de uma única linha, basta utilizar *//* para iniciar o comentário (assim como em linguagem C).
- Quando queremos fazer comentários de várias linhas, iniciamos o comentário com */** e finalizamos com **/* (assim como em linguagem C).



Javadoc é um programa que gera automaticamente documentação em HTML de programas Java a partir de comentários que seguem um determinado formato. O Javadoc é fornecido junto com o *kit* de desenvolvimento Java.

- Quando queremos fazer comentários de várias linhas e gerar documentação Javadoc, iniciamos o comentário com `/**` e finalizamos com `*/`.

```
package pacote1;
public class ClassePrincipal {
    /**
     * Comentário estilo Javadoc
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Comentário de uma linha
        final double peso = 82.5;
        int idade = 25;
        /* Comentários de Várias linhas
         * Imprimindo a mensagem
         */
        System.out.println("O peso vale: " + peso + " e a idade vale " + idade);
    }
}
```

Figura 1.8: Exemplo de utilizações de comentários e de declaração de variáveis e constantes.

Fonte: Elaborada pelos autores

1.8 Atribuição, entrada e saída de dados

Em Java, quando queremos atribuir um determinado valor a uma variável, utilizamos o operador "=" exatamente como fazemos na linguagem C.

Para obter entradas de dados digitados pelo usuário, em linguagem C nós utilizamos o comando `scanf`. Em Java utilizaremos a classe `Scanner`. O uso dessa classe é bastante simples e poderemos entender melhor no exemplo apresentado pela Figura 1.9 a seguir.

```
package pacote1;
import java.util.Scanner;
public class ExemploUsandoClasseScanner {
    public static void main(String[] args) {
        //File fil = new File("c:/arq.txt");
        Scanner scan = new Scanner(System.in);
        int num2 = scan.nextInt();
        int num1 = scan.nextInt();
        double num3 = scan.nextDouble();
        System.out.print(num2+num1);
        System.out.println(num3);
        scan.close();
    }
}
```

Figura 1.9: Utilização da classe `Scanner` para entrada de dados

Fonte: Elaborada pelos autores

Ainda não aprendemos o conceito de classe, mas utilizaremos algumas classes básicas, como a classe *Scanner*.

O primeiro passo é criarmos um objeto do tipo *Scanner*, conforme a linha: `Scanner scan = new Scanner(System.in);`

Para lermos valores do tipo *int* utilizando o objeto *scan*, utilizamos o método *nextInt()*. Já para valores do tipo *double*, utilizamos o método *nextDouble()*.

Não podemos esquecer de fechar o objeto quando findamos seu uso pelo método *close()*.

Para exibir mensagens e dados na tela, utilizamos em linguagem C o comando *printf*. Em Java utilizaremos os “métodos” *System.out.print* e *System.out.println*. A única diferença entre eles é que o primeiro imprime a mensagem na mesma linha em que a mensagem anterior foi impressa, enquanto o segundo imprime a mensagem em uma nova linha. Note que os dois foram utilizados no exemplo da Figura 1.9.

Note que os comandos *System.out.print* e *System.out.println* não utilizam os códigos de formatação (%c, %d, %f...) que nos acostumamos a utilizar com a função *printf* da linguagem C. Em Java basta indicar os dados que se deseja imprimir na tela e o compilador se encarrega de fazer a transformação desses dados para *String*.

1.9 Operadores

Os operadores aritméticos são símbolos que representam operações aritméticas, ou seja, as operações matemáticas básicas. Os operadores aritméticos da linguagem Java são os mesmos da linguagem C, como pode ser visto no Quadro 1.3 a seguir.

Quadro 1.3: Operadores aritméticos da linguagem Java

| Operador | Descrição da operação matemática |
|----------|---|
| + | Soma (inteira e ponto flutuante) |
| - | Subtração (inteira e ponto flutuante) |
| * | Multiplicação (inteira e ponto flutuante) |
| / | Divisão (inteira e ponto flutuante) |
| -- | Decremento unário (inteiro e ponto flutuante) |
| ++ | Incremento unário (inteiro e ponto flutuante) |
| % | Resto da divisão de inteiros |

Fonte: Elaborado pelos autores



Em Java, podemos utilizar bibliotecas definidas em pacotes da linguagem. Para isso, utilizamos o comando *import <caminho do pacote>.<classe>*;

O comando *import* de Java é parecido com o *include* que utilizamos em linguagem C. Veja na segunda linha do exemplo apresentado na Figura 1.9 que a classe *Scanner* foi importada para o nosso programa para que pudéssemos utilizá-la.

Quando o NetBeans identifica que uma classe foi utilizada e não foi feita sua importação, ele sugere a sua inclusão, facilitando o trabalho do programador.



No exemplo da Figura 1.9, existe uma linha comentada que basicamente seria a criação de uma variável do tipo *File* (arquivo). Para que o nosso exemplo lesse de arquivo em vez do teclado, descomentariamos essa linha, na linha seguinte, trocariamos *System.in* por *fil* e pronto. Agora os dados viriam do arquivo *C:/arquivo.txt*.

Quando a linha comentada for descomentada, será necessário acrescentar a linha; no código-fonte ou permitir que o NetBeans o faça.

Os operadores relacionais são utilizados para realizar comparações entre dois valores de um mesmo tipo, retornando como resultado sempre um valor lógico, ou seja, verdadeiro ou falso. A exemplo do que ocorre com os aritméticos, os operadores relacionais de Java também são os mesmos que aprendemos em linguagem C, como exibido no Quadro 1.4 a seguir.

Quadro 1.4: Operadores relacionais de Java

| Descrição | Operador |
|------------------|---------------------------|
| igual a | == (dois sinais de igual) |
| maior que | > |
| menor que | < |
| maior ou igual a | >= |
| menor ou igual a | <= |
| diferente de | != |

Fonte: Elaborado pelos autores

Os operadores lógicos são utilizados para formar novas proposições lógicas a partir da junção de duas outras. Os operadores lógicos de Java também foram obtidos da linguagem C, como mostra o Quadro 1.5 a seguir.

Quadro 1.5: Operadores lógicos de Java

| Descrição | Operador |
|-----------|-------------------------|
| E | && |
| OU | (duas barras verticais) |
| NÃO | ! (exclamação) |

Fonte: Elaborado pelos autores



Revise a teoria sobre operadores lógicos, operadores relacionais e tabelas-verdade, que foi estudada na disciplina de Lógica de Programação e aplicada em todas as disciplinas de programação do curso; por isso, não será repetida aqui.

1.10 Comandos de decisão ou seleção

O Java fornece três tipos de instruções de seleção. A instrução *if* realiza uma ação se uma condição for verdadeira ou pula a ação se ela for falsa. A instrução *if...else* realiza uma ação se uma condição for verdadeira ou realiza uma ação diferente se a condição for falsa. A instrução *switch* realiza uma de muitas ações, dependendo do valor de uma expressão (DEITEL; DEITEL, 2010, p. 84).



Os comandos de decisão existentes em Java são os mesmos aprendidos na linguagem C, até mesmo a sintaxe.

O exemplo apresentado na Figura 1.10 ilustra a utilização do comando *if...else*. Nesse exemplo, lemos uma nota e verificamos se ela está entre 0 e 100: se estiver, a nota é considerada válida; caso contrário, inválida.

```
package pacote1;
import java.util.Scanner;
public class ExemploIf {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Entre com a nota: ");
        double nota = scan.nextDouble();
        if (nota <= 100 && nota >= 0) {
            System.out.println("Nota = " + nota + " ( valor válido );");
        } else {
            System.out.println("Nota = " + nota + " ( valor inválido );");
        }
    }
}
```

Figura 1.10: Exemplo de utilização do comando de decisão *if...else*

Fonte: Elaborada pelos autores

No exemplo apresentado na Figura 1.11, faremos uso do comando *switch-case*. Nesse exemplo, dado um número de mês, temos a impressão do nome do mês no *prompt* de comando.

```
package pacote1;
public class ExemploSwitchCase {
    public static void main(String[] args) {
        int mes;
        mes = Integer.parseInt(args[0]);
        switch (mes) {
            case 1: System.out.println("Janeiro"); break;
            case 2: System.out.println("Fevereiro"); break;
            case 3: System.out.println("Março"); break;
            case 4: System.out.println("Abril"); break;
            case 5: System.out.println("Maio"); break;
            case 6: System.out.println("Junho"); break;
            case 7: System.out.println("Julho"); break;
            case 8: System.out.println("Agosto"); break;
            case 9: System.out.println("Setembro"); break;
            case 10: System.out.println("Outubro"); break;
            case 11: System.out.println("Novembro"); break;
            case 12: System.out.println("Dezembro"); break;
            default: System.out.println("Mes inválido");
        }
    }
}
```

Figura 1.11: Exemplo de utilização do comando de decisão *switch...case*

Fonte: Elaborada pelos autores

1.11 Comandos de repetição

O Java fornece três instruções de repetição que permitem que programas executem instruções repetidamente, contanto que uma condição permaneça verdadeira. As instruções de repetição são *while*, *do...while* e *for* (DEITEL; DEITEL, 2010, p. 84).



Os comandos de repetição de Java funcionam exatamente como os da linguagem C.

Na Figura 1.12, temos um exemplo de uso do comando *for* que imprime números de 0 a 10 na tela.

```
package pacotel;
public class ExemploFor {
    public static void main(String[] args) {
        for( int i = 0; i <= 10; i++)
        {
            System.out.print(i + " ");
        }
        System.out.println();
        System.out.println("Acabou");
    }
}
```

Figura 1.12: Exemplo de utilização do comando de repetição *for*

Fonte: Elaborada pelos autores

A Figura 1.13 apresenta duas novas versões do mesmo exemplo; porém, nessas novas versões utilizamos os comandos *while* e *do-while* em lugar do comando *for*.

```
package pacotel;
public class ExemploWhile {
    public static void main(String[] args) {
        int i = 0;
        while (i <= 10) {
            System.out.print(i + " ");
            i++;
        }
        System.out.println();
        System.out.println("Acabou");
    }
}

package pacotel;
public class ExemploDoWhile {
    public static void main(String[] args) {
        int i = 0;
        do{
            System.out.print(i + " ");
            i++;
        }while (i <= 10);
        System.out.println();
        System.out.println("Acabou");
    }
}
```

Figura 1.13: Exemplos de utilização dos comandos de repetição *while* e *do-while*

Fonte: Elaborada pelos autores



Assim como nos comandos de repetição e de decisão da linguagem C, o uso das chaves em Java só é obrigatório quando temos mais de uma linha dentro do bloco. Em casos de blocos compostos por apenas uma linha, as chaves podem ser omitidas.

1.12 Escopo de variáveis

Em Java, assim como em linguagem C, podemos declarar variáveis a qualquer momento. Entretanto, o trecho de código em que a variável será válida dependerá de onde ela foi declarada.

Quando abrimos um novo bloco com as chaves, as variáveis declaradas ali dentro só existirão até o fim daquele bloco, ou seja, até o ponto onde se fecham as chaves do bloco.

```
public static void main(String [] args){
// Aqui a variável x ainda não existe
int x = 1;
// Aqui a variável x já existe
while ( <condição> ) {
    // Aqui a variável x continua existindo e y ainda não existe
    int y = 0;
    // Aqui a variável y já existe
}
// Aqui a variável já não existe mais. O x continua existindo.
}
```

Tenha cuidado! Se criar uma variável dentro de um bloco e tentar utilizá-la fora desse bloco, haverá um erro de compilação.

A-Z

Escopo da variável

É o nome dado ao trecho de código em que a variável existe e no qual é possível acessá-la.



1.13 Vetores e matrizes

Como já aprendemos nas disciplinas anteriores, vetor (*array*) é uma estrutura de dados utilizada para representar certa quantidade de variáveis de valores homogêneos, ou seja, um conjunto de variáveis, todas do mesmo tipo. Em Java podemos criar vetores de tipos primitivos ou de objetos.

```
Declaração: <tipo_do_dado> <nome_do_vetor> [] = new <tipo_do_dado>[quantidade];
Exemplo: int notas[] = new int[30];
```

No exemplo acima, será criado um vetor de 30 inteiros (inicializados com o valor 0). Também é possível inicializar um vetor com um conjunto de valores ao mesmo tempo em que o declaramos.

```
int notas[] = { 1,2,3};
```

Uma vez que o vetor já foi devidamente criado, sua utilização é idêntica à utilização de vetores em linguagem C, ou seja, acessamos cada elemento pelo seu índice (os índices de um vetor se iniciam do zero). Por exemplo:

```
i.nt numeros[] = new int[3];
numeros[0] = 57;
numeros[1] = 51;
numeros[3] = 37; // Esta linha gera um erro de execução!
```

A última linha do trecho de código acima causa um erro de compilação, pois o índice 3 não existe em um vetor com apenas três elementos.

Para sabermos o tamanho de um vetor, podemos utilizar o atributo *length*. A Figura 1.14 exibe um exemplo de utilização desse atributo.

```
package exemplos;
import java.util.Scanner;

public class ExemploVetor {

    public static void main(String[] args) {
        float notas[]; // Declaro o vetor mas ainda não aloco!
        /* Na próxima linha vou solicitar que o usuário informe a quantidade de
         * dados, lendo essa quantidade para a variável qtd.
         */
        System.out.println("Informe a quantidade de notas a serem digitadas:");
        Scanner scan = new Scanner(System.in);
        int qtd = scan.nextInt();
        // Agora que sei a quantidade vou alocar o vetor do tamanho desejado
        notas = new float[qtd];
        /* Vou fazer um laço para ler todas as notas. Note que usei o atributo
         * length como condição de parada do laço.
         */
        for (int i=0; i<notas.length;i++){
            System.out.println("Digite a nota do aluno numero " + i);
            notas[i]=scan.nextFloat();
        }
        /* Abaixo temos um laço que roda todo o vetor de notas imprimindo na
         * tela todas as notas atribuídas aos alunos.
         */
        for (int i=0; i<notas.length;i++){
            System.out.println("Nota do aluno numero " + i + " foi " + notas[i]);
        }
    }
}
```

Figura 1.14: Utilização de vetores e do atributo *length*

Fonte: Elaborada pelos autores

Vetores podem ter mais de uma dimensão, sendo conhecidos como vetores multidimensionais ou matrizes. As sintaxes para declaração e uso de matrizes são idênticas às de vetores acrescentando valores para cada uma de suas dimensões. Abaixo é apresentado um exemplo:

```
int matriz[][] = new int[10][10];
matriz[9][9] = -3;
matriz[2][10]=5;//Erro de compilação!! A segunda dimensão está acima do máximo!
```

A partir do momento em que um vetor ou matriz foi criado, ele não pode mudar de tamanho. Se você precisar de mais espaço, será necessário criar um novo e copiar os elementos do vetor antigo para o novo.

Resumo

Consideramos que o aluno desta disciplina já estudou linguagem C ou a própria linguagem Java em disciplinas anteriores. Assim, nesta aula fizemos uma rápida apresentação sobre a plataforma Java e uma revisão geral sobre a linguagem Java, destacando que a sintaxe dessa linguagem é idêntica à da linguagem C. Nesse contexto, vimos como declarar variáveis, constantes e vetores e como fazer comentários em Java. Apresentamos os operadores aritméticos, lógicos e relacionais da linguagem e conhecemos os comandos de decisão e repetição da linguagem. Enfim, nesta aula relembramos vários conceitos e começamos a “esquentar as turbinas” para nossa viagem ao mundo da programação orientada a objetos



Leia mais sobre as características básicas de Java em <http://javafree.uol.com.br/artigo/871496/Tutorial-Java-2- Caracteristicas-Basicasindex>

Segundo o texto acima, Java possui 49 palavras-chave. Mas, o que vem a ser palavra-chave no contexto de linguagens de programação? O que acontece se eu definir uma variável que tenha nome igual ao de alguma palavra-chave?



Atividades de aprendizagem

1. Faça um programa que dada a idade de uma pessoa verifique sua classe eleitoral:
 - menor que 16 anos não pode votar;
 - com 16 ou 17 anos ou mais que 65 anos, votar é facultativo;
 - entre 18 e 65 anos (inclusive), votar é obrigatório.
2. Faça um programa que imprima os trinta primeiros elementos da série de Fibonacci. A série é a seguinte: 1, 1, 2, 3, 5, 8, 13 etc. Para calculá-la, o primeiro e segundo elementos valem 1; daí por diante, cada elemento vale a soma dos dois elementos anteriores.
3. Crie um programa que armazene um vetor com as notas de dez alunos, calcule e imprima a média dessas notas e depois informe a quantidade de notas acima e abaixo da média calculada.
4. Faça um programa que fique em laço solicitando a digitação de números inteiros maiores ou iguais a zero. Quando o usuário digitar um número menor que zero, o programa deve exibir a quantidade de números digitados e a média desses números.

Aula 2 – Introdução à Orientação a Objetos

Objetivos

Conhecer os conceitos fundamentais de orientação a objetos: objetos, classes, métodos, atributos e pacotes.

Aprender a criar e a utilizar classes e objetos em Java.

Conhecer e aprender a utilizar a classe String e a classe ArrayList.

2.1 Conceitos básicos

Todos os sistemas de computação, dos mais complexos aos mais simples programas de computadores, são desenvolvidos para auxiliar na solução de problemas do mundo real. Mas, de fato, o mundo real é extremamente complexo. Nesse contexto, a orientação a objetos tenta gerenciar a complexidade inerente aos problemas do mundo real abstraindo o conhecimento relevante e encapsulando-o dentro de objetos.

Na programação orientada a objetos um programa de computador é conceituado como um conjunto de objetos que trabalham juntos para realizar uma tarefa. Cada objeto é uma parte do programa, interagindo com as outras partes de maneira específica e totalmente controlada (CADENHEAD; LEMAY,2005, p. 6).

Na visão da orientação a objetos, o mundo é composto por diversos **objetos** que possuem um conjunto de características e um comportamento bem definido. Assim, quando programamos seguindo o paradigma de orientação a objetos, definimos abstrações dos objetos reais existentes. Todo **objeto** possui as seguintes características:

- Estado: conjunto de propriedades de um objeto (valores dos atributos).
- Comportamento: conjunto de ações possíveis sobre o objeto (métodos da classe).
- Unicidade: todo objeto é único (possui um endereço de memória).

Quando você escreve um programa em uma linguagem orientada a objetos, não define objetos individuais. Em vez disso define as classes utilizadas para criar esses objetos (CADENHEAD; LEMAY, 2005, p. 7).

A-Z

Classe

É um componente de programa que descreve a “estrutura” e o “comportamento” de um grupo de objetos semelhantes – isto é, as informações que caracterizam o estado desses objetos e as ações (ou operações) que eles podem realizar.



Assim, podemos entender uma **classe** como um modelo ou como uma especificação para um conjunto de objetos, ou seja, a descrição genérica dos objetos individuais pertencentes a um dado conjunto. A partir de uma classe é possível criar quantos objetos forem desejados.

Uma **classe** define as características e o comportamento de um conjunto de objetos. Assim, a criação de uma classe implica definir um tipo de objeto em termos de seus atributos (variáveis que conterão os dados) e seus métodos (funções que manipulam tais dados).

Fazendo um paralelo com a linguagem C, podemos visualizar os métodos das classes como funções e os atributos de uma classe como campos de uma *struct*.

A-Z

Objeto

É a criação de uma instância da classe. Quando instanciamos um objeto, criamos fisicamente uma representação concreta de sua classe.

“Ser Humano” é uma classe, um molde, já “Roberto” é uma instância de “Ser Humano”. Apesar de carregar todas as características do molde “Ser Humano”, ele é completamente diferente (independente) das outras instâncias de “Ser Humano”.

Pacotes

Em Java são uma maneira de agrupar classes e interfaces relacionadas. Pacotes permitem que grupos de classes estejam disponíveis apenas se forem necessários e eliminam possíveis conflitos entre nomes de classes em diferentes grupos de classes (CADENHEAD; LEMAY, 2005, p. 17).



Um sistema bem elaborado organiza suas classes em pacotes logicamente consistentes, facilitando, assim, sua manutenção, extensibilidade e flexibilidade.

Já um **objeto** é uma **instância** de uma classe. Ele é capaz de armazenar estados por meio de seus atributos e reagir a mensagens enviadas a ele, assim como se relacionar e enviar mensagens a outros objetos.

João, José e Maria, por exemplo, podem ser considerados objetos de uma classe **Pessoa** que tem como atributos **nome, CPF, data de nascimento**, entre outros, e métodos como **calcularIdade**.

Quando desenvolvemos um sistema orientado a objetos, definimos um conjunto de classes. Para organizar essas classes surge o conceito de pacote. **Pacote** é um envoltório de classes, ou seja, guarda classes e outros pacotes logicamente semelhantes ao pacote que os contém. Podemos visualizar os pacotes como diretórios ou pastas, nos quais podemos guardar arquivos (classes) e outros diretórios (pacotes) que tenham conteúdo relacionado com o pacote que os contém.

2.2 Classes em Java

Sabemos que para desenvolver programa orientado a objetos precisamos criar classes. Uma classe é como um tipo de objeto que pode ser definido pelo programador para descrever uma entidade real ou abstrata. Mas, como definir classes em Java?

Sintaxe:

```
<qualificador> class <nome_da_classe>{  
  
    <declaração dos atributos da classe>  
  
    <declaração dos métodos da classe>  
  
}
```

Como funciona?

<qualificador>: O qualificador de acesso determinará a visibilidade da classe. Pode ser *public* (classe pública) ou *private* (classe privada). Classes privadas só poderão ser visualizadas dentro de seu próprio pacote enquanto as públicas serão acessíveis por qualquer classe de qualquer pacote. Se o qualificador for omitido, a classe será privada por padrão.

<nome>: nome que identifica a classe. Há um padrão entre os programadores de sempre iniciarem os nomes de classes com letras maiúsculas. Mas, apesar de ser uma boa prática, seguir esse padrão não é uma obrigação.

Como exemplo, a Figura 2.1 ilustra a criação de uma classe *Conta* para representar os dados de uma conta bancária. Nesse exemplo definimos três atributos para a classe *Conta*, mas não definimos métodos.

```
1 package exemplos;  
2  
3 public class Conta {  
4     int numero;  
5     String nome_titular;  
6     double saldo;  
7 }
```

Figura 2.1: Criação de classe em Java

Fonte: Elaborada pelos autores

Para ilustrar a utilização do qualificador de acesso, veja o exemplo apresentado nas Figuras 2.2 e 2.3 a seguir.



Para criar uma nova classe em um projeto do NetBeans já existente, siga os seguintes passos:

1. Clique em "Arquivo" > "Novo Arquivo".
2. Dentre as opções "Categorias" da janela aberta, escolha "Java" e dentre as opções "Tipos de Arquivo", escolha "Classe Java". Clique em "Próximo".
3. Dê um nome para a classe e indique o nome do pacote no qual a classe será criada. Clique em "Finalizar".

| | |
|---|--|
| <pre>package pacote1; class ClassePrivada { int atributo1; }</pre> | <pre>package pacote1; public class ClassePublica { int atributo1; }</pre> |
|---|--|

Figura 2.2: Criação de classes públicas e privadas

Fonte: Elaborada pelos autores

A Figura 2.2 ilustra a criação de duas classes que diferem apenas por um detalhe: uma delas é pública e a outra é privada. Note que ambas estão no *pacote1* (primeira linha do código).

A Figura 2.3 exibe o código de uma classe que está em um outro pacote (*pacote2*), mas tenta utilizar as classes criadas na Figura 2.2.

```
package pacote2;

import pacote1.ClassePublica;
import pacote1.ClassePrivada;

public class ClasseOutroPacote {
    ClassePublica obj1;
    ClassePrivada obj2;
}
```

Figura 2.3: Tentativa de acessar classes públicas e privadas

Fonte: Elaborada pelos autores



Para criar um novo pacote em um projeto do NetBeans já existente, faça os seguintes passos:

1. Clique em "Arquivo" > "Novo Arquivo".
2. Dentre as opções "Categorias" da janela aberta, escolha "Java" e dentre as opções "Tipos de Arquivo", escolha "Pacote Java". Clique em "Próximo".

As linhas sublinhadas em vermelho indicam erro no NetBeans. Os erros ocorreram exatamente nas linhas em que tentamos utilizar a classe privada em um pacote externo ao pacote no qual ela foi criada. Mas, note que, como a classe *ClasseOutroPacote* está em um pacote diferente das classes que definimos no exemplo da Figura 2.2, precisamos importar as classes antes de utilizá-las (segunda e terceira linha do código).

2.3 Declaração de atributos e métodos

Já vimos que quando criamos uma classe em Java precisamos declarar os atributos e métodos da classe.

Para declarar um atributo, utilizamos a seguinte sintaxe:

Sintaxe:

```
<qualificador> <tipo_do_atributo> <nome_do_atributo>;
```

Como funciona?

<qualificador>: O qualificador de acesso determinará a visibilidade do atributo. É opcional e, se não for informado, por padrão o atributo será protegido (*protected*). Não se preocupe com isso agora. Voltaremos a falar sobre os qualificadores quando estudarmos encapsulamento.

<tipo_do_atributo>: é um tipo primitivo ou classe que define o atributo.
<nome_do_atributo>: nome que identifica o atributo. Há um padrão entre os programadores de sempre iniciarem os nomes de atributos com letras minúsculas. Mas, apesar de ser uma boa prática, seguir esse padrão não é uma obrigação. Caso queiramos definir vários atributos de mesmo tipo podemos colocar os vários nomes separados por vírgula.

Exemplos:

```
private double saldo;
```

```
String nome, endereço;
```

Para declararmos um método, utilizamos a seguinte sintaxe:

Sintaxe:

```
<qualificador> <tipo_de_retorno> <nome_do_metodo> (<lista_de_argumentos>){
```

```
    <corpo_do_método>
```

```
}
```

Como Funciona?

<qualificador>: Mesmo conceito usado no caso de atributos.

<tipo_do_retorno>: é um tipo primitivo ou classe que define o retorno a ser dado pelo método.

<nome_do_método>: nome que identifica o método.

<corpo_do_método>: código que define o que o método faz.

Exemplo:

```
public double getSaldo(){  
  
    return saldo;  
  
}
```

A Figura 2.4 exibe uma nova versão da classe *Conta*, agora com um método *depositar*.

```
package exemplos;  
  
public class Conta {  
  
    int numero;  
    String nome_titular;  
    double saldo;  
  
    void depositar(double valor) {  
        this.saldo = this.saldo + valor;  
    }  
  
}
```

Figura 2.4: Classe *Conta* com método *depositar*

Fonte: Elaborada pelos autores

O método *depositar* recebe como argumento o valor a ser depositado na conta e soma esse valor ao saldo da conta. Note que no lugar do tipo de retorno do método foi utilizada a palavra *void*, que indica que nenhum valor será retornado pelo método. O uso da palavra *void* em Java é o mesmo que é feito na linguagem C.

Outro detalhe interessante no exemplo da Figura 2.4 é o uso da palavra *this*. A palavra *this* é utilizada para acessar atributos do objeto corrente. A palavra *this* é obrigatória apenas quando temos um argumento ou variável local de mesmo nome que um atributo. Entretanto, é recomendável sua utilização, mesmo quando não obrigatório, por uma questão de padronização.

A Figura 2.5 exibe um novo método para a classe *Conta*: o método *sacar*.

```
boolean sacar(double valor) {
    if (this.saldo >= valor) {
        this.saldo -= valor;
        return true;
    }
    else
        return false;
}
```

Figura 2.5: Método sacar

Fonte: Elaborada pelos autores

O método *sacar* recebe como argumento o valor a ser sacado. Então, o método verifica se a conta tem saldo suficiente para permitir o saque. Se o saldo for maior ou igual ao valor do saque, o valor do saque é descontado do saldo e o método retorna o valor *true* informando que o saque ocorreu com sucesso. Caso contrário, o método apenas retorna o valor *false*, indicando que o saque não pôde ser efetuado. Por isso, o tipo de retorno do método é *boolean*.

2.4 Utilização de objetos

Para utilizar um objeto em Java, precisamos executar dois passos, a saber:

- **Declarar uma variável que referenciará o objeto:** assim como fazemos com tipos primitivos, é necessário declarar o objeto.
- **Instanciar o objeto:** alocar o objeto em memória. Para isso utilizamos o comando *new* e um construtor.

A Figura 2.6 exibe o exemplo de uma classe de um programa que utiliza a classe *Conta* apresentada em exemplo anterior.

```

1 package exemplos;
2
3 public class Programa {
4
5     public static void main(String[] args) {
6         Conta c; //Declarando a variável que conterà a referência ao objeto
7         c = new Conta();//Instanciando um objeto em memória
8         c.nome_titular = "Zé";
9         c.depositar(100);
10        System.out.println("Titular: "+ c.nome_titular);
11        System.out.println("Saldo Atual: "+ c.saldo);
12    }
13 }

```

Figura 2.6: Criação de instâncias da classe *Conta*

Fonte: Elaborada pelos autores

A classe *Conta* tem por objetivo mapear os dados e comportamentos de uma conta bancária. Assim, ela não terá um método *main*. Esse método estará na classe principal do nosso programa. A classe *Programa* exibida na Figura 2.6 é que representa esse nosso programa principal e é nela que utilizaremos a classe *Conta*.

Note que na linha 6 do exemplo 2.6 é declarada uma variável de nome *c* que referenciará um objeto da classe *Conta* enquanto na linha 7 o objeto é instanciado em memória. Para instanciar um objeto em memória utilizamos o operador *new* (o mesmo que utilizamos para alocar vetores) acompanhado pelo construtor *Conta()*. Logo estudaremos mais sobre construtores. No momento encare como um método que tem nome igual ao da classe e não recebe argumento nenhum. Na linha 8 é atribuído o valor "Zé" ao atributo *nome_titular* da conta instanciada e, na linha 9, há uma chamada ao método *depositar* (já exibido em exemplo anterior) passando como argumento o valor 100. Preste atenção ao *ponto* que foi utilizado nas linhas 8 (*c.nome_titular*) e 9 (*c.depositar(100)*). É assim que acessamos métodos e atributos de um objeto em Java. Nas linhas 10 e 11 do código apresentado pela Figura 2.6 há dois comandos de saídas de dados que imprimirão na tela o nome do titular e o saldo atual da conta, respectivamente.



Note que em nosso exemplo da Figura 2.6 foi possível acessar o atributo *nome_titular* e o método *depositar()* da classe *Conta*. Esse acesso foi possível, pois na definição dessa classe *Conta* (feita na Figura 2.4) não definimos um qualificador de acesso para o atributo e nem para o método. Logo, tanto o método quanto o atributo são, por padrão, protegidos (*protected*). Além disso, a classe *Conta* (Figura 2.4) está no mesmo pacote que a classe *Programa* (Figura 2.5). Em breve estudaremos o conceito de encapsulamento e veremos que essa não é uma boa estratégia para controlar os acessos a atributos de uma classe.

O método *depositar* não retorna nenhum valor para a classe que o chamou (note na Figura 2.4 que o retorno dele é *void*). Já o método *sacar* (definido na Figura 2.5), por exemplo, retorna um *boolean* que indica se o saque pôde ser efetuado com sucesso ou não. Mas, então, como utilizar o valor retornado por um método? A Figura 2.7 mostra um exemplo em que o método *sacar()* é chamado pelo programa principal e seu retorno é utilizado para exibir uma mensagem informando se o saque foi realizado com sucesso ou não.

```
1 package exemplos;
2
3 public class Programa {
4
5     public static void main(String[] args) {
6
7         Conta c = new Conta();
8         c.depositar(200);
9         boolean saque_efetuado = c.sacar(250);
10        if (saque_efetuado)
11            System.out.println("Saque Efetuado com Sucesso");
12        else
13            System.out.println("Saque não efetuado! Saldo insuficiente!");
14    }
15 }
```

Figura 2.7: Utilização do valor retornado por um método

Fonte: Elaborada pelo autor

Note que na linha 9 do exemplo apresentado é criada uma variável *booleana* com o nome *saque_efetuado*, que receberá o valor retornado pela chamada ao método *sacar()*. Depois, o valor dessa variável é utilizado para decidir entre exibir a mensagem de “Saque Efetuado com Sucesso” ou a de “Saque não efetuado”.

2.5 Atributos e métodos estáticos

Atributos estáticos são atributos que contêm informações inerentes a uma classe e não a um objeto em específico. Por isso são também conhecidos como atributos ou **variáveis de classe**.

Por exemplo, suponha que quiséssemos ter um atributo que indicasse a quantidade de contas criadas. Esse atributo não seria inerente a uma conta em específico, mas a todas as contas. Assim, seria definido como um atributo estático. Para definir um atributo estático em Java, basta colocar a palavra *static* entre o qualificador e o tipo do atributo.

O mesmo conceito é válido para métodos. Métodos estáticos são inerentes à classe e, por isso, não nos obrigam a instanciar um objeto para que possamos utilizá-los. Para definir um método como estático, basta utilizar a palavra *static*, a exemplo do que acontece com atributos. Para utilizar um método estático devo utilizar o nome da classe acompanhado pelo nome do método.

A-Z

Variável de classe

Define um atributo de uma classe inteira. A variável se aplica à própria classe e a todas as suas instâncias, de modo que somente um valor é armazenado, não importando quantos objetos dessa classe tenham sido criados (CADENHEAD; LEMAY, 2005, p. 9).

Métodos estáticos são muito utilizados em classes do Java que proveem determinados serviços. Por exemplo, Java fornece na classe `Math` uma série de métodos estáticos que fazem operações matemáticas como: raiz quadrada (`sqrt`), valor absoluto (`abs`), seno (`sin`) entre outros. A Figura 2.8 apresenta exemplos de uso dos métodos da classe `Math` a fim de ilustrar a utilização de métodos estáticos.

```
double x = Math.sqrt(634); //Atribui a x o valor da raiz quadrada de 634
double z = Math.sin(4); //Atribui a x o valor do seno de 4 radianos
double y = Math.PI; //Atribui a y o valor da constante matemática pi.
```

Figura 2.8: Utilização de métodos estáticos da classe `Math`

Fonte: Elaborada pelos autores

Note no exemplo da Figura 2.8 que não instanciamos objetos da classe `Math` para utilizar seus métodos e constantes, pois são estáticos.



Um método criado como estático só poderá acessar atributos que também sejam estáticos, além dos seus argumentos e variáveis locais.

2.6 A classe `String`

Já estudamos que Java não conta com um tipo primitivo para trabalhar com cadeia de caracteres. Para isso temos em Java a classe `String`.



String

É uma classe de Java utilizada para representarmos uma cadeia de caracteres, que oferece diversos métodos para manipularmos essa cadeia.

Para criar uma instância de `String`, não precisamos utilizar o operador `new`, como acontece com as outras classes. Para instanciar um objeto do tipo `String`, basta declarar uma variável desse tipo e iniciá-la com um valor. É importante saber também que objetos da classe `String` podem ser concatenados utilizando o operador `+`.

Para comparar se os valores de duas `Strings` são iguais, utilizamos o método `equals` e não o operador `==` que é utilizado para tipos primitivos.

A classe `String` conta ainda com diversos métodos muito úteis, dentre os quais podemos destacar:

`length`: retorna o tamanho (tipo `int`) de uma `String`.

`charAt`: retorna o *character* (`char`) da `String` que se localiza no índice passado como parâmetro. Vale ressaltar que o primeiro índice de uma `String` é o índice zero.

`toUpperCase`: retorna uma `String` com todas as letras maiúsculas a partir da `String` que chamou o método.

toLowerCase: retorna uma *String* com todas as letras minúsculas a partir da *String* que chamou o método.

trim: retorna uma *String* sem espaços em branco no início e no final dela, a partir da *String* que chamou o método.

replace: Retorna uma *String* com *substrings* trocadas, a partir da *String* que chamou o método. As trocas são feitas de acordo com os parâmetros do método: em que aparecer a *substring1* será substituída pela *substring 2*.

valueOf: retorna uma *String* a partir de um valor de outro tipo, como um número por exemplo.

A Figura 2.9 apresenta um exemplo de programa que utiliza esses métodos da classe *String* e, a Figura 2.10 exibe o resultado da execução de tal programa.

```
package exemplos;

public class ExemploString {

    public static void main(String[] args) {
        String nome = "José";
        String frase = " Meu nome é ";
        String completa = frase + nome; //Concateno 2 Strings formando uma nova
        System.out.println(completa + "! ");
        System.out.println("O tamanho do nome é: " + nome.length());
        System.out.println("O caracter da posicao 2 do nome é "+nome.charAt(2));
        System.out.println("Frase Completa toda em maiusculas:"+completa.toUpperCase());
        System.out.println("Substring de 2 a 8: " + completa.subSequence(2,8));
        System.out.println("Tirando os espaços antes e depois da frase completa: " + completa.trim());
        System.out.println("Substituindo José por João na frase completa: " + completa.replace("José", "João"));
    }
}
```

Figura 2.9: Exemplos de utilização dos métodos de *String*

Fonte: Elaborada pelos autores

```
run:
  Meu nome é José!
O tamanho do nome é: 4
O caracter da posicao 2 do nome é s
Frase Completa toda em maiusculas: MEU NOME É JOSÉ
Substring de 2 a 8: Meu no
Tirando os espaços antes e depois da frase completa: Meu nome é José
Substituindo José por João na frase completa: Meu nome é João
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Figura 2.10: Saída gerada pelo exemplo da Figura 2.9

Fonte: Elaborada pelos autores

2.7 Listas

A estrutura de dados lista é utilizada para armazenar conjuntos de elementos. A vantagem em utilizar listas em lugar de vetores é o fato de as listas serem alocadas dinamicamente de forma que não precisamos prever seu tamanho máximo. Java fornece classes que implementam o conceito de lista. Nesse aula utilizaremos uma dessas classes: o *ArrayList*. Para trabalharmos com a classe *ArrayList* precisamos conhecer seus métodos. Seguem os principais:

- `public ArrayList():` cria um `ArrayList` vazio.
- `public boolean add(<elemento>):` adiciona um elemento no final da lista.
- `public void add(index, <elemento>):` adiciona um elemento na posição `index`.
- `public <elemento> get(int index):` obtém o elemento de índice `index`.
- `public <elemento> remove(int index):` retorna o elemento de índice `index` e o elimina da lista.
- `public boolean isEmpty():` verifica se a lista está vazia.

Para navegarmos em uma lista, utilizaremos a *interface Iterator*. Seguem os principais métodos de *Iterator*:

- `boolean hasNext():` verifica se existe próximo elemento na lista;
- `next():` obtém o elemento sob o *Iterator* e avança para o próximo elemento;
- `void remove():` remove o elemento sob o *Iterator*.

A Figura 2.11 exibe um exemplo no qual criamos duas instâncias da classe `Conta`, inserindo-as em uma lista; depois navegamos na lista exibindo os números das contas da lista.

```
package exemplos;

import java.util.ArrayList;
import java.util.Iterator;

public class ExemploLista {

    public static void main(String[] args) {
        ArrayList lista = new ArrayList();
        Conta c = new Conta();
        c.numero = 1;
        lista.add(c);
        c = new Conta();
        c.numero = 2;
        lista.add(0, c);
        Iterator i = lista.iterator();
        while(i.hasNext()) {
            c=(Conta)i.next();
            System.out.println("Conta Numero: "+ c.numero);
        }
    }
}
```

Figura 2.11: Exemplo de utilização de `ArrayList`

Fonte: Elaborada pelos autores

O *ArrayList* pode armazenar objetos de quaisquer tipo. Assim, quando obtemos um objeto ele retorna uma instância do tipo *Object*. Por isso, no exemplo foi necessário fazer a conversão explícita para o tipo *Conta*.



Resumo

Nessa aula iniciamos nossos estudos sobre orientação a objetos. Conhecemos os conceitos fundamentais da orientação a objetos e aprendemos a implementá-los em Java. Por fim, conhecemos duas classes muito úteis de Java: *String* e *ArrayList*.

Atividades de aprendizagem

1. Implementar uma classe *Conta* tendo como atributos o nome do titular, número e saldo e os métodos sacar e depositar (seguindo os exemplos das Figuras 2.4 e 2.5).
2. Criar um programa principal que instancie uma *Conta* (exercício anterior). Solicite ao usuário os dados da conta atribuindo os valores informados aos seus atributos. Depois o sistema deve entrar em laço exibindo as seguintes opções para o usuário: digitar 1 para depositar, 2 para sacar ou outro número para terminar a execução. Se o usuário digitar 1, o sistema deve solicitar o valor a ser depositado, chamar o método *depositar*, exibir o saldo atualizado e voltar a exibir as opções. Se o usuário digitar 2, o sistema deve solicitar o valor a ser sacado, chamar o método *sacar* verificando o retorno. Se o retorno for *true*, exibir uma mensagem de saque efetuado com sucesso; caso contrário, uma mensagem de saque não efetuado e voltar a exibir as opções.
3. Vamos avançar o exercício anterior. Agora o programa principal terá uma lista de instâncias de *Conta*. A lista iniciará vazia. O programa entrará em um laço que exibirá as seguintes opções para o usuário: digitar 1 para criar uma conta, 2 para ver o saldo de uma conta, 3 para sacar, 4 para depositar e outro número para finalizar. Se o usuário escolher a opção 1, o sistema deve instanciar uma nova conta, solicitar o nome do titular e o saldo inicial atribuindo esses valores à conta criada. O número da conta será sequencial, atribuído pelo próprio programa. Então o programa exibirá o número da conta criada, adicionará a conta à lista e exibirá as opções novamente. Se o usuário digitar opção 2, 3 ou 4, perguntar o número da conta que o usuário deseja e localizar essa conta na lista. Depois



Uma outra classe fornecida por Java para implementar o conceito de listas é a classe *LinkedList*. Acesse as documentações oficiais da Oracle sobre as classes *ArrayList* e *LinkedList* respectivamente em <http://download.oracle.com/javase/6/docs/api/java/util/ArrayList.html> e <http://download.oracle.com/javase/6/docs/api/java/util/LinkedList.html>.

Utilizando as informações dessas documentações, reimplente o exemplo apresentado na Figura 2.11 utilizando a classe *LinkedList* em lugar da classe *ArrayList*.

de localizar a conta, se a opção tiver sido 2, o sistema apenas exibe uma mensagem informando o saldo da conta solicitada. Se for 3, o sistema deve efetuar um saque na conta seguindo o mesmo procedimento do exercício 2. Se for 4, o sistema deve efetuar um saque na conta seguindo o mesmo procedimento do exercício 2.

Aula 3 – Construtores, destrutores e encapsulamento

Objetivos

Entender os conceitos de construtores e destrutores.

Entender o conceito de encapsulamento e sua importância.

Criar programas com maior manutenibilidade e extensibilidade pela utilização do conceito de encapsulamento

3.1 Construtores

Como estudamos, sempre que queremos criar um novo objeto de uma determinada classe utilizamos a palavra *new* acompanhada por um **construtor**.

O construtor de uma classe tem, por definição, o mesmo nome que a classe. A Figura 3.1 exibe dois exemplos de construtores para a classe *Conta* que utilizamos em exemplos da aula anterior.

```
public Conta(int numero, String nome_titular, double saldo){
    this.numero = numero;
    this.nome_titular=nome_titular;
    this.saldo = saldo;
}

public Conta(int numero, String nome_titular){
    this.numero = numero;
    this.nome_titular=nome_titular;
    saldo = 0;
}
```

Figura 3.1: Métodos construtores para a classe *Conta*

Fonte: Elaborada pelos autores

É possível definir diversos construtores para a mesma classe, tendo os tipos ou a quantidade de parâmetros diferentes para cada um deles.

O primeiro construtor do exemplo da Figura 3.1 cria objetos já atribuindo aos mesmos valores passados como parâmetros para os atributos *numero*, *nome_titular* e *saldo*. Já o segundo construtor recebe como parâmetros valores apenas para os atributos *numero* e *nome_titular* e atribui o valor zero ao *saldo*. Note a utilização da palavra reservada *this* para diferenciar os atributos dos parâmetros de mesmo nome.

A-Z

Construtor

É um método especial para a criação e inicialização de uma nova instância de uma classe. Um construtor inicializa o novo objeto e suas variáveis, cria quaisquer outros objetos de que ele precise e realiza quaisquer outras operações de que o objeto precise para inicializar-se (CADENHEAD; LEMAY, 2005, p. 41).



A Figura 3.2 exibe o código de um programa que cria dois objetos da classe *Conta*, utilizando cada um dos construtores apresentados no exemplo da Figura 3.1.

```
package exemplos;

public class Programa {

    public static void main(String[] args) {
        Conta c1, c2;
        c1 = new Conta(1, "Zé", 0);
        c2 = new Conta(2, "João");
    }
}
```

Figura 3.2: Criação de instâncias da classe *Conta*

Fonte: Elaborada pelos autores

Note que na criação do objeto *c1* foi chamado o construtor que recebe três parâmetros enquanto na criação do objeto *c2* foi chamado o outro construtor que recebe apenas dois parâmetros. A conta referenciada por *c1* terá como o valor *1* para o atributo *numero*, o valor *Zé* para o atributo *nome_titular* e o valor *0* para o saldo.



O Java define um construtor padrão para classes que não tem nenhum construtor definido. O construtor padrão não recebe nenhum argumento. No exemplo apresentado na Figura 2.4 (aula anterior), como a classe *Conta* não tinha construtores definidos, Java utilizou o construtor padrão para instanciar os objetos. Porém, a partir do momento em que você declara um construtor, o construtor padrão deixa de existir. Assim, agora que criamos esses dois construtores é impossível criar uma nova instância de *Conta* sem utilizar um dos dois.

3.2 Destruutores

Em C, aprendemos que sempre devíamos desalocar (comando *free*) tudo o que alocássemos dinamicamente em memória. Em Java isso não é necessário, pois essa linguagem possui um **Coletor Automático de Lixo (Garbage Collector)**, o qual é responsável por desalocar tudo aquilo que não é mais utilizado. Assim, os objetos que não são mais referenciados por um programa são automaticamente desalocados por esse coletor, liberando memória.

A-Z

Método destrutor

É um método acionado imediatamente antes de o objeto ser desalocado.

Em Java, o **método destrutor** de uma classe é o método *finalize*. Quando não é definido um método destrutor para uma classe, Java utiliza um método destrutor padrão que não faz nada.

O método *finalize* é raramente utilizado porque pode causar problemas de desempenho e há uma incerteza sobre se ele será chamado (DEITEL; DEITEL, 2010, p. 258).

3.3 Encapsulamento

Encapsulamento é uma técnica utilizada para restringir o acesso a variáveis (atributos), métodos ou até à própria classe. Os detalhes da implementação ficam ocultos ao usuário da classe, ou seja, o usuário passa a utilizar os métodos de uma classe sem se preocupar com detalhes sobre como o método foi implementado internamente.

Para facilitar o entendimento, façamos uma analogia com um carro. Para dirigir um carro uma pessoa não precisa conhecer os detalhes sobre como funciona o motor ou os demais componentes dele. Um motorista não precisa saber o que acontece internamente no carro quando ele acelera ou quando troca de marcha. Para dirigir ele precisa apenas saber como dirigir o carro utilizando pedais de acelerador, freio e embreagem, volante e alavanca de câmbio. Esses componentes encapsulam toda a complexidade existente no carro sob a ótica do motorista que o utiliza.

A ideia do encapsulamento na programação orientada a objetos é que não seja permitido acessarmos diretamente as propriedades de um objeto. Nesse caso, precisamos operar sempre por meio dos métodos pertencentes a ele. A complexidade de um objeto é escondida, portanto, pela abstração de dados que estão “por trás” de suas operações.

Quando aprendemos a criar classes, vimos que na definição de cada método e atributo poderíamos definir um qualificador de acesso, mas foi falado que ainda não era hora de nos preocuparmos com tais detalhes e que poderíamos omitir o qualificador. Pois bem, são exatamente esses qualificadores que nos permitem implementar o encapsulamento.

Esses qualificadores nos permitem modificar o nível de acesso aos atributos, aos métodos e até mesmo às classes. São três os possíveis qualificadores de acesso em Java:



Encapsulamento

Consiste na separação entre os aspectos externos de um objeto, acessíveis por outros objetos.

- *public* (público): indica que o método ou o atributo são acessíveis por qualquer classe, ou seja, que podem ser usados por qualquer classe, independentemente de estarem no mesmo pacote ou estarem na mesma hierarquia;
- *private* (privado): indica que o método ou o atributo são acessíveis apenas pela própria classe, ou seja, só podem ser utilizados por métodos da própria classe;
- *protected* (protegido): indica que o atributo ou o método são acessíveis pela própria classe, por classes do mesmo pacote ou classes da mesma hierarquia (estudaremos hierarquia de classes quando tratarmos de herança).



Quando omitimos o qualificador de acesso, o atributo ou método são considerados *protected* por padrão.

Agora que entendemos o conceito de encapsulamento e conhecemos os três qualificadores de acesso, vem a pergunta: por que restringir o acesso aos atributos de uma classe?

Para responder a essa pergunta, utilizaremos como exemplo a classe *Conta* criada em aula anterior. Quando o método *sacar()* daquela classe foi criado, tomamos o cuidado de não permitir saques maiores que o saldo, a fim de que este nunca ficasse negativo. Mas, como não nos preocupamos em restringir o acesso aos atributos daquela classe, um problema que poderia acontecer seria de alguém alterar o saldo, atribuindo um número negativo diretamente ao atributo, como no exemplo da Figura 3.3:

```
public static void main(String[] args) {  
    Conta c = new Conta();  
    c.nome_titular = "Ana";  
    c.saldo = -100.0; //Note como é possível atribuir um valor negativo...
```

Figura 3.3: Alterando o valor do atributo saldo diretamente

Fonte: Elaborada pelos autores

Assim, o melhor que teríamos a fazer seria impossibilitar o acesso ao atributo *saldo*, obrigando que as alterações a ele fossem obrigatoriamente feitas pelos métodos *depositar()* e *sacar()*.

De forma geral, a ideia do encapsulamento é a de que cada classe é responsável por controlar seus atributos; portanto, ela deve julgar se aquele novo valor é válido ou não. Essa validação não deve ser controlada por quem está

utilizando a classe, e sim pela própria classe, centralizando essa responsabilidade e facilitando futuras mudanças no sistema. Assim, em geral, os atributos de uma classe devem ser privados e deve haver métodos públicos que permitam o acesso a eles. A Figura 3.4 apresenta uma nova versão da classe *Conta* seguindo essa *ideia* (os construtores não são exibidos na figura).

```
public class Conta {
    private int numero;
    private String nome_titular;
    private double saldo;

    public void depositar(double valor) {
        this.saldo = this.getSaldo() + valor;
    }

    public boolean sacar(double valor) {
        if (this.getSaldo() >= valor) {
            this.saldo -= valor;
            return true;
        }
        return false;
    }

    public double getSaldo() {
        return saldo;
    }

    public int getNumero() {
        return numero;
    }

    public String getNome_titular() {
        return nome_titular;
    }

    public void setNome_titular(String nome_titular) {
        this.nome_titular = nome_titular;
    }
}
```

Figura 3.4: Classe *Conta* com atributos encapsulados

Fonte: Elaborada pelos autores

Os atributos *private* de uma classe só podem ser manipulados pelos métodos da classe. Portanto, um cliente de um objeto – isto é, qualquer classe que utilize o objeto – deverá chamar os métodos *public* da classe para manipular os campos *private* de um objeto da classe (DEITEL; DEITEL, 2010, p. 66).



Por padrão, os atributos “encapsulados” devem ter um método que obtenha o valor atual do atributo (método *get*) e um método que altere o valor do atributo (método *set*). Por exemplo, note que na nova versão da classe *Conta* há um método *getNomeTitular()* que retorna o nome do titular da conta e um método *setNomeTitular(String)* que atribui um novo nome ao ti-



O NetBeans oferece uma funcionalidade que gera automaticamente os métodos *get* e *set* para os atributos. Para utilizar essa funcionalidade, basta clicar com o botão direito do *mouse* sobre o nome da classe e escolher as opções "Refatorar > Encapsular Campos". Depois, na nova tela que é exibida, escolha os métodos que deseja que sejam criados e clique no botão "Refatorar".

tular da conta. Mas, como consideramos que o número da conta é atribuído em sua criação (note que os dois construtores da classe exigem o número) e nunca pode ser alterado, criamos apenas o método *getNumero()*. No caso do saldo, como ele só pode ser alterado por saques e depósitos, não faria sentido criar um método *setSaldo*. Assim, os métodos *depositar* e *sacar* servem para alterar o saldo e o *getSaldo()* nos retorna o valor atual do saldo.

Vale ressaltar que o qualificador *private* também pode ser usado para modificar o acesso a um método quando este existe apenas para auxiliar a própria classe e não queremos que outras pessoas o utilizem.

Entenderemos melhor o uso do qualificador *protected* na próxima aula, quando abordarmos os conceitos de herança e de hierarquias de classes.

Resumo

Nesta aula aprendemos dois conceitos importantíssimos de programação orientada a objetos: construtores e encapsulamentos. Vimos que os construtores são utilizados sempre que um objeto é instanciado e que ao encapsular os atributos de uma classe conseguimos esconder a complexidade de um objeto pela definição de métodos. Com a utilização desses conceitos, criaremos sistemas mais modulares e com maior manutenibilidade e segurança.

Atividades de aprendizagem

1. Partindo do código da classe *Conta* que você criou no exercício 1 da aula 2, crie dois construtores como os apresentados na Figura 3.1 e encapsule os atributos como na Figura 3.3.
2. Agora altere o código do programa que você desenvolveu no exercício 3 da aula 2, de forma que ele utilize a nova classe *Conta* gerada no exercício anterior.

Aula 4 – Herança e polimorfismo

Objetivos

Conhecer o conceito de herança e aprender a implementar esse conceito em Java.

Compreender o conceito de poliformismo.

Conhecer os conceitos de sobrecarga e sobrescrita de métodos.

Aprender a utilizar poliformismo pela aplicação dos conceitos de herança, sobrecarga e sobrescrita de métodos.

4.1 Herança

Uma das vantagens das linguagens orientadas a objeto é a possibilidade de se reutilizar código. Mas, quando falamos em reutilização de código, precisamos de recursos muito mais poderosos do que simplesmente copiar e alterar o código.

Um dos conceitos de orientação a objetos que possibilita a reutilização de código é o conceito de **herança**. Pelo conceito de herança é possível criar uma nova classe a partir de outra classe já existente.

Para ilustrar o conceito de herança vamos criar uma classe para representar as contas especiais de um banco. Em nosso exemplo, uma conta especial é um tipo de conta que permite que o cliente efetue saques acima de seu saldo até um limite, ou seja, permite que o cliente fique com saldo negativo até um dado limite. Assim, criaremos uma classe *ContaEspecial* que herdará da classe *Conta* que criamos em aulas anteriores.

Adotaremos essa estratégia, já que uma *ContaEspecial* é um tipo de *Conta* que tem, além de todos os atributos comuns a todas as contas, o atributo *limite*. Sendo assim, deve-se utilizar a palavra reservada **extends** para que *ContaEspecial* herde de *Conta* suas características. A Figura 4.1 exibe o código da classe da classe *ContaEspecial*. Note na definição da classe a utilização da palavra *extends*.

A-Z

Herança

É um mecanismo que permite que uma classe herde todo o comportamento e os atributos de outra classe (CADENHEAD; LEMAY, 2005, p. 13)

Em uma herança, a classe da qual outras classes herdam é chamada de classe pai, classe base ou superclasse. Já a classe que herda de uma outra é chamada de classe filha, classe derivada ou subclasse.

```

public class ContaEspecial extends Conta {
    private double limite;

    public double getLimite() {
        return limite;
    }

    public void setLimite(double limite) {
        this.limite = limite;
    }
}

```

Figura 4.1: Código da classe *ContaEspecial* que herda da classe *Conta*

Fonte: Elaborada pelos autores

Nesse caso dizemos que *ContaEspecial* é uma **subclasse** ou **classe filha** de *Conta*. Podemos também dizer que *Conta* é **ancestral** ou **classe pai** de *ContaEspecial*. Note que *ContaEspecial* define um tipo mais especializado de conta. Assim, ao mecanismo de criar novas classes herdando de outras é dado o nome de **especialização**.

Agora suponha que tenhamos um outro tipo de conta: a *ContaPoupanca*. A *ContaPoupanca* tem tudo o que a *Conta* tem com um método a mais que permite atribuir um reajuste percentual ao saldo. Agora teríamos duas classes herdando da classe *Conta*. Nesse contexto podemos dizer que a classe *Conta* **generaliza** os conceitos de *ContaEspecial* e *ContaPoupanca*.

A Figura 4.2 exibe o código da classe *ContaPoupanca*. Note que ela herda as características da classe *Conta* e apenas implementa um novo método: *reajustar*.

```

public class ContaPoupanca extends Conta{
    public void reajustar(double percentual){
        double saldoAtual = this.getSaldo();//Obtenho o saldo atual da conta
        double reajuste = saldoAtual * percentual;//Calculo o reajuste
        this.depositar(reajuste);//deposito o reajuste na conta;
    }
}

```

Figura 4.2: Código da classe *ContaPoupanca* que herda da classe *Conta*

Fonte: Elaborada pelos autores



Quando visualizamos uma hierarquia partindo da classe pai para filhas, dizemos que houve uma **especialização** da superclasse. Quando visualizamos partindo das classes filhas para as classes ancestrais, dizemos que houve uma **generalização** das **subclasses**.

Após definir as classes *ContaPoupanca* e *ContaEspecial*, conforme ilustrado nas Figuras 4.1 e 4.2, o NetBeans nos sinalizará com erros nas duas classes pois, na aula anterior definimos construtores para a classe *Conta* que é ancestral das duas que criamos agora e não definimos construtores para as duas subclasses que acabamos de gerar.

Caso você não tenha definido um construtor em sua superclasse, não será obrigado a definir construtores para as subclasses, pois Java utilizará o construtor padrão para a superclasse e para as subclasses. Porém, caso haja algum construtor definido na superclasse, obrigatoriamente você precisará criar ao menos um construtor para cada subclasse. Vale ressaltar que os construtores das subclasses utilizarão os construtores das superclasses pelo uso da palavra reservada *super*.

A Figura 4.3 exibe o construtor criado para a classe *ContaEspecial* enquanto a Figura 4.4 exibe o construtor da classe *ContaPoupanca*.

```
public ContaEspecial(int numero, String nome_titular, double limite) {  
    super(numero, nome_titular);  
    this.limite = limite;  
}
```

Figura 4.3: Construtor da classe *ContaEspecial*

Fonte: Elaborada pelos autores

```
public ContaPoupanca(int numero, String nome_titular){  
    super(numero, nome_titular);  
}
```

Figura 4.4: Construtor da classe *ContaPoupanca*

Fonte: Elaborada pelos autores

Note que o construtor da classe *ContaEspecial* recebe como parâmetros o número, o nome do titular e o limite. Então, pelo uso da palavra *super*, esse construtor “chama” o construtor da classe *Conta* (que criamos em aula anterior) repassando o número e o nome do titular e, depois, atribui ao *limite* o valor recebido como parâmetro. Já o construtor da classe *ContaPoupanca* apenas se utiliza do construtor da superclasse, pois ele não recebe atributos além dos já tratados pela superclasse.

4.2 Utilização de atributos *protected*

Quando estudamos **encapsulamento** aprendemos que devemos preferencialmente manter os atributos com nível de acesso privado (*private*) de forma que para acessá-los outras classes precisem utilizar métodos.

Mas, vimos também que há um nível de acesso protegido (*protected*) que faz com que o atributo se comporte como público para classes da mesma hierarquia ou do mesmo pacote e como privado para as demais classes. Definir alguns atributos da superclasse como *protected* pode trazer algumas facilidades para implementar métodos das subclasses.

Por exemplo, note que na implementação do método *reajustar* da classe *ContaPoupanca* apresentado na Figura 4.2, tivemos de obter o saldo, calcular o reajuste e depois depositar esse reajuste na conta para que fosse somado ao saldo, pois o atributo *saldo* foi definido na superclasse *Conta* como privado, não permitindo que o alterássemos diretamente de dentro da classe *ContaPoupanca*.

A Figura 4.5 apresenta uma nova implementação para o método *reajustar* da classe *ContaPoupanca*, considerando que o atributo *saldo* teve sua definição alterada para protegido (*protected*).

```
public class ContaPoupanca extends Conta {  
  
    public void reajustar(double percentual) {  
        //Recalculo o saldo acessando diretamente o atributo  
        saldo = saldo + saldo * percentual;  
    }  
}
```

Figura 4.5: Nova implementação do método *reajustar* considerando o atributo *saldo* como *protected*

Fonte: Elaborada pelos autores

Note que na nova implementação podemos acessar diretamente o atributo, o que a torna mais simples.



Apesar de potencialmente facilitar a implementação de métodos nas subclasses, a utilização de atributos protegidos é perigosa, pois o atributo ficará acessível a todas as classes que estejam no mesmo pacote e não somente às subclasses. Assim, pense bastante sobre as vantagens e desvantagens antes de se decidir por definir um atributo como *protected*.

4.3 Polimorfismo

A palavra polimorfismo vem do grego *poli morfos* e significa muitas formas. Na orientação a objetos, isso representa uma característica que permite que classes diferentes sejam tratadas de uma mesma forma.

O polimorfismo permite escrever programas que processam objetos que compartilham a mesma superclasse (direta ou indiretamente) como se todos fossem objetos da superclasse; isso pode simplificar a programação (DEITEL; DEITEL, 2010, p. 305).

Em outras palavras, podemos ver o polimorfismo como a possibilidade de um mesmo método ser executado de forma diferente de acordo com a classe do objeto que aciona o método e com os parâmetros passados para o método.

Com o polimorfismo podemos projetar e implementar sistemas que são facilmente extensíveis – novas classes podem ser adicionadas com pouca ou nenhuma alteração a partes gerais do programa, contanto que as novas classes façam parte da hierarquia de herança que o programa processa genericamente. As únicas partes de um programa que devem ser alteradas para acomodar as novas classes são aquelas que exigem conhecimento direto das novas classes que adicionamos à hierarquia (DEITEL; DEITEL, 2010, p. 305).

O polimorfismo pode ser obtido pela utilização dos conceitos de herança, sobrecarga de métodos e sobrescrita de método (também conhecida como redefinição ou reescrita de método).

4.4 Sobrescrita

A técnica de sobrescrita permite reescrever um método em uma subclasse de forma que tenha comportamento diferente do método de mesma assinatura existente na sua superclasse.

Para ilustrar o conceito de sobrescrita, vamos criar um método *imprimirTipoConta()* na superclasse *Conta* e vamos sobrescrevê-lo nas duas subclasses da nossa hierarquia de exemplo (*ContaEspecial* e *ContaPoupanca*). Esse método simplesmente imprimirá na tela uma mensagem de acordo com o tipo da conta, ou seja, de acordo com o tipo do objeto ele imprimirá uma mensagem diferente. A Figura 4.6 exibe apenas a linha de definição de cada classe e seu respectivo método *imprimirTipoConta()* (omitimos o resto dos códigos das classes e as juntamos todas em uma única figura por uma questão de espaço).

```

public class Conta {

    public void imprimirTipoConta(){
        System.out.println("Conta Comum");
    }
}
public class ContaEspecial extends Conta {

    @Override
    public void imprimirTipoConta(){
        System.out.println("Conta Especial");
    }
}
public class ContaPoupanca extends Conta {

    @Override
    public void imprimirTipoConta() {
        System.out.println("Conta Poupança");
    }
}

```

Figura 4.6: Exemplo de sobrescrita de método

Fonte: Elaborada pelos autores



Note que no exemplo da Figura 4.6, nas linhas anteriores aos métodos *ImprimirTipoConta* das classes *ContaEspecial* e *ContaPoupança*, há uma notação **@Override**. A notação **@Override** é inserida automaticamente pelo NetBeans para indicar que esse método foi definido no ancestral e está sendo redefinido na classe atual. A não colocação da notação **@Override** não gera erro, mas gera um aviso (*Warning*). Isso ocorre porque entende-se que, quando lemos uma classe e seus métodos, é importante existir alguma forma de sabermos se um certo método foi ou não definido numa classe ancestral. Assim a notação **@Override** é fundamental para aumentar a legibilidade e manutenibilidade do código.

A Figura 4.7 exibe um exemplo de utilização dos métodos apresentados na Figura 4.6, a fim de ilustrar o polimorfismo.

```

package banco;
import java.util.Scanner;

public class UsaContaPolimorfa {
    public static void main(String[] args) {
        Conta c = null;
        Scanner scan = new Scanner(System.in);
        int opcao;
        System.out.println("Qual tipo de conta deseja criar para José?");
        System.out.println("1 - Conta");
        System.out.println("2 - Conta especial");
        System.out.println("3 - Conta poupança");
        opcao = scan.nextInt();
        switch (opcao) {
            case 1:
                c = new Conta(1, "José");
                break;
            case 2:
                c = new ContaEspecial(1, "José", 100.00);
                break;
            case 3:
                c = new ContaPoupanca(1, "José");
                break;
        }
        c.imprimirTipoConta();
    }
}

```

Figura 4.7: Exemplo de polimorfismo

Fonte: Elaborada pelos autores

No exemplo da Figura 4.7, vemos que, de acordo com a opção escolhida no menu impresso, temos a criação de um objeto diferente. Quando o usuário digita 2, por exemplo, é criada uma instância de *ContaEspecial*. Note que a variável *c* é do tipo *Conta*. Mas, ainda assim, é possível atribuir a ela uma instância de *ContaPoupanca* ou de *ContaEspecial* pois ambas herdam da classe *Conta*.

Na última linha do código apresentado, o método *imprimirTipoConta()* é chamado. Mas, sabemos que esse método foi implementado na classe *Conta* e sobrescrito nas duas classes filhas. Assim, qual das implementações será usada por essa chamada? A resposta a essa pergunta vai depender da opção digitada pelo usuário! Por exemplo, caso o usuário digite 2, a variável *c* receberá uma instância de *ContaEspecial*. Nesse caso, na última linha será chamado o método *imprimirTipoConta()* da classe *ContaEspecial*. Analogamente, caso o usuário digite a opção 1, será utilizado o método da classe *Conta* e, caso digite 3, será utilizado o método da classe *ContaPoupanca*.

Nesse exemplo, a mesma linha de código pode ter um comportamento diferente, dependendo das circunstâncias. Isso é polimorfismo!



Agora que entendemos o conceito de sobrescrita, vamos corrigir uma falha que cometemos ao definir nossa hierarquia de classes! A classe *ContaEspecial* tem um atributo *limite* que define um valor que o proprietário da conta poderia sacar mesmo não tendo saldo. O problema é que quando implementamos a classe *ContaEspecial* não reescrevemos o método *sacar*; logo, essa classe está utilizando o método da superclasse *Conta*. Assim, independentemente do valor do atributo *limite* da *ContaEspecial*, o saque não será efetuado caso não haja saldo suficiente, pois essa é a lógica implementada no método *sacar* da classe *Conta*. A Figura 4.8 exibe uma implementação para o método *sacar* na classe *ContaEspecial* que sobrescreve o método da superclasse e permite a realização do saque caso o valor a ser sacado seja menor ou igual a soma entre o saldo e o limite da conta.

```
@Override
public boolean sacar(double valor) {
    if (valor <= this.limite + this.saldo) {
        this.saldo -= valor;
        return true;
    } else {
        return false;
    }
}
```

Figura 4.8: Método *sacar* sobrescrito na classe *ContaEspecial*

Fonte: Elaborada pelos autores

4.5 Sobrecarga



Métodos de mesmo nome podem ser declarados na mesma classe, contanto que tenham diferentes conjuntos de parâmetros (determinado pelo número, tipos e ordem dos parâmetros). Isso é chamado sobrecarga de método (DEITEL; DEITEL, 2010, p. 174).

Para que os métodos de mesmo nome possam ser distinguidos, eles devem possuir **assinaturas** diferentes. A assinatura (*signature*) de um método é composta pelo nome do método e por uma lista que indica os tipos de todos os seus argumentos. Assim, métodos com mesmo nome são considerados diferentes se recebem um diferente número de argumentos ou tipos diferentes de argumentos e têm, portanto, uma assinatura diferente.

Quando um método sobrecarregado é chamado, o compilador Java seleciona o método adequado examinando o número, os tipos e a ordem dos argumentos na chamada (DEITEL; DEITEL, 2010, p. 174).

Mesmo sem saber, nós já utilizamos o conceito de sobrecarga quando criamos os construtores da classe *Conta*. Ao criar a classe *Conta* nós definimos dois construtores, sendo que um deles recebendo três parâmetros e o outro recebendo dois parâmetros (veja a Figura 4.1 da Aula 4).

Para ilustrar melhor o conceito de sobrecarga, implementaremos na classe *Conta* um novo método *imprimirTipoConta* que receberá como parâmetro uma *String* e imprimirá na tela o tipo da conta seguido pela *String* recebida. A Figura 4.9 exibe os dois métodos *imprimirTipoConta* da classe *Conta*.

```
public void imprimirTipoConta() {
    System.out.println("Conta Comum");
}
public void imprimirTipoConta(String s) {
    System.out.println("Conta Comum - String recebida:" + s);
}
```

Figura 4.9: Exemplo de sobrecarga

Fonte: Elaborada pelos autores

Na Figura 4.10 é apresentado um exemplo que visa ilustrar o comportamento dos objetos diante o uso de herança, sobrecarga e sobrescrita. A figura exibe tanto o código-fonte de uma classe que utiliza as classes *Conta* e *ContaEspecial* quanto a saída da execução desse programa (destacada em vermelho).

```
1 package banco;
2
3 public class UsaSobrecargaSobrescrita {
4
5     public static void main(String[] args) {
6         Conta c1 = new Conta(1, "Ze");
7         ContaEspecial c2 = new ContaEspecial(2, "João", 100);
8         c1.imprimirTipoConta();
9         c1.imprimirTipoConta("Teste Sobrecarga");
10        c2.imprimirTipoConta();
11        c2.imprimirTipoConta("Teste Sobrecarga em subclasse");
12    }
13 }
14
```

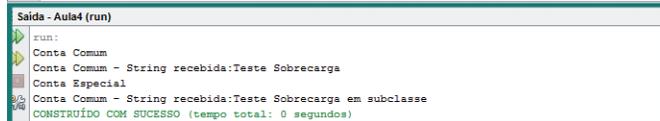


Figura 4.10: Utilização de sobrecarga e sobrescrita

Fonte: NetBeans IDE 7.0.1

No exemplo da Figura 4.10, a variável *c1* contém uma instância da classe *Conta* e *c2*, uma instância da classe *ContaEspecial*. Após a criação dos objetos, primeiro é feita uma chamada ao método *imprimirTipoConta* de *c1* sem passar nenhum parâmetro, e depois outra chamada passando uma *String*. Em seguida foram feitas as mesmas chamadas a partir de *c2*. Note que como não criamos na classe *ContaEspecial*, um método *imprimirTipoConta* que

receba uma *String* como argumento, a última chamada feita a partir de *c2* foi atendida pelo método da classe *Conta*. O que aconteceu foi que, como *c2* contém uma instância de *ContaEspecial*, o compilador procurou por um método *imprimirTipoConta(String)* na classe *ContaEspecial*. Como não encontrou, então o compilador fez uso do método existente na superclasse, já que *ContaEspecial* herda de *Conta*.

4.6 Classe *Object*

Todas as classes no Java herdam direta ou indiretamente da classe *Object*; portanto, seus 11 métodos são herdados por todas as outras classes (DEITEL; DEITEL, 2010, p. 258).

Vejamos alguns métodos:

toString(): esse método indica como transformar um objeto de uma classe em uma *String*. Ele é utilizado, automaticamente, sempre que é necessário transformar um objeto de uma classe em uma *String*. Na classe *Conta*, por exemplo, esse método poderia ser definido da seguinte forma:

```
@Override  
  
public String toString() {  
  
    return ("Conta: " + this.numero);  
  
}
```



Note o uso da notação **@Override** para o método ***toString()***. Isso ocorre porque estamos sobrescrevendo um **método definido em *Object***.

getClass(): retorna a classe de um objeto. Muito utilizado quando se trabalha na criação de ferramentas geradoras de código ou *frameworks*. Utilizaremos esse método no exemplo da Figura 4.11 para construir o método *toString* .

equals(): esse método possibilita comparar os valores de dois objetos. Se considerarmos esses objetos iguais, devemos retornar *true*; caso sejam diferentes, devemos retornar *false*.

Quando comparamos dois objetos com o operador `==`, na realidade estamos comparando se eles são o mesmo objeto e não se seus valores são iguais. Isso ocorre porque os objetos em Java são ponteiros para espaços de memória. Assim, dois objetos podem ter os mesmos valores em seus atributos e não serem iguais, pois podem apontar para locais diferentes. Dessa forma, para comparar os valores de dois objetos, devemos utilizar o método *equals*.

Por isso que quando queremos comparar *Strings*, por exemplo, utilizamos o método *equals*. A Figura 4.11 exibe uma implementação do método *equals* para a classe *Conta*. Nesse método consideramos que duas contas são iguais se são de uma mesma classe e se têm o mesmo número.

```
@Override
public boolean equals(Object o) {
    if (o == null) {
        return false;
    } else if (o.getClass() != this.getClass()) {
        return false;

    } else if (((Conta) o).getNumero() != this.getNumero()) {
        return false;

    } else {
        return true;
    }
}
```

Figura 4.11; Exemplo de método *equals* para a classe *Conta*

Fonte: Elaborada pelos autores

No método *equals* apresentado são feitas as seguintes verificações:

- *if (o == null)*: estamos prevendo que se pode tentar comparar um objeto *Conta* com um valor nulo (variável não instanciada). Como o objeto *Conta* que acionou o método *equals* está instanciado, ele não pode ser igual a *null*.
- *if (o.getClass() != this.getClass())*: estamos verificando se o objeto passado como parâmetro é da mesma classe que o objeto que está invocando o método, ou seja, se estamos comparando duas instâncias da classe *Conta*. Caso os objetos sejam de classes diferentes, consideramos que eles são diferentes.
- *if (((Conta) o).getNumero() != this.getNumero())*: caso os dois objetos sejam do mesmo tipo (*Conta*), então comparamos os valores do atributo *numero* dessas contas. Se os números são diferentes, consideramos que são contas diferentes; caso contrário, as consideramos iguais.



É importante conhecermos a hierarquia de uma classe para evitarmos replicar códigos de forma desnecessária.



Acesse a URL <http://download.oracle.com/javase/6/docs/api/java/lang/Object.html>. Lá você encontrará a documentação oficial da Oracle sobre a classe *Object*.

Conforme você verá na documentação acima, a classe *Object* provê o método *clone*, cujo objetivo é criar uma cópia de um objeto. Nesse contexto, explique por que a expressão *x.clone().equals(x)* é tipicamente verdadeira, enquanto a expressão *x.clone() == x* é tipicamente falsa.

Resumo

Nesta aula aprendemos dois dos principais conceitos de orientação a objetos: herança e polimorfismo. O conceito de herança nos permite criar uma classe a partir de outra. Assim, quando temos um conjunto de classes com características comuns, utilizamos o conceito de herança para agrupar essas características em vez de repetirmos suas implementações várias vezes. Já o polimorfismo permite que um mesmo método seja executado de formas diferentes de acordo com a classe do objeto que o aciona e com os parâmetros passados para o método. De fato, o polimorfismo é conseguido pela implementação de conceitos como herança, sobrecarga e sobrescrita de métodos. Esses conceitos nos permitem desenvolver códigos mais reutilizáveis e elegantes. O entendimento desses conceitos é fundamental para desenvolver programas realmente orientados a objetos.

Atividades de aprendizagem

1. Crie a hierarquia de classes utilizada como exemplo nesta aula: crie a classe *ContaEspecial* (Figura 4.1) e *ContaPoupanca* (Figura 4.2), crie os construtores dessas classes (Figuras 4.3 e 4.4), crie o método *reajustar* na classe *ContaPoupanca* (Figura 4.5) e o método *sacar* na classe *ContaEspecial* (Figura 4.8).
2. Agora que temos vários tipos de contas, vamos alterar o programa que iniciamos no exercício 3 da aula 2 e incrementamos no exercício 2 da aula 3. Nessa nova versão, quando o usuário escolher a opção 1 (criar nova Conta) o sistema deve questionar o tipo de conta a ser criada dando como opções 1 para *Conta*, 2 para *ContaEspecial* e 3 para *ContaPoupanca*. Com base na resposta do usuário, o sistema deve instanciar o tipo correto de conta (lembre-se que os argumentos necessários para instanciar um objeto variam de acordo com o tipo da conta). Após criada, a conta continua sendo inserida na lista. Não são necessárias alterações no resto do código graças ao polimorfismo!

Note que no exercício 2 da aula 4 o único trecho de código que será alterado devido ao fato de termos contas de tipos diferentes é o trecho que trata de criação de contas. Todo o resto continua funcionando, pois, independentemente do tipo, toda conta tem métodos para sacar, depositar e exibir saldo. Note também que, de acordo com o tipo da conta, o método correto é chamado de forma automática. É o polimorfismo funcionando na prática!

Aula 5 – Classes abstratas e associações

Objetivos

Conhecer os conceitos de classes e métodos abstratos.

Aprender a utilizar classes abstratas em hierarquias de classes.

Conhecer o conceito de associação entre classes.

Aprender a implementar associações entre classes em Java.

5.1 Classes abstratas

Já aprendemos que uma **classe** define as características e o comportamento de um conjunto de objetos. Assim, os objetos são criados (instanciados) a partir de classes.

Mas, nem todas as classes são projetadas para permitir a criação de objetos. Algumas classes são usadas apenas para agrupar características comuns a diversas classes e, então, ser herdada por outras classes. Tais classes são conhecidas como classes abstratas.

Às vezes é útil declarar classes – chamadas classes abstratas – para as quais você nunca pretende criar objetos. Como elas só são usadas como superclasses em hierarquias de herança, são chamadas superclasses abstratas. Essas classes não podem ser usadas para instanciar objetos, porque são incompletas. Suas subclasses devem declarar as “partes ausentes” para tornarem-se classes concretas, a partir das quais você pode instanciar objetos (DEITEL; DEITEL, 2010, p. 309).



As classes que não são abstratas são conhecidas como classes concretas. As classes concretas podem ter instâncias diretas, ao contrário das classes abstratas que só podem ter instâncias indiretas, ou seja, apesar de a classe abstrata não poder ser instanciada, ela deve ter subclasses concretas que por sua vez podem ser instanciadas.



O conceito de classe abstrata está intimamente ligado ao conceito de herança estudado na aula anterior. Assim, caso você ainda tenha alguma dúvida sobre a aula anterior, tente esclarecê-la antes de continuar.

Para ilustrar o conceito de classe abstrata, voltemos ao nosso exemplo de contas bancárias. No exemplo apresentado na aula sobre herança e polimorfismo, tínhamos as classes *ContaEspecial* e *ContaPoupança* herdadas da classe *Conta*.

Agora, suponha que toda conta criada no nosso banco tenha que ser uma conta especial ou uma conta poupança. Nesse caso, nunca teríamos uma instância da classe *Conta*, pois toda conta criada seria uma instância de *ContaEspecial* ou de *ContaPoupança*.

Nesse contexto surgem algumas perguntas: teria sentido criar a classe *Conta*? Por que criar uma classe que nunca será instanciada?

A resposta à primeira pergunta é sim! A classe *Conta* continuaria existindo para organizar as características comuns aos dois tipos de contas. Então, para garantir que a classe *Conta* exista, mas nunca seja instanciada, essa classe deve ser criada como abstrata.

Para definir uma classe abstrata em Java, basta utilizar a palavra reservada *abstract*. A palavra *abstract* deve ser inserida entre o qualificador de acesso e o nome da classe. A Figura 5.1 exibe o código da classe *Conta* agora como classe abstrata.

```
package banco;

public abstract class Conta {

    private int numero;
    private String nome_titular;
    protected double saldo;

    public Conta(int numero, String nome_titular, double saldo) {
        this.numero = numero;
        this.nome_titular = nome_titular;
        this.saldo = saldo;
    }

    public Conta(int numero, String nome_titular) {
        this.numero = numero;
        this.nome_titular = nome_titular;
        saldo = 0;
    }
}
```

Figura 5.1: Versão abstrata da classe *Conta*

Fonte: Elaborada pelos autores

Note que a única diferença em relação ao código original é o uso da palavra *abstract*. A Figura 5.1 apresenta apenas os atributos e construtores da classe *Conta*, não apresentando os seus métodos. Mas, vale ressaltar que a transformação de uma classe em abstrata não traz impacto para nenhum de seus métodos e nem para os códigos das suas subclasses.

Observando o exemplo da Figura 5.1, surge uma nova questão: tendo em vista que uma classe abstrata não pode ser instanciada, faz algum sentido uma classe abstrata ter construtores?

A resposta é sim! Como vimos na aula sobre herança, os construtores das subclasses se utilizam dos construtores da superclasse. Assim, mesmo não podendo ser instanciadas, é comum classes abstratas terem construtores que inicializam seus próprios atributos e são utilizados pelas subclasses. Mas, vale ressaltar que, assim como em qualquer outra classe, não é obrigatório definir construtores para as classes abstratas.

Mas, e se eu tentar utilizar um construtor de uma classe abstrata para instanciar um objeto, o que acontece? A resposta é **erro de compilação**. A Figura 5.2 exibe um exemplo de classe que tenta criar uma instância de *Conta*, uma instância de *ContaEspecial* e uma instância de *ContaPoupanca*.

```
1 package banco;
2
3 public class UsaClasseAbstrata {
4
5     public static void main(String[] args) {
6         Conta c1 = new Conta(1, "Ze");//ERRO!
7         ContaEspecial c2 = new ContaEspecial(2, "João", 100);
8         c1 = new ContaPoupanca(1, "Ze");
9     }
```

Figura 5.2: Tentativa de instanciar uma classe abstrata

Fonte: Elaborada pelos autores

Note que na linha 6 do código exibido na Figura 5.2 tentamos instanciar um objeto da classe *Conta*, mas o NetBeans apontou o erro. Já na linha 7, criamos normalmente uma instância de *ContaEspecial*. Por fim, preste atenção à linha 8. Apesar de *c1* ser uma variável do tipo *Conta* que é abstrato, foi possível atribuir a ela uma instância de classe *ContaPoupanca*, pois *ContaPoupanca* é uma classe concreta e herda de *Conta*.

Métodos abstratos

São aqueles que não possuem código e devem ser implementados em classes concretas que herdem da classe abstrata (não abstratas).

5.2 Métodos abstratos

Em algumas situações as classes abstratas podem ser utilizadas para prover a definição de métodos que devem ser implementados em todas as suas subclasses, sem apresentar uma implementação para esses métodos. Tais métodos são chamados de **métodos abstratos**.



Toda classe que possui pelo menos um método abstrato é uma classe abstrata, mas uma classe pode ser abstrata sem possuir nenhum método abstrato.

Para definir um método abstrato em Java, utiliza-se a palavra reservada `abstract` entre o especificador de visibilidade e o tipo de retorno do método. Vale ressaltar que um método abstrato não tem corpo, ou seja, apresenta apenas uma assinatura.

Para exemplificar o conceito de método abstrato, vamos novamente recorrer ao exemplo das contas bancárias. É sensato imaginar que todo tipo de conta bancária deve ter uma forma de sacar. Mas, de acordo com o tipo da conta, há regras diferentes para o saque. Em nosso exemplo, a *ContaEspecial* possui um limite de forma que ela permite saques acima do saldo disponível até o limite da conta. Já a *ContaPoupanca* não permite saques acima do saldo disponível.

Nesse contexto, podemos definir, na classe *Conta*, um método abstrato *sacar*. Assim, toda classe que herdar de *Conta* deverá sobrescrever (lembra-se do conceito de sobrescrita?) esse método implementando suas próprias regras de saque.

A Figura 5.3 exibe o código do método abstrato *sacar* definido na classe *Conta*.

```
public abstract boolean sacar(double valor);
```

Figura 5.3: Método abstrato *sacar* da classe *Conta*

Fonte: Elaborada pelos autores

Note que o método apresentado na Figura 5.3 não possui corpo, ou seja, ele não define como deve ser feito um saque em uma conta. Esse método serve apenas para obrigar todas as classes que herdem de *Conta* a sobrescrever o método *sacar*. Caso uma subclasse de *Conta* não sobrescreva o método *sacar*, ela não poderá ser definida como uma classe concreta.

A Figura 5.4 exibe as implementações dos métodos *sacar* nas classes *ContaPoupanca* e *ContaEspecial*.

```

public class ContaEspecial extends Conta {
    private double limite;

    @Override
    public boolean sacar(double valor){
        if (valor <= this.limite + this.saldo) {
            this.saldo -= valor;
            return true;
        } else {
            return false;
        }
    }
}

public class ContaPoupanca extends Conta {

    @Override
    public boolean sacar(double valor) {
        if (this.getSaldo() >= valor) {
            this.saldo -= valor;
            return true;
        } else {
            return false;
        }
    }
}

```

Figura 5.4: Implementações concretas do método *sacar* nas classes *ContaEspecial* e *ContaPoupanca*

Fonte: Elaborada pelos autores

Para que uma subclasse de uma classe abstrata seja concreta, ela deve obrigatoriamente apresentar implementações concretas para todos os métodos abstratos de sua superclasse. Por exemplo, se o método *sacar* não fosse implementado na classe *ContaEspecial*, essa classe teria de ser abstrata ou ocorreria erro de compilação.



5.3 Associações

Como sabemos, a base do paradigma de orientação a objetos é que o mundo pode ser visto como um conjunto de objetos e, objetos com estruturas similares podem ser agrupados em classes de objetos. Assim, as classes descrevem as estruturas (pelos atributos) e o comportamento (pelos métodos) de um conjunto de objetos.

É fácil notar que no mundo real esses objetos se ligam uns aos outros de diversas maneiras. Voltando ao exemplo do banco, podemos dizer que o banco tem **clientes** e esses clientes possuem **contas** bancárias. Note que os clientes constituem um conjunto de objetos desse universo bancário que possui um conjunto de características próprias como nome, CPF, entre outras. Ou seja, podemos criar uma classe *Cliente* para representar nossos clientes. Além disso, como clientes possuem contas bancárias, é necessário mapear as ligações que relacionam o cliente à sua conta. Essas ligações entre classes são chamadas de **associações**.



Associação
Representa uma conexão entre classes.

As associações podem ser implementadas como atributos e/ou como métodos. Por exemplo, vimos que a classe *Conta* poderia ser associada a classe *Cliente*. Assim, uma forma de implementar essa associação seria criar na classe *Conta* um atributo do tipo *Cliente* e métodos que nos permitam obter o cliente titular da conta e atribuir um novo cliente à conta.



Utilizamos associações quando desejamos representar relacionamentos que existam entre objetos de duas classes distintas, como por exemplo:

- Um *Curso* tem *Disciplinas*
- Um *Produto* é de uma *Marca*.



Veja que no método *SetCPF* apenas recebemos uma *String* e atribuímos ao CPF. Na verdade, o ideal seria validar se a *String* passada é um CPF válido e só atribuir em caso positivo. Nesse caso o método retornaria um *boolean* indicando se a operação foi realizada ou não. Como vimos na Aula 4, esse é o maior motivo para encapsular atributos: validar os valores atribuídos.

No exemplo que criamos anteriormente, a classe *Conta* tinha três atributos: o número, o saldo e o nome do titular. Note que número e saldo são realmente características da conta. Porém o *nome_titular* é na verdade uma característica do cliente. Assim, agora que criamos uma classe *Cliente* para encapsular as características do cliente, tiraremos o atributo *nome_titular* da classe *Conta* e colocaremos em seu lugar um atributo *titular* que será do tipo *Cliente*. Esse atributo representará a associação entre a classe *Conta* e a classe *Cliente*. Veja na Figura 5.5 o código da classe *Cliente* e na Figura 5.6 o novo código da classe *Conta* implementando a associação com a classe *Cliente* (a Figura 5.6 exibe apenas a parte do código da classe *Conta* que foi alterada devido à associação com a classe *Cliente*).

```
package banco;

public class Cliente {
    private String nome;
    private String cpf;

    public Cliente(String nome, String cpf) {
        this.setNome(nome);
        this.setCpf(cpf);
    }

    public String getNome() {
        return nome;
    }

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

    public String getCpf() {
        return cpf;
    }

    public void setCpf(String cpf) {
        //Faltou Verificar se o cpf é válido antes de atribuir...
        this.cpf = cpf;
    }
}
```

Figura 5.5: Classe *Cliente*

Fonte: Elaborada pelos autores

```

public abstract class Conta {

    private int numero;
    private Cliente titular;
    protected double saldo;

    public Conta(int numero, Cliente titular, double saldo) {
        this.numero = numero;
        this.titular = titular;
        this.saldo = saldo;
    }

    public Conta(int numero, Cliente titular) {
        this.numero = numero;
        this.titular = titular;
        saldo = 0;
    }

    public Conta(int numero, String nomeTitular, String cpfTitular) {
        this.numero = numero;
        this.titular = new Cliente(nomeTitular, cpfTitular);
        saldo = 0;
    }

    public Cliente getTitular() {
        return titular;
    }

    public void setTitular(Cliente titular) {
        this.titular = titular;
    }
}

```

Figura 5.6: Classe Conta

Fonte: Elaborada pelos autores

Note que na classe *Conta* os atributos *numero* e *saldo* são representados por tipos primitivos enquanto o atributo *titular* é representado por outro objeto, ou seja, é um atributo cujo tipo é uma outra classe (no caso, a classe *Cliente*). Isso significa que um objeto pode ser construído pela associação de outros objetos.

Na Figura 5.6 vemos que alguns construtores da classe *Conta* recebem como argumento um objeto da classe *Cliente*. Porém, é possível fazer a instanciação de um objeto de uma Classe no construtor de uma outra classe com a qual a primeira tenha uma associação. Por exemplo, note na Figura 5.6 que há um construtor da classe *Conta* que recebe como parâmetros os dados do *Cliente* e instancia um objeto *Cliente* dentro do seu construtor. Mas, deve-se tomar cuidado com essa abordagem quando se trabalha com sistemas grandes nos quais uma mesma classe tem associação com diversas outras.



Nas classes que utilizam associação, portanto, existem instanciações de objetos de uma dada classe ou recebimento de objetos construídos em outras partes do código.

A associação é um tipo de estruturação de classes que facilita o reuso de código. Em nosso exemplo, agora que temos a classe *Cliente* criada, se formos desenvolver um sistema para uma loja varejista, por exemplo, podemos reutilizar essa classe.



Não confunda o conceito de herança com o de associação. A herança é um tipo de estruturação entre classes que indica que um objeto “é” de um certo tipo, contendo algumas especializações. Uma *ContaEspecial*, por exemplo, é um tipo de *Conta*. Assim, para representar essa estruturação utilizamos herança. Já *Cliente* não é um tipo de *Conta* e nem *Conta* é um tipo de *Cliente*. Na verdade uma *Conta* pertence a um *Cliente*. Daí a utilização de associação para representar essa estruturação.

Resumo

Esta aula foi dedicada ao estudo de dois conceitos muito interessantes da orientação a objetos: classes e métodos abstratos. As classes abstratas não podem ser instanciadas e são utilizadas para agrupar um conjunto de características a serem herdadas por classes concretas. Os métodos abstratos são aqueles que não têm uma implementação, mas são obrigatoriamente implementados nas classes concretas que herdam da abstrata que a define. Toda classe que tem um método abstrato é abstrata, mas nem toda classe abstrata tem um método abstrato. A utilização desses conceitos possibilita trazer mais flexibilidade e qualidade para os programas orientados a objetos.

Atividades de aprendizagem

1. Com base na hierarquia de contas criada no exercício 1 da aula 4, transforme a classe *Conta* em uma classe abstrata (Figura 5.1). Transforme o método *sacar* da classe *Conta* em abstrato (Figura 5.3), implementando-o nas subclasses (Figuras 5.4 e 5.5).
2. Crie, no mesmo pacote, a classe *Cliente* (Figura 5.5) e altere a classe *Conta* para que tenha uma associação com a classe *Cliente* (Figura 5.6). Faça as alterações necessárias nos construtores das subclasses de *Conta*.
3. Altere o programa construído no exercício 2 da aula 4, para que continue funcionando, utilizando nossa nova hierarquia de *Contas* e a classe *Cliente*. Note que como *Conta* agora é abstrata, dentro da opção de criar nova conta não deve mais haver a opção de criar uma instância de *Conta*.

Aula 6 – Herança múltipla e interfaces

Objetivos

Entender o conceito de herança múltipla.

Conhecer o conceito de interface.

Aprender a aplicar o conceito de interface em Java.

Entender como o conceito de interface pode ser utilizado para simular uma herança múltipla em Java.

6.1 Herança múltipla

Imaginemos que precisamos fazer um sistema para a Federação Brasileira de Atletismo e, para isso, precisamos definir classes que representem Nadadores, Corredores, Ciclistas e Triatletas. Então, quando começamos a analisar o problema, notamos que esses indivíduos têm várias características em comum, como, por exemplo, todos eles devem se aquecer antes da prova. Assim, decidimos criar uma classe que represente todos os tipos de Atletas de forma que as demais classes herdarão dessa. Avaliando o problema mais a fundo, chegaremos à conclusão que todo Atleta é uma Pessoa, o que nos levaria a criar uma classe para organizar as características comuns a todas as pessoas.

Então, até o momento, decidimos criar uma superclasse *Pessoa* da qual herdará a classe *Atleta* que, por sua vez, tem como subclasses *Nadador*, *Corredor*, *Ciclista* e *Triatleta*. Mas, note que, um *Triatleta* deve saber correr como um *Corredor*, nadar como um *Nadador* e pedalar como um *Ciclista*. Assim, seria natural que *Triatleta* herdasse características dessas três classes.

Nesse contexto então teríamos a seguinte hierarquia:

- Teríamos uma superclasse *Pessoa* para agrupar as características comuns a todos as pessoas.
- Todo atleta é uma pessoa. Assim, *Atleta* herdaria de *Pessoa* (Em Java: *Atleta extends Pessoa*).

A-Z

Herança múltipla

O conceito de herança múltipla torna possível que uma classe descenda de várias classes.



Java não implementa herança múltipla por opção. Isso ocorre porque a herança múltipla pode nos gerar situações inusitadas.

Suponhamos que temos um método **aquecer()** em *Atleta* e esse método é redefinido em *Nadador*, *Corredor* e *Ciclista*. Suponhamos, ainda, que esse método não foi implementado em *Triatleta*. Dessa forma, o *Triatleta* deveria, por herança, utilizar o método **aquecer()** de seu ancestral.

Entretanto, *Triatleta* tem três ancestrais; aí vem a dúvida: qual dos três ele iria utilizar? Em outras palavras, um *Triatleta* aquece como um *Nadador*, como um *Corredor* ou como um *Ciclista*? Java, para evitar esse problema, eliminou a possibilidade de herança múltipla fazendo uso de interfaces. Outras linguagens utilizam outros tipos de soluções e permitem herança múltipla, mas acabam, de certa forma, entrando em conflito com o conceito de herança.

A-Z

Interface

É uma coleção de métodos que indica que uma classe possui algum comportamento além do que herda de suas superclasses.

Os métodos incluídos em uma interface não definem esse comportamento; essa tarefa é deixada para as classes que implementam a interface (CADENHEAD; LEMAY, 2005, p. 17).

- Todo nadador é um atleta. Assim, *Nadador* herdaria de *Atleta* (Em Java: *Nadador extends Atleta*).
- Todo corredor é um atleta. Assim, *Corredor* herdaria de *Atleta* (Em Java: *Corredor extends Atleta*).
- Todo ciclista é um atleta. Assim, *Ciclista* herdaria de *Atleta* (Em Java: *Ciclista extends Atleta*).
- Todo triatleta é nadador, corredor e ciclista. Assim *Triatleta* deveria herdar de *Nadador*, *Corredor* e *Atleta*.

No caso do *Triatleta* emprega-se o conceito de **herança múltipla**.

O problema é que Java não implementa o conceito de herança múltipla.

Mas, como implementar o comportamento da herança múltipla em Java, se Java não suporta herança múltipla? Isso é possível por meio de interfaces.

6.2 Interfaces

Uma **interface** pode ser vista como um conjunto de declarações de métodos, sem as respectivas implementações.

Uma interface é parecida com uma classe; porém, em uma interface, todos os métodos são públicos e abstratos e todos os atributos são públicos, estáticos e constantes.

A sintaxe para criar uma interface é muito parecida com a sintaxe para criar uma classe: `public interface <nome_da_interface>`.

O problema de herança múltipla apresentado na seção anterior pode ser resolvido com a criação de:

- quatro interfaces: *Atleta*, *Corredor*, *Nadador* e *Ciclista*;
- duas classes: *Pessoa* e *Triatleta*.

A Figura 6.1 exhibe as implementações das interfaces citadas. A implementação da classe *Pessoa* é exibida na Figura 6.2, enquanto a Figura 6.3 exhibe o código da classe *Triatleta*.

```

package interfaces;
public interface Atleta {
    public static final int i = 0;
    public abstract void aquecer();
}

package interfaces;
public interface Nadador extends Atleta {
    public void nadar();
}

package interfaces;
public interface Corredor extends Atleta {
    public void correr();
}

package interfaces;
public interface Ciclista extends Atleta {
    public void correrDeBicicleta();
}

```

Figura 6.1: Implementações das interfaces *Atleta*, *Nadador*, *Corredor* e *Ciclista*

Fonte: Elaborada pelos autores

Note que *Nadador*, *Corredor* e *Ciclista* herdam de *Atleta*. A herança entre interfaces é feita da mesma forma que a herança entre classes: utilizando a palavra reservada *extends*.

```

package interfaces;
public class Pessoa {
    private String nome, endereco;
    public Pessoa(String nome) {
        this.setNome(nome);
    }
    public String getEndereco() {
        return endereco;
    }
    public void setEndereco(String endereco) {
        this.endereco = endereco;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
}

```

Figura 6.2: Implementação da classe *Pessoa*

Fonte: Elaborada pelos autores

A classe *Pessoa* traz os atributos *nome* e *endereco*, os métodos que manipulam esses atributos e um construtor.

```

package interfaces;
public class Triatleta extends Pessoa implements Nadador, Corredor, Ciclista {
    public Triatleta(String nome) {
        super(nome);
    }
    public void aquecer() {
        System.out.println(this.getNome() + " está aquecendo");
    }
    public void nadar() {
        System.out.println(this.getNome() + " está nadando");
    }
    public void correr() {
        System.out.println(this.getNome() + " está correndo");
    }
    public void correrDeBicicleta() {
        System.out.println(this.getNome() + " está correndo de bike");
    }
}

```

Figura 6.3: Implementação da classe *Triatleta*

Fonte: Elaborada pelos autores

A classe *Triatleta* herda da classe *Pessoa* e implementa as interfaces *Nadador*, *Corredor* e *Ciclista*.

O construtor de *Triatleta* apenas chama o construtor de sua superclasse (*Pessoa*) repassando para ele o nome que recebera como parâmetro.

A classe *Triatleta* teve de implementar os métodos *nadar()*, *correr()* e *correrDeBicicleta()* porque ela implementa as interfaces *Nadador*, *Atleta* e *Ciclista*.

Note que também é obrigatória a implementação do método *aquecer()* pois, as interfaces *Nadador*, *Atleta* e *Ciclista* que são implementadas pela classe *Triatleta* herdam da interface *Atleta*.



O uso de interfaces é recomendável no desenvolvimento de sistemas para fornecer um contexto menos acoplado e mais simplificado de programação. Vamos supor, por exemplo, que temos uma interface responsável pela comunicação com banco de dados; dessa forma, qualquer classe que implementar a interface responderá a todas as funcionalidades para acesso a banco. Suponhamos que um novo banco seja elaborado e que desejemos fazer a troca do banco antigo por esse banco novo; será necessário apenas elaborar a classe que implemente a interface de acesso a esse banco novo e, ao invés de utilizarmos um objeto da classe antiga, utilizaremos um objeto da nova classe elaborada.

Resumo

O principal conceito trabalhado nesta aula foi o de herança múltipla. Herança múltipla é um tipo especial de herança na qual uma classe herda de duas ou mais outras classes. Mas, Java não implementa herança múltipla. Então o conceito de interface surge como uma alternativa. Uma interface define um conjunto de métodos sem suas implementações, de forma que toda classe que implemente uma interface deve trazer implementações para todos os métodos definidos em tal interface.

Atividades de aprendizagem

1. Implemente as interfaces *Nadador*, *Corredor* e *Ciclista* e as classes *Pessoa* e *Triatleta* como apresentadas nos exemplos deste capítulo. Faça um programa principal que instancie um triatleta e utilize seus métodos.
2. Crie uma interface *Investimento* que defina um método *reajustar* que receba um *double* como *parâmetro* e retorne *void*. Então altere a classe *Poupança* que criamos em exercícios anteriores de forma que ela implemente a interface *Investimento* e continue herdando da classe *Conta*, pois se trata de uma conta que também é um investimento.

Aula 7 – Interfaces gráficas em Java – Parte I

Objetivos

Construir nossas primeiras interfaces gráficas em Java.

Conhecer algumas classes para construção de interfaces gráficas em Java: `JFrame`, `JLabel`, `ImageIcon`, `JOptionPane`, `JTextField` e `JPasswordField`.

Aprender a fazer tratamento de eventos sobre interface gráfica em Java.

7.1 Java Swing

A construção de interfaces gráficas para programas *desktop* em Java se baseia em duas bibliotecas principais: a *AWT* (*Abstract Window Toolkit*) e a *Swing*.

Ao contrário do que muitos pensam, a *Swing* não é um substituto da *AWT*. Em vez disso, a *Swing* funciona como uma camada disposta acima da *AWT*. Dessa forma, apesar de nosso estudo nessa disciplina se concentrar em *Swing*, é inevitável referenciar a *AWT*.

Para utilizar as classes **Java Swing**, os programas a serem implementados devem importar classes de três pacotes:

- `import java.awt.*` *
- `import java.awt.event.*` *
- `import javax.swing.*` *

Se não fizermos essas importações, o NetBeans poderá indicar erro quando utilizarmos classes *Java Swing*. Assim, podemos fazer as três importações e, quando alguma das importações não for utilizada, nós a apagaremos.

Outra opção é deixar o NetBeans fazer as correções de importação classe a classe.

A-Z

Java Swing

É um conjunto de classes destinado à elaboração de aplicações com interface gráfica "padrão Windows".



JFrame

É uma classe do pacote Swing que fornece todas as propriedades, métodos e eventos de que precisamos para construir janelas “padrão Windows”.

Frame

É uma classe do pacote AWT responsável pela criação de janelas, parecidas com as encontradas no ambiente Windows (FUGIERI, 2006, p. 176).

7.2 JFrame

A janela é a parte mais importante da interface do aplicativo, pois é sobre ela que os demais componentes serão construídos. A janela principal de aplicativos Java para *desktop* é criada como uma instância da classe **JFrame** (herança da classe **Frame** implementada no pacote Java Swing).

A classe *JFrame* provê um conjunto de métodos que permitem criar e configurar janelas. Abaixo são citados alguns deles:

- *JFrame()*: construtor padrão. Apenas cria uma nova janela.
- *JFrame(String t)*: cria uma janela atribuindo um título a mesma.
- *getTitle()*: obtém o título da janela.
- *setTitle(String t)*: atribui um título à janela.
- *isResizable()*: verifica se a janela é redimensionável.
- *setResizable(boolean b)*: especifica se a janela é ou não redimensionável. Caso seja passado *true* como parâmetro, a janela será redimensionável. Caso o parâmetro passado seja *false*, a janela não será redimensionável.
- *setSize(int l, int a)*: define o tamanho da janela. Os parâmetros passados são a largura e a altura da janela.
- *setLocation(int x, int y)*: define a posição da janela na tela. O primeiro parâmetro a ser passado é a posição horizontal da janela a partir do lado esquerdo da tela. O segundo parâmetro define a posição vertical a partir da parte superior da tela.

A Figura 7.1 exibe um exemplo de código de janela criada a partir da classe *JFrame*. Note que nesse exemplo foi utilizada grande parte dos métodos citados acima.

```

package exemplosSwing;

import java.awt.Color;
import javax.swing.JFrame;

public class PrimeiraJanela extends JFrame {

    public PrimeiraJanela() {
        //Titulo da Janela
        this.setTitle("Primeiro aplicativo Swing");
        //Dimensões da Janela
        this.setSize(1000, 500);
        //Posição:Canto esquerdo superior da tela
        this.setLocation(150, 50);
        //Impedir que a janela seja redimensionada
        this.setResizable(false);
        //Colocar cor de fundo azul na janela
        this.getContentPane().setBackground(Color.blue);
    }

    public static void main(String[] args) {
        // Criar uma instancia do tipo "PrimeiraJanela"
        PrimeiraJanela jan = new PrimeiraJanela();
        //Tornar a janela visível
        jan.setVisible(true);
    }
}

```

Figura 7.1: Primeiro exemplo de janela *JFrame*

Fonte: Elaborada pelos autores

A classe *PrimeiraJanela* apresentada no exemplo 7.1 herda de *JFrame*. Assim, ela tem todos os métodos e atributos de *JFrame*.



No construtor da classe *PrimeiraJanela* temos chamadas a vários dos métodos da classe *JFrame* sendo chamados (*this.<metodo>*).

O método *getContentPane()* retorna um objeto que representa a parte interior da janela; esse objeto possui um método que possibilita alterarmos a cor de fundo (*setBackground*) desse objeto. Note, ainda, que configuramos a cor de fundo da janela para azul utilizando a classe *Color* e seu atributo estático *blue*.

A Figura 7.2 exibe a janela gerada pela execução do código apresentado na Figura 7.1.

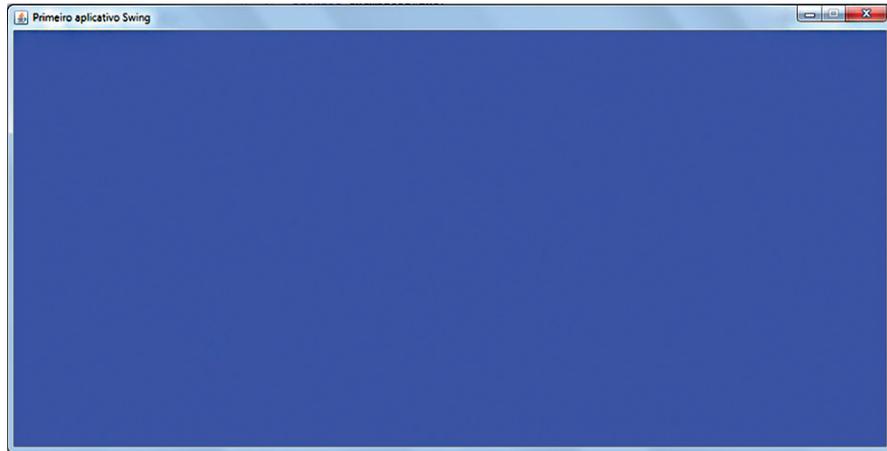


Figura 7.2: Janela gerada pelo exemplo da Figura 7.1

Fonte: Elaborada pelos autores

7.3 JLabel e ImageIcon

A classe *JLabel* é utilizada para criar etiquetas (*labels*) de textos nas janelas (FUGIERI, 2006, p. 178). Ela permite o controle de propriedades do texto a ser utilizado, tais como: alinhamento, tipo de letra, tamanho, cor, etc. A classe *JLabel* fornece vários construtores dentre os quais destacamos:

- *JLabel()*: construtor padrão.
- *JLabel(String)*: recebe como parâmetro uma *String* que será o texto apresentado pelo *Label*.
- *JLabel(String, int)*: além do texto a ser apresentado, recebe como parâmetro um inteiro que representa o tipo de alinhamento a ser utilizado.
- *JLabel(String, Image)*: além do texto a ser apresentado, recebe como parâmetro um *image* que será o ícone a ser exibido.
- *JLabel(String, Image, int)*: recebe como parâmetros o texto a ser apresentado, o ícone a ser exibido e o tipo de alinhamento a ser utilizado.

Outros dois métodos essenciais para o uso de *JLabels* são o método *getText()* que retorna o texto do *Label* e o *setText(String)* que especifica (altera) o texto a ser apresentado pelo *Label*.

A classe *ImageIcon* é utilizada, entre outras possibilidades, para colocar um ícone no *JLabel*. A Figura 7.3 exibe o código de uma janela que utiliza as classes *JLabel* e *ImageIcon*.

```

package javaswing_1;
import java.awt.*;
import javax.swing.*;
public class UsaJLabel_ImagemIcon extends JFrame {
    private JLabel label1, label2;
    private ImageIcon icone = new ImageIcon("C:/Users/Giovany/Desktop/AppInstalled.gif");
    public UsaJLabel_ImagemIcon() {
        this.setTitle("Labels");
        this.setSize(350, 120);
        this.setLocation(50, 50);
        this.getContentPane().setBackground(new Color(220, 220, 0));

        this.label1 = new JLabel(" Esquerda ", icone, JLabel.LEFT);
        this.label2 = new JLabel(" Direita ", JLabel.RIGHT);

        this.getContentPane().setLayout(new GridLayout(2, 1));
        this.getContentPane().add(this.label1);
        this.getContentPane().add(this.label2);
    }
    public static void main(String[] args) {
        JFrame janela = new UsaJLabel_ImagemIcon();
        janela.setUndecorated(true);
        janela.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        janela.setVisible(true);
    }
}

```

Figura 7.3: Exemplo de utilização de *JLabel* e *ImagemIcon*

Fonte: Elaborada pelos autores

No exemplo da Figura 7.3 note que a classe *UsaJLabel_ImagemIcon* herda da classe *Jframe*; assim, conseguimos utilizar seus métodos.



O método *setUndecorated* da classe *JFrame* retira da janela a borda padrão Windows. A linha *janela.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);*, por sua vez, coloca na janela a borda padrão Java.

Temos também a definição do *layout* da janela: *this.getContentPane().setLayout(new GridLayout(2, 1));*. O *layout Grid* divide a tela em partes iguais de linhas e colunas. No nosso exemplo, temos a tela dividida em duas linhas e uma coluna. A ordem de inserção de objetos nesse *layout* é da esquerda para a direita e de cima para baixo. A inserção é feita com o método *add* (*this.getContentPane().add(this.label1);*). Nos aprofundaremos no estudo dos *layouts* futuramente.

A Figura 7.4 exibe a janela gerada pelo código exibido na Figura 7.3.



Figura 7.4: Janela gerada pelo código da Figura 7.3

Fonte: Elaborada pelos autores

7.4 JOptionPane

A classe *JOptionPane* é utilizada para gerar caixas de diálogo (FUGIERI, 2006, p. 207). Ela nos permite criar vários tipos de caixas de diálogo, a saber: *MessageDialog*, *ConfirmDialog*, *InputDialog* e *OptionDialog*.

7.4.1 MessageDialog

Uma *MessageDialog* é uma caixa de diálogo que apresenta uma mensagem.

Sintaxe para criação: *JOptionPane.showMessageDialog* (<componente>, <mensagem>, <título da mensagem>, <tipo da mensagem>)

Parâmetros para criação:

<componente>: objeto contêiner que permite definir a posição da tela em que a caixa de mensagem aparecerá. Normalmente esse parâmetro é *null*.

<mensagem>: mensagem a ser exibida na caixa.

<título da mensagem>: será exibido na barra de título da caixa de mensagem.

<tipo da mensagem>: determina o ícone que aparecerá junto à mensagem. Podendo ser:

- Pergunta: *QUESTION_MESSAGE*
- Informação: *INFORMATION_MESSAGE*
- Alerta: *WARNING_MESSAGE*
- Erro: *ERROR_MESSAGE*
- Definido pelo Usuário: *INFORMATION_MESSAGE* (e acrescenta-se o argumento do tipo *ImageIcon* na chamada do *showMessageDialog*)
- Vazio: *PLAIN_MESSAGE*

Exemplo de utilização:

```
JOptionPane.showMessageDialog(null, s, "Login confirmado", JOptionPane.INFORMATION_MESSAGE);
```

7.4.2 ConfirmDialog

Uma *ConfirmDialog* é uma caixa de diálogo que apresenta uma mensagem e possibilita ao usuário responder uma pergunta.

Sintaxe para utilização:

```
int resposta = JOptionPane.showConfirmDialog (<componente>, <mensagem>, <título da mensagem>, <botões presentes>, <tipo da mensagem>
```

Parâmetros para utilização: os parâmetros são os mesmos apresentados no *MessageDialog* com acréscimo do parâmetro que indica os botões a serem apresentados:

< botões presentes>: determinam quais botões irão aparecer. Podendo ser:

YES_NO_OPTION ou 0: aparecem *yes* e *no*.

YES_NO_CANCEL ou 1: aparecem *yes*, *no* e *cancel*.

OK_CANCEL_OPTION ou 2: aparecem *ok* e *cancel*.

Retorno: O valor retornado pode ser:

YES_OPTION = 0

NO_OPTION = 1

CANCEL_OPTION = 2

Exemplo de utilização:

```
JOptionPane.showConfirmDialog(null, "Confirma login ?", "Caixa de confirmação", JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);
```

7.4.3 InputDialog

Uma *InputDialog* é uma caixa de diálogo que apresenta uma mensagem e possibilita que o usuário digite um texto.

Sintaxe para utilização:

```
String resposta = JOptionPane.showInputDialog(<componente>, <mensagem>, <título da mensagem>, <tipo da mensagem>)
```

Exemplo de utilização:

```
String s = JOptionPane.showInputDialog(null, "Digite seu login", "Login no sistema", JOptionPane.INFORMATION_MESSAGE);
```

7.4.4 *OptionDialog*

Uma *OptionDialog* é uma caixa de diálogo que possibilita a exibição de várias opções para escolha do usuário. Ela apresenta vários botões para o usuário e retorna um número inteiro indicando em qual dos botões o usuário clicou (o primeiro botão é representado pelo número zero; o segundo, pelo um e assim sucessivamente).

Sintaxe para utilização:

```
int resposta = JOptionPane.showOptionDialog (<componente>, <mensagem>, <título da mensagem>, <botoes presentes>, <tipo da mensagem>, <icone>, <array de objetos>, <seleção padrão>)
```

Parâmetros para utilização: os parâmetros são os mesmos apresentados nas anteriores com os seguintes acréscimos:

<array de objetos>: utilizado para exibir opções fora do padrão *YES_NO_OPTION*, *YES_NO_CANCEL_OPTION* ou *OK_CANCEL_OPTION* (personalizar botões de seleção).

< seleção padrão >: determina o botão padrão selecionado.

Exemplo:

```
String[] nomes = { "João", "Maria", "Pedro" };
```

```
int resp = JOptionPane.showOptionDialog (null, "Escolha um login padrão", "Login no sistema", 0, JOptionPane.INFORMATION_MESSAGE, null, nomes, nomes[0]);
```

A Figura 7.5 exibe um código que utiliza a classe *JOptionPane* para criar várias caixas de diálogo diferentes.

```

package javaswing_3;
import javax.swing.*;
public class UsaJOptionPane {
    public static void main(String[] args)
    {
        // TODO code application logic here
        String s = JOptionPane.showInputDialog(null, "Digite seu login", "Login no sistema", JOptionPane.QUESTION_MESSAGE);
        if ( s == null ) return;
        if ( JOptionPane.showConfirmDialog(null, "Confirma login ?", "Caixa de confirmação", JOptionPane.OK_CANCEL_OPTION,
            JOptionPane.QUESTION_MESSAGE) == 0)
        {
            JOptionPane.showMessageDialog(null, s, "Login confirmado", JOptionPane.INFORMATION_MESSAGE);
        }
        else
        {
            JOptionPane.showMessageDialog(null, s, "Login NÃO CONFIRMADO", JOptionPane.WARNING_MESSAGE);
            String[] nomes = { "João", "Maria", "Pedro", "Janaina" };
            int resp = JOptionPane.showOptionDialog(null, "Escolha um login padrão", "Login no sistema", 0,
                JOptionPane.INFORMATION_MESSAGE, null, nomes, nomes[1]);
            if ( resp == 1)
            {
                JOptionPane.showMessageDialog(null, "Login " + s + " inválido !!!", "Login confirmado",
                    JOptionPane.ERROR_MESSAGE);
            }
        }
    }
}

```

Figura 7.5: Exemplos de utilização da classe *JOptionPane*

Fonte: Elaborada pelos autores

O exemplo da Figura 7.5 não tem uma funcionalidade específica. Seu objetivo é apenas ilustrar o uso de métodos da classe *JOptionPane*.



7.5 Tratamento de eventos e *JButton*

Quando desenvolvemos aplicativos com interface gráfica precisamos programar as respostas que o sistema dará às interações do usuário com a interface gráfica. A essa programação do comportamento do sistema de acordo com as ações do usuário dá-se o nome de **tratamento de eventos**.

Quando o usuário clica em um botão, pressiona a tecla *ENTER* ou seleciona algo na tela, são disparados eventos correspondentes à ação do usuário e cabe ao programador definir qual o procedimento a ser adotado na ocorrência de tal evento. A interface *ActionListener* possibilita a identificação dos eventos, permitindo assim que seja programado o comportamento do sistema ao evento.

Para ilustrar a utilização da interface *ActionListener* para o tratamento de eventos, vamos desenvolver um exemplo (apresentado na Figura 7.6) que exibirá uma janela com um botão e, ao clicar no botão, será exibida uma caixa de diálogo. Para criar um botão na janela precisaremos utilizar a classe *JButton*. Dentre os métodos mais utilizados da classe *JButton* estão:

- *JButton()*: construtor que cria um botão sem texto.
- *JButton(String)*: construtor que cria um botão exibindo o texto passado como parâmetro.
- *JButton(String, Image)*: construtor que cria um botão com texto e imagem.

- `getLabel()`: obtém o texto apresentado pelo botão.
- `setLabel(String)`: define o texto a ser apresentado pelo botão.
- `setEnabled(boolean)`: define se o botão está habilitado (*true*) ou desabilitado (*false*).
- `setHorizontalTextPosition()`: define o alinhamento horizontal, que pode ser: *LEFT* (esquerda) ou *RIGHT* (direita).
- `setVerticalTextPosition()`: define o alinhamento vertical que pode ser: *TOP* (por cima) ou *BOTTOM* (por baixo).
- `setMnemonic(int)`: define o atalho (combinação de teclas) para acionar o botão (equivalente ao clique sobre o botão).
- `setToolTipText(String)`: possibilita colocar uma mensagem de ajuda no botão.



No exemplo da Figura 7.6 temos o uso do *layout* **Flow** (*FlowLayout*). Esse padrão de *layout* preenche as linhas da janela e, quando não é possível mais inserir componentes numa linha, avança para a linha seguinte. Os *layouts* serão objetos de estudo da próxima aula.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class UsaJButton extends JFrame implements ActionListener
{
    private JButton b1;
    public UsaJButton()
    {
        // Criando e setando atributos de b1
        b1 = new JButton("Mensagem");
        b1.setHorizontalTextPosition(AbstractButton.RIGHT);
        b1.setBackground(new Color(100, 180, 180));
        b1.setForeground(Color.black);
        b1.setFont(new Font("Scripts", Font.BOLD, 20));
        b1.setEnabled(true);
        b1.setToolTipText("Clique aqui para ver a mensagem");
        b1.setMnemonic(KeyEvent.VK_B); // Alt + B = Clique do mouse
        b1.addActionListener(this);

        this.setTitle("Inserindo botoes na janela");
        this.setSize(350, 100);
        this.setLocation(50, 50);
        this.getContentPane().setBackground(new Color(180, 180, 180));
        this.getContentPane().setLayout(new FlowLayout());
        this.getContentPane().add(this.b1);
    }
}
```

```

// Método declarada na classe ActionListener
public void actionPerformed(ActionEvent e)
{
    if ( e.getSource() == b1 )
        JOptionPane.showMessageDialog(null, "Mensagem", "Botão clicado",
        JOptionPane.INFORMATION_MESSAGE);
}

public static void main(String[] args)
{
    JFrame janela = new UsaJButton();
    janela.setUndecorated(true);
    janela.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
    janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    janela.setVisible(true);
}
}

```

Figura 7.6: Exemplo de tratamento de evento e de utilização de *JButton*

Fonte: Elaborada pelos autores

Note que no exemplo da Figura 7.6, nossa janela, além de herdar de *JFrame*, implementa a interface *ActionListener*. Sempre que quisermos tratar eventos como cliques em botões de nossas janelas, elas deverão implementar tal interface e, por consequência, teremos de implementar o método *actionPerformed* (*ActionEvent e*) que definirá o comportamento do programa sempre que um evento for detectado. Note o método *actionPerformed* do exemplo da Figura 7.6: sempre que um evento for disparado, é verificado se ele foi disparado pelo botão (*if e.getSource() == b1*) e, em caso positivo, uma caixa de diálogo é exibida.

É importante ressaltar que para que o método *actionPerformed* seja chamado na ocorrência de um evento em um objeto, esse objeto deve “avisar” para a janela que ele deseja ser “ouvido”. Isso é feito pelo método *addActionListener*. No exemplo da Figura 7.6 os eventos do botão só serão “ouvidos” devido à linha *b1.addActionListener(this)*.

Assim como tratamos eventos de um botão, podemos tratar eventos de outros componentes de interface, bastando para isso “avisar” que eles devem ser “ouvidos” e programar o método *actionPerformed* para responder ao evento.



7.6 *JTextField* e *JPasswordField*

A classe *JTextField* é utilizada para criar caixas de texto. Normalmente, utiliza-se a classe *JLabel* para apresentar um texto fixo e a classe *JTextField* para campos de texto a serem digitados pelos usuários.

Dentre os métodos utilizados para manipular um *TextField*, destacam-se:

- *TextField()*: construtor padrão utilizado para criar uma caixa de texto vazia.
- *TextField(String)*: construtor utilizado para criar uma caixa de texto com a *String* fornecida.
- *TextField(int)*: construtor utilizado para criar uma caixa de texto com a quantidade de colunas especificada.
- *TextField(String, int)*: construtor utilizado para criar uma caixa de texto com uma determinada *String* e com a quantidade de colunas especificada.
- *getText()*: obtém o texto do objeto.
- *setText(String)*: atribui uma *String* ao objeto.
- *getSelectedText()*: obtém o texto selecionado no objeto.
- *isEditable()*: verifica se o componente é editável ou não e retornando um *boolean* (*true* ou *false*).
- *setEditable(boolean)*: especifica se o componente é editável ou não.

Há uma classe especial para criar caixas de texto próprias para campos de senha: *JPasswordField* (FUGIERI, 2006, p. 190). O funcionamento dessa classe é bastante semelhante ao de *TextField*. A diferença básica é que os caracteres digitados são substituídos por outros para ocultar a senha. Assim, os principais métodos de *JPasswordField* são idênticos aos de *TextField* (exceto os construtores, obviamente) merecendo destaque adicionam apenas os seguintes métodos:

- *setEchoChar(char)*: determina o caracter que será utilizado para esconder a senha.
- *getPassword()*: gera um vetor de char com a senha digitada.
- *getText()*: método obsoleto que retorna a *String* do objeto.

A Figuras 7.7 e 7.8 exibem o código de uma janela que utiliza as classes *JTextField* e *JPasswordField* (o código de uma janela está dividido nas duas figuras). Note que nesse exemplo foi criado um método privado *montarLayout* que é responsável por montar o *layout* da janela, ou seja, por inserir e organizar os componentes na janela. Assim como no exemplo da Figura 7.4 foi utilizado o *layout GridLayout*.

```
package javaswing_1;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class UsaJTextFieldJPasswordField extends JFrame implements ActionListener{
    private JLabel l1, l2;
    private JTextField t1;
    private JPasswordField p2;
    private JButton b1;
    private JPanel painel;
    public UsaJTextFieldJPasswordField() {
        this.l1 = new JLabel("Nome ", JLabel.LEFT);
        this.t1 = new JTextField();
        this.l2 = new JLabel("Senha ", JLabel.LEFT);
        this.p2 = new JPasswordField();
        this.p2.setEchoChar('*');
        this.painel = new JPanel();
        this.b1 = new JButton("Mostrar");

        this.setTitle("Usando JTextField e JPasswordField");
        this.setSize(350, 100);
        this.setLocation(50, 50);
        this.getContentPane().setBackground(new Color(180, 180, 180));

        this.montarLayout();
    }
}
```

Figura 7.7: Primeira parte do exemplo de utilização de *JTextField* e *JPasswordField*

Fonte: Elaborada pelos autores

```
private void montarLayout() {
    this.getContentPane().setLayout(new GridLayout(3, 2));
    this.getContentPane().add(this.l1);
    this.getContentPane().add(this.t1);
    this.getContentPane().add(this.l2);
    this.getContentPane().add(this.p2);
    this.getContentPane().add(this.painel);
    this.getContentPane().add(this.b1);
    this.b1.addActionListener(this);
}

public void actionPerformed(ActionEvent e) {
    if ( e.getSource() == this.b1 ) {
        String s = "Usuário: " + this.t1.getText() + " Senha: " + new String(this.p2.getPassword());
        JOptionPane.showMessageDialog(null, s, "Mensagem",
            JOptionPane.INFORMATION_MESSAGE);
    }
}

public static void main(String[] args) {
    JFrame janela = new UsaJTextFieldJPasswordField();
    janela.setUndecorated(true);
    janela.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
    janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    janela.setVisible(true);
}
}
```

Figura 7.8: Segunda parte do exemplo de utilização de *JTextField* e *JPasswordField*

Fonte: Elaborada pelos autores

No método *actionPerformed*, que como vimos anteriormente define o tratamento de eventos para a janela, temos a criação de uma caixa de mensagem por meio da classe *JOptionPane* e seu método *showMessageDialog* (veja a caixa de mensagem criada na Figura 7.10). Lembre-se que para tratar o evento de clique no botão, não podemos esquecer de fazer a janela “ouvir” esse evento. Para isso foi utilizado o comando `this.b1.addActionListener(this);`.



Note que no exemplo das Figuras 7.7 e 7.8 fizemos uso também de um componente gráfico que ainda não conhecíamos: o *JPanel*. Utilizando *JPanel* podemos dividir uma janela em painéis independentes de forma que podemos aplicar *layouts* diferentes a cada painel e inserir nos painéis outros componentes de interface. Porém, no caso específico desse exemplo utilizamos um *JPanel* apenas para ocupar o espaço abaixo dos rótulos de texto de forma que o botão ficasse alinhado abaixo dos campo para preenchimento de senha. A Figura 7.9 exibe a janela criada.



A Figura 7.9 exibe a janela criada pela execução do código apresentado nas Figuras 7.7 e 7.8.

Para facilitar o entendimento do uso de *JPanel* nesse exemplo, refaça o exemplo das Figuras 7.7 e 7.8 retirando do código a utilização do *JPanel* e veja o resultado. Note que o botão será deslocado para a esquerda.



Figura 7.9: Janela gerada pela execução do código apresentado nas Figuras 7.7 e 7.8

Fonte: Elaborada pelos autores

A Figura 7.10, por sua vez, exibe a caixa de diálogo criada pelo evento de clique no botão conforme define o método *actionPerformed*.

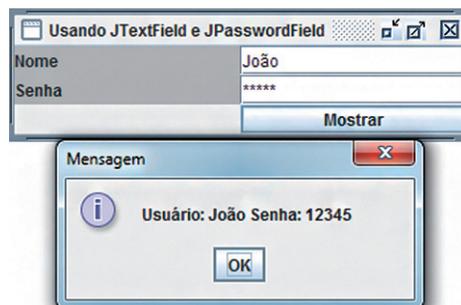


Figura 7.10: Caixa de mensagem criada ao clicar no botão

Fonte: Elaborada pelos autores

Resumo

Nesta aula começamos a criar interfaces gráficas em Java. Aprendemos que a principal biblioteca de Java para trabalhar interfaces gráficas é a biblioteca *Swing*. Vimos que a classe *JFrame* é responsável por criar uma janela e aprendemos diversas outras classes que representam vários tipos de componentes gráficos. Por fim, vimos como tratar eventos ocorridos nas interfaces. Com esses conhecimentos já somos capazes de criar nossos primeiros programas com interfaces gráficas em Java.

Atividades de aprendizagem

1. Faça um programa com interface gráfica que permita criar contas do tipo *ContaEspecial* (utilize as classes criadas em exercícios anteriores). Sua janela deve conter campos para que o usuário informe o nome do titular, o saldo da conta e o limite da conta e um botão que o usuário deverá clicar para criar a conta. Assim, o usuário deverá preencher os campos e, quando clicar no botão o programa, deve instanciar uma nova conta com os dados informados e o número da conta será sequencial atribuído pelo próprio programa. Então o programa deve inserir a conta criada em uma lista que conterá todas as contas já criadas e deve exibir uma janela de mensagem (*JOptionPane*) informando o número da conta criada e o nome do seu titular.
2. Acrescente à janela criada no exercício anterior um botão que, ao ser clicado, exiba os números e os nomes dos titulares de todas as contas criadas em uma janela de mensagem (*JOptionPane*).
3. No mesmo projeto do exercício anterior, crie uma nova janela que contenha um campo no qual o usuário digitará um número de uma conta e um botão para consultar o saldo da conta. Na janela criada no exercício anterior, crie um botão que, ao ser clicado, instanciará a nova janela e a tornará visível. O construtor da nova janela receberá como parâmetro um *ArrayList*. Assim, quando a nova janela for instanciada, já receberá como parâmetro o *ArrayList* de contas que existe na antiga. Então o usuário deverá preencher na nova janela o número da conta e clicar no botão de consultar saldo. Nesse momento o sistema deve procurar pela conta desejada na lista. Se encontrar, deve exibir o saldo da conta em uma janela de mensagem (*JOptionPane*). Se não encontrar, deve exibir uma mensagem de erro informando que a conta não existe.



A classe *JTextField* não foi projetada para permitir a digitação de textos em múltiplas linhas. Para isso Java tem a classe *JTextArea* que tem funcionamento muito parecido com a classe *JTextField*. Acesse a documentação oficial da Oracle sobre a classe *JTextArea* em <http://download.oracle.com/javase/6/docs/api/javax/swing/JTextArea.html> e, utilizando as informações lá apresentadas, acrescente ao exemplo da Figura 7.7 um *JTextArea* que permita ao usuário digitar uma ou mais frases de lembrete sobre sua senha.



Todas as janelas de exemplo que fizemos têm um método *main*, ou seja, são executáveis. Porém, em programas comuns temos várias janelas que não são executáveis, e sim são exibidas a partir de outras. Para exibir uma janela a partir de um clique em um botão de uma outra janela, basta programar esse evento de forma que a janela a ser exibida seja instanciada e se torne visível, como fazemos no método *main* do exemplo apresentado na Figura 7.8.

Aula 8 – Interfaces gráficas em Java – parte II

Objetivos

Aprender o conceito de padrões de *layout*.

Conhecer alguns padrões de *layout* de Java.

Conhecer mais classes para construção de interfaces gráficas em Java: *JComboBox*, *JCheckBox*, *JRadioButton*, *ButtonGroup*, *JMenuBar*, *JMenu* e *JMenuItem*.

Desenvolver novos exemplos de tratamento de eventos sobre interface gráfica em Java.

8.1 Padrões de *layout*

Os padrões de *layout* definem como os objetos serão dispostos na janela (FUGIERI, 2006, p. 230). É possível desenvolver uma janela sem a utilização de um padrão de *layout*, mas, nesse caso, é necessário, para cada objeto, definir manualmente sua posição na janela, o que gera mais flexibilidade, mas também mais trabalho para a definição do *layout*. O uso de padrões de *layout* deixa nossas janelas mais genéricas e adaptáveis a modificações.

Nesta aula aprenderemos como utilizar dois padrões de *layout*: *FlowLayout* e *GridLayout*.

8.1.1 *FlowLayout*

O padrão de *layout* *FlowLayout* insere os objetos, um após o outro, linha a linha. Assim, quando não é mais possível inserir objetos numa linha, é criada uma nova linha e novos objetos podem ser inseridos. Esse padrão de *layout* assemelha-se a um editor de textos, pois quando não existe mais espaço numa dada linha é criada uma nova linha para inserção de mais conteúdos.

A Figura 8.1 apresenta um exemplo de uso do padrão *FlowLayout*.

```

package padroesLayout;

import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JRootPane;

public class ExemploFlowLayout extends JFrame {

    public ExemploFlowLayout(){
        FlowLayout layout = new FlowLayout(FlowLayout.CENTER, 10, 10);
        this.setLayout(layout);
        this.add(new JLabel("O rato roeu a roupa do rei de Roma"));
        this.add(new JLabel("5 minutos para daqui a pouco"));
        this.add(new JLabel("Cara caramba cara cara oh"));
        this.add(new JLabel("A"));
        this.add(new JLabel("B"));
    }

    public static void main(String[] args) {
        JFrame janela = new ExemploFlowLayout();
        janela.setBounds(250, 50, 280, 170);
        janela.setTitle("FlowLayout");
        janela.setUndecorated(true);
        janela.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        janela.setVisible(true);
    }
}

```

Figura 8.1: Exemplo de uso de *FlowLayout*

Fonte: Elaborada pelos autores



Note no exemplo que o construtor do *FlowLayout* recebe três parâmetros: o alinhamento desejado (no caso utilizamos alinhamento centralizado), o espaçamento horizontal e o espaçamento vertical. Para aplicar o *layout* à janela, utilizamos o método *setLayout*.

A Figura 8.2 exibe o resultado da execução do código apresentado na Figura 8.1.

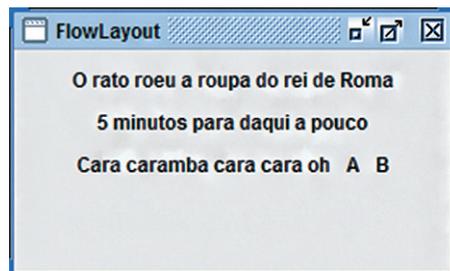


Figura 8.2: Janela utilizando *FlowLayout*

Fonte: Elaborada pelos autores

8.1.2 GridLayout

O padrão de *layout* *GridLayout* organiza os objetos como uma tabela, com células de objetos de mesmo tamanho. É um *layout* flexível, pois uma vez redimensionada a janela ele ajusta automaticamente os objetos de forma que o padrão se mantenha, ou seja, que cada objeto de cada célula seja apresentado com o mesmo tamanho.

A Figura 8.3 apresenta um exemplo de uso do padrão *GridLayout*.

```
package padroesLayout;

import java.awt.GridLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JRootPane;

public class ExemploGridLayout extends JFrame {

    public ExemploGridLayout(){
        GridLayout layout = new GridLayout(2, 3, 10, 10);
        this.setLayout(layout);
        this.add(new JLabel("O rato roeu a roupa do rei de Roma"));
        this.add(new JLabel("5 minutos para daqui a pouco"));
        this.add(new JLabel("Cara caramba cara cara oh"));
        this.add(new JLabel("A"));
        this.add(new JLabel("B"));
    }

    public static void main(String[] args) {
        JFrame janela = new ExemploGridLayout();
        janela.setBounds(250, 50, 600, 170);
        janela.setTitle("GridLayout");
        janela.setUndecorated(true);
        janela.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        janela.setVisible(true);
    }
}
```

Figura 8.3: Exemplo de uso de *GridLayout*

Fonte: Elaborada pelos autores

Note no exemplo que o construtor do *GridLayout* recebe quatro parâmetros: o número de linhas em que a janela será dividida (no caso 2), o número de colunas em que a janela será dividida (no caso 3), o espaçamento horizontal e o espaçamento vertical. Para aplicar o *layout* à janela, utilizamos o método *setLayout*.

A Figura 8.4 exibe o resultado da execução do código apresentado na Figura 8.3.

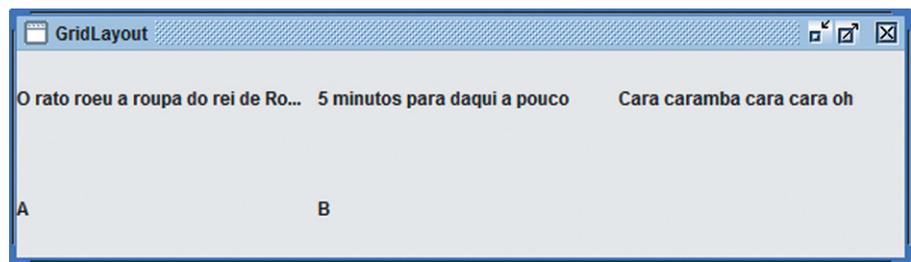


Figura 8.4: Janela utilizando *GridLayout*

Fonte: Elaborada pelos autores

8.2 *JComboBox* e tratamento de eventos

A classe *JComboBox* é utilizada para criar caixas que permitem que o usuário selecione apenas um item da sua lista de opções.

Para criar um objeto do tipo *JComboBox*, é necessário passar um vetor de *Strings* que indicará as opções de seleção a serem exibidas na caixa.

Exemplo de Criação:

```
String[] animais = { "Leão", "Elefante", "Cobra", "Jabuti" };  
JComboBox combo = new JComboBox(animais);
```

Dentre os métodos mais utilizados da classe *JComboBox*, destacam-se:

- *getSelectedIndex*: retorna o índice do item selecionado no *combobox*. É importante notar que o índice do primeiro elemento é 0.
- *removeItemAt*: dado um índice, elimina do *combobox* o item que está nessa posição.
- *removeAllItems*: remove todos os itens do *combobox*.
- *addItem*: dado um novo item, insere-o no *combobox*.
- *getSelectedItem*: retorna o item selecionado.
- *getItemCount*: retorna a quantidade de itens do *combobox*.

A seguir é apresentado um exemplo em que utilizamos um *combobox* para exibir uma listagem de animais. A janela construída no exemplo permite incluir novas opções no *combobox* e exibe um texto informando a opção se-

lecionada. Para isso, foi preciso tratar o evento de clique em um botão (para adicionar item) e o evento de mudança de item selecionado.

Como o exemplo ficou muito extenso, ele foi dividido em três Figuras (8.5, 8.6 e 8.7). Mas note que as três figuras juntas apresentam o código de uma única classe.

```
package javaswing2;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class UsaJComboBox extends JFrame implements ActionListener, ItemListener {
    JLabel label1;
    JTextField tadicionar, t1, t2;
    JComboBox combo;
    JButton novoItem;

    private void criarComponentesJanela()
    {
        this.label1 = new JLabel("Animais");
        this.label1.setForeground(Color.blue);
        this.label1.setFont(new Font("Arial", Font.BOLD, 15));
        this.novoItem = new JButton("Adiciona item");
        this.t1 = new JTextField();
        this.t2 = new JTextField();
        this.tadicionar = new JTextField();
        String[] animais = { "Leão", "Elefante", "Cobra", "Jaboti" };
        combo = new JComboBox(animais);
    }
}
```

Figura 8.5: Exemplo de utilização de JComboBox – parte I

Fonte: Elaborada pelos autores

```
public void actionPerformed(ActionEvent e) {
    //adiciona item
    if ( e.getSource() == this.novoItem && this.tadicionar.getText().length() != 0 )
    {
        this.combo.addItem(tadicionar.getText());
        this.tadicionar.setText("");
    }
}

public void itemStateChanged(ItemEvent e) {
    this.t1.setText(" O valor selecionado é: " + this.combo.getSelectedItem().toString());
    this.t2.setText("Existem " + String.valueOf(this.combo.getItemCount()) + " itens ");
}

private void setarAtributosJanela()
{
    this.setTitle("Usando JComboBox");
    this.setBounds(50, 50, 400, 170);
    this.getContentPane().setBackground(new Color(190, 190, 190));
    this.getContentPane().setLayout(new GridLayout(3, 2));
    this.getContentPane().add(this.label1);
    this.getContentPane().add(this.combo);
    this.getContentPane().add(this.novoItem);
    this.getContentPane().add(this.tadicionar);
    this.getContentPane().add(this.t1);
    this.getContentPane().add(this.t2);
}
}
```

Figura 8.6: Exemplo de utilização de JComboBox– parte II

Fonte: Elaborada pelos autores

```

private void adicionarListeners()
{
    this.novoItem.addActionListener(this);
    this.combo.addItemListener(this);
}

public UsaJComboBox()
{
    this.criarComponentesJanela();
    this.setarAtributosJanela();
    this.adicionarListeners();
}

public static void main(String[] args) {
    JFrame janela = new UsaJComboBox();
    janela.setUndecorated(true);
    janela.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
    janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    janela.setVisible(true);
}
}

```

Figura 8.7: Exemplo de utilização de <IT>JComboBox<IT>– parte III

Fonte: Elaborada pelos autores

No exemplo apresentado temos um campo de texto que informa qual é o item atualmente selecionado no *combobox*. Assim, sempre que o usuário alterar o item selecionado, temos que identificar a ocorrência desse evento e atualizar o texto exibido. Para que o evento de mudança de item selecionado seja acionado, é necessário adicionar o *combobox* ao controlador de eventos de mudança de item (*ItemListener*) do próprio *combobox*. Para isso, utilizamos o comando: *this.combo.addItemListener(this)*. Adicionalmente, para que a janela seja capaz de “ouvir”, ou melhor, de tratar o evento, ela deverá implementar a interface *ItemListener* e, conseqüentemente, implementar o método *itemStateChanged* que definirá o comportamento da janela sempre que houver mudança de item selecionado em algum dos *combobox* adicionados na janela. O método *itemStateChanged* do nosso exemplo é apresentado na Figura 8.6.



No exemplo apresentado, se houvesse mais combos na janela seria necessário determinar o combo selecionado utilizando *e.getSource() == this.combo* (como foi feito no método *actionPerformed* do mesmo exemplo).

A Figura 8.8 exibe a janela gerada pela execução do exemplo apresentado pelas três figuras anteriores.

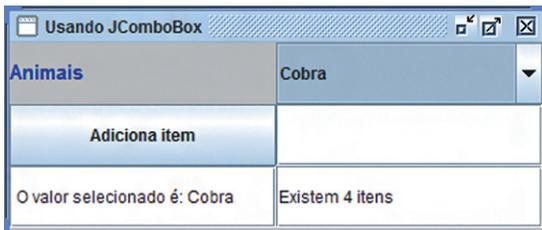


Figura 8.8: Janela gerada pela execução do exemplo anterior

Fonte: Elaborada pelos autores

8.3 JCheckBox

As caixas de opção são criadas a partir da classe *JCheckBox* e permitem representar uma opção que está ativada (*true*) ou desativada (*false*) (FUGIERI, 2006, p. 201).

As Figuras 8.9 e 8.10 apresentam um exemplo que utiliza *JCheckBox*. Nesse exemplo há um texto e um *checkbox*. Se o *checkbox* estiver selecionado, o texto será apresentado em negrito e, caso contrário, será apresentado em fonte plana. Para verificar se um objeto *JCheckBox* está selecionado, utilizamos o método *isSelected()*.

Como o exemplo ficou muito extenso, ele foi dividido em duas Figuras (8.9 e 8.10). Mas, note que as duas figuras juntas apresentam o código de uma única classe.



```
package javaswing2;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class UsaJCheckBox extends JFrame implements ItemListener {
    JLabel label1;
    JCheckBox c1;

    private void criarComponentesJanela()
    {
        this.label1 = new JLabel("O rato roeu");
        this.label1.setFont(new Font("Arial", Font.PLAIN, 20));
        this.c1 = new JCheckBox("Negrito");
    }

    private void setarAtributosJanela()
    {
        this.setTitle("Usando JCheckBox");
        this.setBounds(250, 50, 300, 70);
        this.getContentPane().setBackground(new Color(190, 190, 190));
        this.getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER));
        this.getContentPane().add(this.label1);
        this.getContentPane().add(this.c1);
    }
}
```

Figura 8.9: Exemplo de utilização de *JCheckBox* – parte I

Fonte: Elaborada pelos autores

```

private void adicionarActionListeners()
{
    this.c1.addItemListener(this);
}
public UsaJCheckBox() {
    this.criarComponentesJanela();
    this.setarAtributosJanela();
    this.adicionarActionListeners();
}
public void itemStateChanged(ItemEvent e)
{
    if ( e.getSource() == this.c1)
    {
        if ( this.c1.isSelected() )
            this.label1.setFont(new Font("Arial", Font.BOLD, 20));
        else
            this.label1.setFont(new Font("Arial", Font.PLAIN, 20));
    }
}
public static void main(String[] args) {
    JFrame janela = new UsaJCheckBox();
    janela.setUndecorated(true);
    janela.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
    janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    janela.setVisible(true);
}
}

```

Figura 8.10: Exemplo de utilização de *JCheckBox* – parte II

Fonte: Elaborada pelos autores

Note no exemplo que, assim como acontece com *JComboBox*, os eventos de alteração de estados da classe *JCheckBox* acionam o método *ItemStateChanged* da interface *ItemListener* (veja o método *ItemStateChanged* do exemplo na Figura 8.10).

Note também que é necessário acrescentar objetos *JCheckBox* ao tratador de eventos para que o método *itemStateChanged* reconheça os eventos ocorridos com os *checkbox*. No exemplo isso é feito pelo comando: *this.c1.addItemListener(this);*.

A Figura 8.11 exibe a janela gerada pela execução do exemplo apresentado pelas duas figuras anteriores.

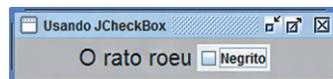


Figura 8.11: Janela gerada pela execução do exemplo da Figura 8.9 e 8.10

Fonte: Elaborada pelos autores

8.4 *JRadioButton* e *ButtonGroup*

A classe *JRadioButton* é utilizada para criar “botões de rádio”. Esses botões são graficamente muito semelhantes aos *checkbox*. A diferença entre *radio-*

button e *checkbox* está no fato que os primeiros são usados em conjuntos de forma que apenas um elemento do conjunto pode estar selecionado em um dado momento. Mas, para haver esse controle, os *JRadioButton* devem estar em um grupo representado pela classe *ButtonGroup*.

Um objeto da classe *ButtonGroup* não é visual, ou seja, não tem impacto na interface gráfica, mas sem ele perdemos a garantia de que apenas um botão de rádio está selecionado.



Dentre os métodos mais utilizados da classe *JRadioButton*, destacamos:

- *setMnemonic*: permite que uma combinação de teclas tenha o mesmo efeito do clique sobre um objeto *JRadioButton*.
- *isSelected*: determina se um botão de rádio está selecionado.
- *setSelected*: recebe um parâmetro *booleano* (*true* ou *false*) que faz com que o botão de rádio seja selecionado ou não.

As Figuras 8.12, 8.13 e 8.14 exibem um exemplo de utilização de *JRadioButton*.

Como o exemplo ficou muito extenso, ele foi dividido em três Figuras (8.12, 8.13 e 8.14). Mas, note que as três figuras juntas apresentam o código de uma única classe.



```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class UsaJRadioButton extends JFrame implements ItemListener {
    JLabel label1, label2;
    JTextField t1, t2, t3;
    JRadioButton radio1, radio2, radio3;
    ButtonGroup radiogroup;
    private void criarComponentesJanela()
    {
        this.label1 = new JLabel("Números");
        this.label2 = new JLabel("Resultado: ");
        this.t1 = new JTextField(5);
        this.t2 = new JTextField(5);
        this.t3 = new JTextField(5);
        this.t3.setEditable(false);
        this.radio1 = new JRadioButton("+");
        this.radio2 = new JRadioButton("-");
        this.radio3 = new JRadioButton("**");
        this.radiogroup = new ButtonGroup();
    }
}
```

Figura 8.12: Exemplo de utilização de *JRadioButton* – parte I

Fonte: Elaborada pelos autores

```

private void setarAtributosButtonGroup()
{
    this.radiogroup.add(this.radio1);
    this.radiogroup.add(this.radio2);
    this.radiogroup.add(this.radio3);
}

private void setarAtributosJanela()
{
    this.setTitle("RadioButtons");
    this.setBounds(230, 50, 340, 120);
    this.getContentPane().setLayout(new GridLayout(3,3));
    this.getContentPane().add(this.label1);
    this.getContentPane().add(this.t1);
    this.getContentPane().add(this.t2);
    this.getContentPane().add(this.radio1);
    this.getContentPane().add(this.radio2);
    this.getContentPane().add(this.radio3);
    this.getContentPane().add(this.label2);
    this.getContentPane().add(this.t3);
    this.getContentPane().add(new JPanel());
}

```

Figura 8.13: Exemplo de utilização de *JRadioButton* – parte II

Fonte: Elaborada pelos autores

```

private void adicionarItemListeners() {
    this.radio1.addItemListener(this);
    this.radio2.addItemListener(this);
    this.radio3.addItemListener(this);
}

public UsaJRadioButton() {
    this.criarComponentesJanela();
    this.setarAtributosJanela();
    this.setarAtributosButtonGroup();
    this.adicionarItemListeners();
}

public void itemStateChanged(ItemEvent e) {
    try {
        float n1 = Float.parseFloat(t1.getText());
        float n2 = Float.parseFloat(t2.getText());
        float result = 0;
        if (e.getSource() == this.radio1) result = n1 + n2;
        if (e.getSource() == this.radio2) result = n1 - n2;
        if (e.getSource() == this.radio3) result = n1 * n2;
        t3.setText("" + result);
    } catch (NumberFormatException erro) { this.t3.setText("Erro"); }
}

public static void main(String[] args) {
    JFrame janela = new UsaJRadioButton();
    janela.setUndecorated(true);
    janela.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
    janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    janela.setVisible(true);
}
}

```

Figura 8.14: Exemplo de utilização de *JRadioButton* – parte III

Fonte: Elaborada pelos autores

Note que o tratamento dos eventos de mudança de estados dos *radiobuttons* também é feito através do método *itemStateChanged*. Note também que, para que o método *itemStateChanged* seja disparado quando ocorre um evento em um *radiobutton* é necessário acrescentar o objeto *JRadioButton* ao tratador de eventos (*this.radio1.addItemListener(this)*).

No exemplo apresentado são utilizados dois campos de texto nos quais o usuário deve informar dois números, e os *radiobuttons* são utilizados para permitir que se selecione a operação aritmética a ser realizada com os números digitados. Assim, quando o usuário seleciona a operação, o resultado dela é apresentado em um terceiro campo de texto posicionado na parte inferior da janela. A Figura 8.15 exibe a janela gerada pela execução desse exemplo.

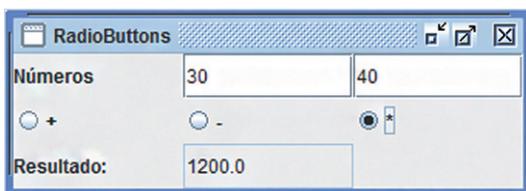


Figura 8.15: Janela gerada pela execução do exemplo de utilização de *radiobutton*

Fonte: Elaborada pelos autores



É comum os *JRadioButton* serem colocados em painéis (*JPanel*) nas janelas como forma de destacar seu agrupamento. Os *JPanel* podem assumir vários tipos de *layout*, assim como um *JFrame* qualquer. Pode-se colocar quantos *JPanel* forem desejados em uma *JFrame*.

8.5 *JMenuBar*, *JMenu* e *JMenuItem*

Agora aprenderemos a criar barras de menu em nossas janelas (FUGIERI, 2006, p. 222). Para que nossa janela conte com uma barra de menu, é necessário, inicialmente, instanciar uma barra de menus, ou seja, uma *JMenuBar* e associá-la à janela. Para isso, utilizamos comandos como os apresentados abaixo:

```
JMenuBar menuBar = new JMenuBar();  
this.setJMenuBar(menuBar);
```

Para acrescentar um menu na barra de Menu, precisamos instanciar um objeto da classe *JMenu* e depois adicioná-lo à barra de menu. As linhas de código abaixo apresentam um exemplo de como fazer isso:

```
JMenu menu1 = new JMenu(<nome do menu>);  
menuBar.add(menu1)
```

Por fim, para acrescentar itens a um menu, precisamos instanciar objetos da classe *JMenuItem* e adicioná-los ao menu, como nas linhas apresentadas a seguir:

```
JMenuItem menuItem1 = new JMenuItem(< nome do item do menu >);
menu1.add(menuItem1);
```

As Figuras 8.16, 8.17 e 8.18 exibem um exemplo de criação de janela com barra de menu.



Como o exemplo ficou muito extenso, ele foi dividido em três Figuras (8.16, 8.17 e 8.18). Mas, note que as três figuras juntas apresentam o código de uma única classe.

```
package javaswing2;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class UsaJMenu extends JFrame implements ActionListener {

    JMenuBar menuBar;
    JTextField t1;
    JMenu menuArquivo, menuCliente;
    JMenuItem miAjuda, miClienteEspecial, miClienteComum, miFornecedor, miSair;

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == miClienteEspecial) {
            t1.setText("Escolhido o item Cliente Especial");
        } else if (e.getSource() == miClienteComum) {
            t1.setText("Escolhido o item Cliente Comum");
        } else if (e.getSource() == miFornecedor) {
            t1.setText("Escolhido o item Fornecedor");
        } else if (e.getSource() == miAjuda) {
            t1.setText("Escolhido o menu Ajuda");
        } else if (e.getSource() == miSair) {
            System.exit(0);
        } else {
            t1.setText("Item escolhido inválido");
        }
    }
}
```

Figura 8.16: Exemplo de criação de janela com barra de menu – parte I

Fonte: Elaborada pelos autores

```

public UsaJMenu() {
    this.setBounds(150, 50, 480, 280);
    this.setTitle("Usando o JMenu");
    this.getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER));
    this.t1 = new JTextField(30);
    this.getContentPane().add(t1);
    this.menuBar = new JMenuBar();
    this.setJMenuBar(this.menuBar);
    this.menuArquivo = new JMenu("Arquivo");
    this.menuCliente = new JMenu("Cliente");
    this.miClienteComum = new JMenuItem("Cliente Comum");
    this.miClienteEspecial = new JMenuItem("Cliente Especial");
    this.menuArquivo.add(menuCliente);
    this.menuCliente.add(this.miClienteComum);
    this.menuCliente.add(this.miClienteEspecial);
    this.miFornecedor = new JMenuItem("Fornecedor");
    this.menuArquivo.add(this.miFornecedor);
    this.miSair = new JMenuItem("Sair");
    this.menuArquivo.add(this.miSair);
    this.miAjuda = new JMenuItem("Ajuda");
    this.menuBar.add(this.menuArquivo);
    this.menuBar.add(this.miAjuda);
    this.miAjuda.addActionListener(this);
    this.miClienteComum.addActionListener(this);
    this.miClienteEspecial.addActionListener(this);
    this.miFornecedor.addActionListener(this);
    this.miSair.addActionListener(this);
}

```

Figura 8.17: Exemplo de criação de janela com barra de menu – parte II

Fonte: Elaborada pelos autores

```

public static void main(String[] args) {
    JFrame janela = new UsaJMenu();
    janela.setUndecorated(true);
    janela.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
    janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    janela.setVisible(true);
}

```

Figura 8.18: Exemplo de criação de janela com barra de menu – parte III

Fonte: Elaborada pelos autores

No exemplo apresentado, é exibido um texto em uma caixa de acordo com o menu selecionado na janela. Note que, para isso, mais uma vez implementamos o método *actionPerformed* da *interface ActionListener* (Figura 8.16), a exemplo do que fizemos para tratar o evento de clique em botão. Note também que foi necessário adicionar os menus aos *actionListeners* (Figura 8.17).



Note que apenas os *JMenuItem* respondem à interface *ActionListener*. Os *JMenu* e os *JMenuBar* não têm como acionar o método *actionPerformed*.

Na Figura 8.19 é exibida a janela gerada pela execução desse exemplo.

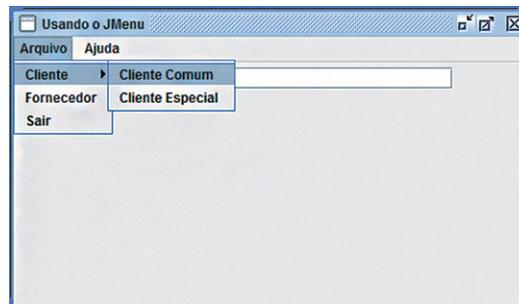


Figura 8.19: Janela gerada pela execução do exemplo de criação de janela com barra de menu

Fonte: Elaborada pelos autores

Resumo

Nesta aula aprendemos a utilizar várias classes *Swing* que criam diversos componentes de interface gráfica. Aprendemos o conceito de padrões de *layout* e dois dos padrões mais utilizados em Java e vimos novos exemplos de tratamento de eventos. Com o que aprendemos nas duas últimas aulas, somos capazes de criar programas com interfaces gráficas em Java; porém, este assunto não foi esgotado nessas aulas. A intenção dessas aulas é dar uma boa introdução ao assunto, de forma que você tenha base para se aprofundar futuramente.

Atividades de aprendizagem

1. Faça um programa com interface gráfica que permita ao usuário criar contas (use o exercício 1 da aula 7 como base). Assim como no exercício 1 da aula 7, as contas criadas devem ser armazenadas em um *ArrayList*. Neste exercício, porém, o usuário deve selecionar o tipo de conta que deseja criar: *ContaPoupanca* ou *ContaEspecial* (utilizar as classes criadas no exercício 1 da aula 4). Assim, deve ser utilizado um *RadioGroup* com *RadioButtons* para indicar o tipo de conta a ser criada. Note que *ContaPoupanca* não tem o atributo *limite*. Se for selecionado o tipo *ContaPoupanca*, o campo limite deve ficar desabilitado na tela (dica: você terá de programar os eventos dos *RadioButtons*).
2. Faça uma janela para consultar saldo, idêntica a que foi feita no exercício 3 da aula 7. Crie um menu na janela do exercício 1 da aula 8 e programe o evento de clique nesse menu para exibir a janela de saldo.

Aula 9 – Integração com Banco de Dados – parte I

Objetivos

Compreender o conceito de programação cliente-servidor.

Aprender a acessar bancos de dados em Java utilizando *drivers* nativos e ODBC.

Aprender a construir aplicações em Java capazes de consultar dados de bancos de dados.

9.1 Programação cliente-servidor

Originalmente, o termo cliente-servidor era usado para descrever *softwares* que se utilizavam de mais de um *hardware* em sua execução. Atualmente, porém, é comum utilizar este termo para caracterizar situações nas quais diferentes componentes de *software* se comunicam uns com os outros, mesmo que rodando em uma mesma máquina. Nesse contexto podemos ter arquiteturas cliente-servidor nas quais os componentes de *software* executam em máquinas espalhadas ao redor do mundo e se comunicam por uma arquitetura de rede ou, em outro extremo, podemos ter arquiteturas cliente-servidor nas quais todos os componentes de *software* executam em uma mesma máquina. Vale ressaltar, que, em ambas as situações, cada componente de *software* executa uma função independente e se comunica com os demais componentes.

Um uso comum para arquiteturas cliente-servidor é manter no lado cliente o gerenciamento das interfaces gráficas com o usuário e utilizar o servidor para manter a integridade dos dados do negócio havendo diversas variações para essa abordagem.

Uma classificação muito comumente utilizada para designar a filosofia utilizada em uma arquitetura cliente-servidor define dois tipos de arquitetura:

- Cliente pesado: quando a interface com o usuário e toda a lógica de negócio estão no lado cliente, ficando o servidor dedicado apenas ao armazenamento, acesso e distribuição de dados.
- Servidor pesado: quando o cliente é responsável apenas pela interface com o usuário, ficando sob responsabilidade do servidor a maior parte da lógica de negócio e o gerenciamento dos dados. Para implementar uma arquitetura desse tipo, é necessário que se faça uma separação entre a porção de código que trata da interface com o usuário e a porção que implementa as regras de negócio.

A forma de programação que utilizaremos nesta disciplina pode ser enquadrada como “cliente pesado” pois desenvolveremos programas que ficarão responsáveis pela interface com o usuário e toda a lógica de negócio (cliente) e que acessarão os dados armazenados em servidores de bancos de dados (servidor).

De fato, nos exemplos e exercícios que faremos nesta aula, nossos programas serão executados na mesma máquina que o servidor de banco de dados. Porém, veremos que para colocar o servidor de banco de dados para funcionar em uma máquina separada, precisaríamos apenas de uma estrutura de rede entre as duas máquinas e seria necessário alterar uma linha de código que define o endereço do servidor.

9.2 Acesso a Bancos de Dados em Java

A linguagem Java possui um conjunto de classes para acesso e manipulação de dados em banco de dados. Essas classes ficam no pacote JDBC.

O primeiro passo para acessar um banco pela aplicação é estabelecer a conexão. Para isso, é necessário utilizar um *driver* que estabelecerá a conexão com o banco. Em ambientes Windows temos basicamente dois tipos de *drivers* para conexão com banco de dados:

- **Drivers ODBC:** o Windows possui uma fonte de dados, de forma que, uma vez existente um driver ODBC para um dado banco, fica sob responsabilidade do Windows a comunicação entre o *software* e o banco.
- **Drivers nativos:** são implementações de um *driver* diretamente para uma dada linguagem, no nosso caso, Java.

O Quadro 9.1 exibe um comparativo entre os dois tipos de *drivers* para conexão com banco de dados em Java.

| Quadro 9.1: <i>Drivers</i> nativos x <i>drivers</i> ODBC | |
|--|--|
| Drivers nativos | Drivers ODBC |
| Normalmente, são mais rápidos, pois são otimizados para uma dada plataforma. | Facilitam a troca de banco em ambiente Windows (MySQL para Sybase por exemplo), pois basta trocar a fonte de dados no Windows que o programa atuará sobre o novo banco desejado. |
| Um programa feito em Java utilizando um <i>driver</i> nativo é migrado para outro SO (<i>Windows</i> para Linux, por exemplo) apenas recompilando os códigos fonte existentes (obviamente esse outro SO deverá possuir a JVM e o <i>driver</i> nativo para o banco desejado). | Se for necessário migrar para outro sistema operacional que não seja Windows, obviamente, não será possível continuar utilizando o ODBC. |

Fonte: Elaborado pelos autores

9.3 Fontes de dados ODBC

Se optarmos por utilizar um banco de dados com um *driver* ODBC, precisaremos, inicialmente, configurar a fonte de dados ODBC, de forma que, seja criado um *Data Source* e que por este possa ser feita a conexão em nossa aplicação.

Para criarmos o *Data Source* de um banco, no Windows XP, por exemplo, precisamos ir em Painel de Controle->Ferramentas Administrativas->Fontes de Dados (ODBC) (Figura 9.1).



Figura 9.1: Fonte de dados ODBC

Fonte: Windows XP Professional

Na janela apresentada, clicando em Adicionar, temos a janela apresentada pela Figura 9.2, na qual devemos selecionar o driver de acordo com o servidor de banco de dados a ser utilizado.

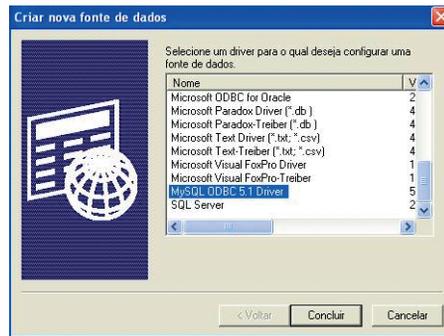


Figura 9.2: Adicionando fonte de dados

Fonte: Windows XP Professional



Utilizaremos nesta disciplina o sistema gerenciador de banco de dados (SGBD) MySQL. Escolhemos esse SGBD por ser gratuito e pelo fato de ter sido utilizado na disciplina Banco de Dados deste curso. Assim, é necessário que a máquina a ser utilizada para implementar os exemplos e exercícios desta aula tenha o MySQL instalado e configurado.

Como vamos incluir uma fonte de dados MySQL ODBC 5.1 *Driver*, devemos selecionar essa opção e clicar em Concluir. Então será exibida a janela apresentada pela Figura 9.3.



Figura 9.3: Configurando um *DataSource MySQL*

Fonte: MySQL Conector/ODBC para Windows XP

Na Figura 9.3 temos a configuração da Fonte de Dados ODBC. Para finalizar a configuração, devemos definir:

- *Data Source*: nome que identificará o banco em nossos programas Java (ou em outra linguagem que faça uso dessa tecnologia).
- *Description*: apenas uma descrição acerca da fonte de dados configurada.

- *Server*: IP da máquina onde está o banco que se deseja conectar. Caso o servidor de banco de dados seja a mesma máquina em que a aplicação rodará, podemos preencher esse campo com *localhost* ou com o IP *127.0.0.1* ou ainda com o endereço IP da máquina.
- *Port*: porta por onde será feita a comunicação (a porta utilizada pelo MySQL por padrão é a 3306).
- *User*: nome do usuário que irá conectar ao banco de dados. Note que não se trata do usuário do sistema operacional e sim do usuário do banco de dados (usuário que você utilizava para acessar o MySQL na disciplina Banco de Dados).
- *Password*: senha do usuário que irá conectar ao banco de dados.
- *Database*: irão aparecer no *combobox* os bancos de dados existentes no servidor informado, de forma que possa ser escolhido o banco de dados desejado.

Agora basta clicar em Ok que o *Data Source* será criado.

9.4 Conectando com o Banco de Dados

Como vimos, aplicativos Java podem se conectar com um banco de dados utilizando *drivers* nativos ou por fontes de dados ODBC (FUGIERI, 2006, p.2 97). Em ambos os casos algumas informações são necessárias para o estabelecimento da conexão:

- O nome do usuário do banco de dados e sua senha.
- O *driver* utilizado é nativo ou ODBC?
- Caso seja nativo:
 - IP da máquina onde está o banco que se deseja conectar.
 - A porta a ser utilizada pela conexão.
 - O nome do banco de dados.

- Caso seja ODBC:
 - *Data Source* do banco: identificação do banco no Windows (fonte de dados ODBC).

Como primeiro exemplo, a Figura 9.4 exibe um código que estabelece uma conexão com um banco de dados utilizando um *driver* ODBC.

```
package newpackage;
import java.sql.*;
public class TesteConexao {
    public static void main(String[] args) {
        String url;
        url = "jdbc:odbc:Banco_LP";
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection minhaConexao = DriverManager.getConnection(url, "root", "");
            System.out.println("Conexão realizada com sucesso");
            minhaConexao.close();
        } catch ( ClassNotFoundException erro ) {
            System.out.println("Driver MySQL não encontrado");
        } catch ( SQLException erro ) {
            System.out.println("Problemas na conexao com banco de dados");
        }
    }
}
```

Figura 9.4: Conexão com banco de dados via ODBC

Fonte: Elaborada pelos autores

Avaliemos alguns trechos principais desse código.

```
String url;
url = "jdbc:odbc:Banco_LP";
try{
```

Figura 9.5: Definição do *Data Source* a ser utilizado

Fonte: Elaborada pelos autores

Como exibe a Figura 9.5, foi definida uma *String* com nome *url* que faz referência ao nome do *Data Source* (fonte de dados ODBC): Banco_LP.

Já a Figura 9.6 exibe o trecho de código que utiliza o método *Class.forName* para verificar se o *driver* ODBC existe na máquina (nas máquinas com Windows praticamente sempre haverá). Caso não exista o *driver*, será disparada a exceção *ClassNotFoundException*.

```
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Figura 9.6: Verificação da existência do *driver* ODBC

Fonte: Elaborada pelos autores

No código exibido pela Figura 9.7 é feita a conexão com o banco, utilizando para isso o método `DriverManager.getConnection` passando como parâmetros a `url` contendo o nome do `data source`, o nome do usuário a ser utilizado para acessar o banco e a senha desse usuário.

```
Connection minhaConexao = DriverManager.getConnection(url, "root", "");
```

Figura 9.7: Estabelecimento da conexão com o banco

Fonte: Elaborada pelos autores

Finalizando nossa análise sobre o código da Figura 9.4, vemos que a tentativa de conexão está envolta em um bloco `try...catch`. Assim, em caso de erro na conexão (`data source` inexistente, usuário ou senha inválidos, entre outros) será disparada uma exceção do tipo `SQLException`. Caso não ocorra erro, é exibida uma mensagem e a conexão é fechada utilizando o método `close`: `minhaConexao.close()`;

Vamos agora exemplificar como fazer uma conexão utilizando um `driver` nativo.

Para utilizar um `driver` nativo precisamos acrescentá-lo ao projeto como uma biblioteca, uma vez que, ele foi compilado separadamente. Para fazer isso, no NetBeans, devemos clicar no projeto com o botão direito do `mouse`, escolher a opção `Propriedades` e nela escolher a opção `Bibliotecas`. Então o arquivo do `driver` deve ser escolhido e adicionado ao projeto. A Figura 9.8 ilustra essa funcionalidade.

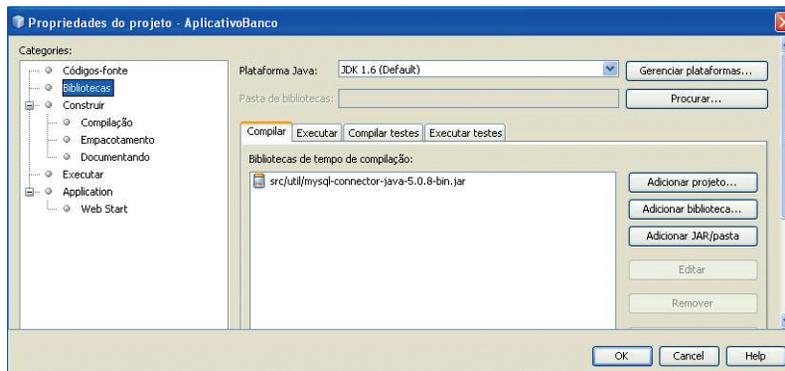


Figura 9.8: Acrescentando uma biblioteca de `driver` nativo ao projeto

Fonte: NetBeans IDE 7.0.1

Uma vez o `driver` nativo adicionado ao projeto, podemos utilizá-lo para criar nossa conexão. O programa exibido pela Figura 9.9 ilustra a conexão com um banco de dados nativo.

```

package newpackage;
import java.sql.*;
public class TesteConexao {
    public static void main(String[] args) {
        String url;
        url = "jdbc:mysql://localhost:3306/banco_lp";
        try
        {
            Class.forName("com.mysql.jdbc.Driver");
            Connection minhaConexao = DriverManager.getConnection(url, "root", "")
            System.out.println("Conexão realizada com sucesso");
            minhaConexao.close();
        } catch ( ClassNotFoundException erro ){
            System.out.println("Driver MySQL não encontrado");
        }
        catch ( SQLException erro) {
            System.out.println("Problemas na conexao com banco de dados");
        }
    }
}

```

Figura 9.9: Criando uma conexão com banco de dados via *driver* nativo

Fonte: Elaborada pelos autores

Note que o código é muito semelhante ao que utilizamos para conectar a um banco de dados utilizando um *driver* ODBC.

Na *url*, onde havíamos informado o nome do *Data Source*, agora tivemos que informar o nome do servidor, a porta a ser utilizada e o nome do banco. Já a classe buscada pelo *Class.forName* agora é a do *driver* MySQL nativo. O restante do código continua a mesma coisa.

9.5 Consultando dados no banco

Para uma aplicação Java buscar dados num banco de dados, precisaremos de uma sequência de ações:

- conectar com o banco;
- escrever a consulta que desejamos que seja feita no banco;
- submeter essa consulta;
- ler os dados do *resultset* (resultado da consulta).



A consulta deve ser escrita na linguagem SQL, que é a linguagem padrão utilizada para manipular dados em bancos de dados relacionais. A linguagem SQL não faz parte da ementa dessa disciplina, pois foi objeto de estudo na disciplina Banco de Dados. Assim, é fundamental que este assunto seja revisado.

As Figuras 9.10, 9.11 e 9.12 exibem um exemplo de aplicação que se conecta a um banco de dados, consulta todos os dados da tabela “filme” e imprime na tela o “código”, o “gênero”, a “produtora” e a “data da compra” de todos os filmes dessa tabela.

O exemplo foi dividido em três Figuras (9.10, 9.11 e 9.12), mas as três figuras juntas apresentam o código de uma única classe.



```
package bancodadosexemplosimples;
import java.sql.*;
public class ConsultaSQL {
    Statement meuState;

    public static void main(String argsp[]) {
        ConsultaSQL obj = new ConsultaSQL();
        obj.buscarDados();
    }
}
```

Figura 9.10: Definição da classe e do método main

Fonte: Elaborada pelos autores

```
public void buscarDados() {
    ResultSet rs;
    String sql = "SELECT * FROM FILME";
    try {
        rs = meuState.executeQuery(sql);
        while (rs.next()) {
            try {
                System.out.println(rs.getString("CODIGO"));
                System.out.println(rs.getString("GENERO"));
                System.out.println(rs.getString("PRODUTORA"));
                System.out.println(rs.getString("DATACOMPRA"));

            } catch (SQLException erro) {
                System.out.println("Erro na leitura de dados da consulta");
            }
        }
    } catch (SQLException erro) { System.out.println("Comando SQL inválido");}
}
}
```

Figura 9.11: Construtor da classe

Fonte: Elaborada pelos autores

```
ConsultaSQL() {
    String url = "jdbc:odbc:banco_lp";
    try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection minhaConexao = DriverManager.getConnection(url, "root", "");
        this.meuState = minhaConexao.createStatement();
    } catch (ClassNotFoundException erro) {
        System.out.println("Driver JDBC-ODBC não encontrado");
    } catch (SQLException erro) {
        System.out.println("Problemas na conexao com banco de dados");
    }
}
}
```

Figura 9.12: Execução da consulta

Fonte: Elaborada pelos autores

Nesse exemplo, o construtor da classe estabelece a conexão com o banco de dados.

O método *main* cria uma instância da classe utilizando seu construtor e aciona então o método *buscarDados()*. Nesse método, a consulta sql é definida em uma *String*, e é executada pelo método *meuState.executeQuery(sql)*. Note que *meuState* é o objeto de conexão criado anteriormente dentro do construtor.

Ainda no método *buscarDados()*, após a execução da consulta é gerado um *ResultSet* que, nada mais é do que o resultado da consulta submetida ao banco. A partir do *ResultSet* é possível obter os dados resultantes da consulta, navegando linha a linha pelo método *rs.next()*. Note que o método *rs.next()* é utilizado como condição de um laço *while*. O comando *while (rs.next())* é equivalente a dizer: enquanto tivermos mais uma linha, permaneça no laço. Na primeira vez esse comando marca a primeira linha para que os dados sejam buscados a partir dela; uma nova chamada de *next()* marca a linha seguinte, até que não haja mais linhas.

Dentro do laço *while* há quatro comandos de impressão *println* com o objetivo de imprimir os dados de cada coluna na tela de saída. Nesse contexto, merece destaque o comando *rs.getString(<nome_da_coluna_no_banco>)*. Esse comando retorna o valor da linha atual para a coluna passada como parâmetro. Por exemplo, o comando *rs.getString("CODIGO")* vai retornar uma *String* com o valor da coluna "CODIGO" da linha atual da tabela.



Para esse exemplo funcionar será necessário uma sequência de passos:

- criar um banco de dados no MySQL com o nome *banco_lp* (pode usar outro nome, mas, nesse caso, precisará adequar o *script* apresentado na Figura 9.13);
- criar dentro desse banco uma tabela com o nome "FILME" utilizando o código SQL apresentado na Figura 9.13;
- inserir algumas linhas nessa tabela para que tenhamos dados para consultar;
- criar uma fonte de dados ODBC com o nome "*banco_lp*" que aponte para o banco criado no primeiro item. Na seção 9.3 aprendemos a criar fontes de dados ODBC.

```
CREATE TABLE banco_lp.filme (  
  codigo INT AUTO_INCREMENT,  
  nome VARCHAR(20),  
  genero VARCHAR(20),  
  produtora VARCHAR(20),  
  data_compra DATETIME,  
  PRIMARY KEY (codigo)  
) ENGINE = InnoDB ROW_FORMAT = DEFAULT;
```

Figura 9.13: Código SQL para criação da tabela *filme*

Fonte: Elaborada pelos autores

Resumo

Nesta aula tivemos uma introdução ao conceito de programação cliente-servidor e aprendemos a acessar servidores de bancos de dados por nossas aplicações em Java. Vimos que, para conectar com os servidores de bancos de dados, utilizamos *drivers* que podem ser de dois tipos: nativos ou ODBC. Por fim, aprendemos a consultar dados em bancos de dados.

Atividades de aprendizagem

1. Para fazer os exercícios desta aula, antes crie um banco de dados com uma tabela *filme*, como a exibida na Figura 9.13 e insira alguns registros nessa tabela utilizando para isso as ferramentas que você aprendeu a utilizar na disciplina de banco de dados.
2. Construa um programa que acesse o banco de dados criado para obter os dados de todos os filmes da tabela e exiba esses dados em uma janela. Para exibir os dados organizadamente, utilize um *JLabel* para exibir o valor de cada campo de forma a exibir os dados em formato tabular. Acesse o banco utilizando *driver* ODBC.
3. Altere o exercício anterior agora utilizando um *driver* nativo para acessar o banco.
4. Construa um programa que apresente uma janela com um campo de texto no qual o usuário deverá digitar o código do filme desejado e um botão com o texto "Consultar". Quando o usuário clicar no botão, o programa deverá buscar no banco de dados os dados do filme do código informado e exibir esses dados na tela.



Existe na biblioteca *Swing* a classe *JTable*, que é própria para exibir dados tabulares em janelas. Assim, uma boa tarefa seria estudar esse componente e utilizá-lo para fazer os exercícios 1 e 2 da aula 9 em lugar de criar vários *JLabels* para exibição dos dados.

Aula 10 – Integração com Banco de Dados – parte II

Objetivos

Aprender a construir aplicações em Java capazes de inserir, alterar e excluir dados de bancos de dados.

10.1 Introdução

Nesta aula criaremos uma classe para encapsular o processo de conexão a banco de dados, a fim de facilitar a implementação de nossos aplicativos. Nessa classe vamos optar pelo uso de conexões utilizando *driver* nativo do MySQL.

À classe criada para nos dar suporte a conexão com bancos de dados daremos o nome de *MySQLNativeDriver*. O código dessa classe é exibido na Figura 10.1.

```
package insercaoDados;
import java.sql.*;
public class MySQLNativeDriver {
    protected String nomeBanco;
    protected String usuario;
    protected String senha;
    private String ip, porta;

    private void setarValores(String nomeBanco, String usuario, String senha, String ip, String porta){
        this.nomeBanco = nomeBanco;
        this.usuario = usuario;
        this.senha = senha;
        this.ip = ip;
        this.porta = porta;
    }

    public MySQLNativeDriver(String nomeBanco, String usuario, String senha, String ip, String porta){
        this.setarValores(nomeBanco, usuario, senha, ip, porta);
    }

    public MySQLNativeDriver(String nomeBanco, String usuario, String senha){
        this.setarValores(nomeBanco, usuario, senha, "localhost", "3306");
    }

    public Connection obterConexao() {
        Connection conexao = null;
        try {
            String url;
            url = "jdbc:mysql://" + this.ip + ":" + this.porta + "/" + this.nomeBanco;
            Class.forName("com.mysql.jdbc.Driver");
            conexao = DriverManager.getConnection(url, this.usuario, this.senha);
        } catch (ClassNotFoundException erro) {
            System.out.println("Classe ODBC não existe");
        } catch (SQLException erro) {
            System.out.println("SQL inválido");
        } finally {
            return conexao;
        }
    }
}
```

Figura 10.1: Classe *MySQLNativeDriver* – parte I

Fonte: Elaborada pelos autores

O método mais importante da classe *MySQLNativeDriver* é o método *obterConexao()* que cria e retorna uma conexão com um banco de dados utilizando o *driver* nativo do MySQL.

Note também que nessa classe temos atributos para definir todos os parâmetros necessários para o estabelecimento de uma conexão com o banco de dados: IP do servidor, porta, nome do banco, nome do usuário e senha.

A classe conta ainda com dois construtores: um que recebe como parâmetros as cinco informações necessárias para a conexão, e um segundo que recebe apenas o nome do banco, o nome do usuário e a senha e assume que o banco está na máquina local (*localhost*) e que a porta a ser utilizada é a porta padrão do MySQL (3306).



É importante lembrar que para fazermos uso do *driver* MySQL nativo precisamos incluí-lo no projeto, como aprendemos na aula anterior (seção 9.4 – Figura 9.8).

Para os exemplos deste capítulo, utilizaremos a mesma tabela “FILME” já criada na aula anterior (Figura 9.13).

Nesta aula faremos inserção, atualização e exclusão de dados nessa tabela. Para isso é importante entendermos o código de criação dessa tabela, apresentado pela Figura 9.13.

Note que a coluna *codigo* é a chave primária da tabela e é uma coluna autoincrementável. Isso significa que ela não precisa participar do *INSERT*, pois esse dado será preenchido, automaticamente, pelo banco, utilizando o valor do último registro inserido mais uma unidade (conforme já estudado na disciplina Banco de Dados).

Na atualização (*UPDATE*) e na exclusão (*DELETE*) a coluna *codigo* identificará de forma única cada registro encontrado na tabela *filme*, possibilitando assim, sua atualização ou exclusão.

10.2 Inserção de dados

A Figura 10.2 ilustra como pode ser feita a inserção de dados na tabela *filme*.

```
package insercaoDados;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class InserindoDados {

    public static void main(String[] args) {
        Connection conexao;
        MySQLNativeDriver driver = new MySQLNativeDriver("banco_lp", "root", "");
        conexao = driver.obterConexao();

        String sql = "INSERT INTO filme ( nome, genero, produtora, data_compra) values ( ?, ?, ?, ?)";
        PreparedStatement pstmt;
        try {
            pstmt = conexao.prepareStatement(sql);
            pstmt.setString(1, "Jogos Mortais");
            pstmt.setString(2, "Terror");
            pstmt.setString(3, "Sony");
            pstmt.setString(4, "2005-03-03");
            pstmt.executeUpdate();
        } catch (SQLException ex) {
            System.out.println("Erro na inserção dos dados (" + ex.getMessage() + ")");
        }
    }
}
```

Figura 10.2: Exemplo de inserção de dados

Fonte: Elaborada pelos autores

Note que utilizamos a classe *MySQLNativeDriver* para obter uma conexão com o banco de dados. Em seguida, montamos o nosso comando *INSERT* (variável *sql*). As interrogações são os locais onde serão inseridos os valores a serem atribuídos aos campos em nosso *INSERT*. Esses valores serão inseridos no comando *INSERT* pelo método *pstmt.setString* posicionalmente a cada interrogação (1 para a primeira interrogação, 2 para a segunda, e assim sucessivamente).

Para entender o funcionamento desse código é fundamental conhecer a sintaxe do comando *INSERT* da linguagem SQL que foi estudada na disciplina Banco de Dados.

Finalmente, o comando *pstmt.executeUpdate()* efetua a inserção dos dados no banco de dados. Note que o código está envolto por uma cláusula *try...catch* de forma que caso ocorra algum erro na inserção, será disparada uma exceção do tipo *SQLException* que será tratada pelo seu respectivo *catch*, que, nesse caso, basicamente irá apresentar uma mensagem de erro.



No caso do nosso exemplo, informamos valores fixos para cada uma das colunas da nossa tabela o que não é muito útil e serve apenas para ilustrar e para facilitar o entendimento. Normalmente, teremos uma interface gráfica na qual o usuário informará o valor desejado para cada campo e clicará em um botão. Nesse caso, o evento de clicar no botão deverá disparar um método que terá código muito parecido com o método *main* do nosso exemplo mas, no lugar dos valores fixos que inserimos em cada campo, buscaremos o valor que o usuário preencheu em cada campo da interface gráfica. Por exemplo, suponha que tenhamos na interface um *JTextField* de nome "nomeFilme" no qual o usuário deverá digitar o nome do filme. Assim, no lugar do comando *pstmt.setString(1, "Jogos Mortais");* do exemplo teríamos o comando *pstmt.setString(1, "nomeFilme.getText());*.



10.3 Atualização de dados

A atualização de dados ocorre de forma semelhante à inserção. A Figura 10.3 ilustra como pode ser feita a atualização de dados.

```
package alteracaoDados;
import insercaoDados.MySQLNativeDriver;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
public class AlterandoDados {
    public static void main(String[] args) {
        Connection conexao;
        MySQLNativeDriver driver = new MySQLNativeDriver("banco_lp", "root", "");
        conexao = driver.obterConexao();

        String sql = " UPDATE filme SET "
            + " nome = ?, "
            + " genero= ?, "
            + " produtora = ?, "
            + " data_compra = ? "
            + " WHERE codigo = ?";

        PreparedStatement pstmt;
        try {
            pstmt = conexao.prepareStatement(sql);
            pstmt.setString(1, "Novo nome");
            pstmt.setString(2, "Novo genero");
            pstmt.setString(3, "Nova produtora");
            pstmt.setString(4, "2005-05-05");
            pstmt.setString(5, "1");
            pstmt.executeUpdate();
        } catch (SQLException ex) {
            System.out.println("Erro na atualização dos dados (" + ex.getMessage() + ")");
        }
    }
}
```

Figura 10.3: Exemplo de atualização de dados

Fonte: Elaborada pelos autores



Para entender o funcionamento desse código, é fundamental conhecer a sintaxe do comando *UPDATE* da linguagem SQL que foi estudada na disciplina Banco de Dados.



Aqui também é importante notar que, em uma aplicação comercial, os valores a serem utilizados no *UPDATE* não serão fixos e sim obtidos a partir de alguma entrada, normalmente, uma interface gráfica. Além disso, antes de uma atualização, normalmente, ocorre uma consulta a fim de que os dados sejam apresentados para que então o usuário os atualize.

Assim como no exemplo de inserção de dados, a classe *MySQLNativeDriver* serve para obter uma conexão com o banco. Em seguida montamos uma *String* com o nosso comando SQL *UPDATE*. Note que, também a exemplo do que ocorreu na inserção, nos lugares dos valores a serem atribuídos aos campos das tabelas foram colocadas interrogações. Note que o campo código foi utilizado na cláusula *where* do seu comando SQL a fim de selecionar o registro desejado para a alteração.

10.4 Exclusão de dados

O código para executar uma exclusão de dados é ainda mais simples que os códigos para atualização e para inserção de dados. A Figura 10.4 ilustra como pode ser feita a atualização de dados.

```
package exclusaoDados;
import insercaoDados.MySQLNativeDriver;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
public class ExcluindoDados {
    public static void main(String[] args) {
        Connection conexao;
        MySQLNativeDriver driver = new MySQLNativeDriver("banco_lp", "root", "");
        conexao = driver.obterConexao();
        String sql = "DELETE FROM filme WHERE codigo = ?";
        PreparedStatement pstmt;
        try {
            pstmt = conexao.prepareStatement(sql);
            pstmt.setString(1, "1");
            pstmt.executeUpdate();
        } catch (SQLException ex) {
            System.out.println("Erro na exclusão dos dados (" + ex.getMessage() + ")");
        }
    }
}
```

Figura 10.4: Exemplo de exclusão de dados

Fonte: Elaborada pelos autores

Assim como nos exemplos anteriores, a classe *MySQLNativeDriver* serve para obter uma conexão com o banco.

Para entender o funcionamento desse código, é fundamental conhecer a sintaxe do comando *DELETE* da linguagem SQL que foi estudada na disciplina Banco de Dados.

Note que na montagem do comando *DELETE* foi utilizada apenas uma interrogação. Isso se deve ao fato de que para excluir um registro basta que tenhamos o valor de sua chave primária, que no caso da nossa tabela *filme* é a coluna *codigo*.



Assim como aconteceu na inclusão e na alteração de dados, esse exemplo é apenas ilustrativo e tem foco nas classes e métodos a serem utilizados. Obviamente, que em uma aplicação comercial o código do elemento a ser excluído seria informado pelo usuário. Normalmente, é feita uma consulta, os dados são apresentados e só então o usuário pode optar por solicitar a exclusão do registro apresentado.



Resumo

Nesta aula demos sequência aos estudos iniciados na aula anterior sobre acesso a bancos de dados pelas aplicações em Java. Aprendemos a inserir, atualizar e excluir dados em bancos de dados. Vimos que, para isso, precisamos utilizar comandos SQL aprendidos na disciplina Bancos de Dados.

Atividades de aprendizagem

1. Crie uma tabela *ContaEspecial* em um banco de dados. Altere o programa criado no exercício 1 da aula 7 de forma que ao invés de armazenar as contas em um *ArrayList*, os dados das contas criadas sejam inseridos nessas tabelas.
2. Faça um programa que exiba uma janela com um campo de texto “número da conta”, um botão “Consultar” e um botão “Excluir”. A janela deve ser exibida com o botão “Excluir” desabilitado. O usuário deve informar o número da conta desejada e clicar no botão consultar. Então o sistema deve buscar os dados da conta desejada no banco de dados, exibir os dados na janela e o botão excluir deve ser habilitado. Se o usuário clicar então em excluir, a conta deve ser excluída do banco e uma mensagem deve ser exibida. Se o usuário informar alterar o número da conta e clicar novamente em consultar, os dados da nova conta informada devem ser exibidos no lugar da anterior. (Dica: utilize um *JTextField* para exibir cada um dos dados da conta).

Referências

CADENHEAD, Rogers; LEMAY, Laura, **Aprenda em 21 dias Java 2**. 4. ed. São Paulo: Campus, 2005.

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

FUGIERI, Sérgio. Java 2 – Ensino Didático. 6ª Edição. São Paulo: Érica, 2006.

PAMPLONA, Vitor Fernando. **Tutorial Java: o que é Java?** Disponível em: <<http://javafree.uol.com.br/artigo/871498/Tutorial-Java-O-que-e-Java.html>>. Acesso em: 12 out. 2011.

PAMPLONA, Vitor Fernando. **Tutorial Java 2<BOLD>: características básicas**. Disponível em : <<http://javafree.uol.com.br/artigo/871496/Tutorial-Java-2-Caracteristicas-Basicas.html>>. Acesso em: 12 out. 2011.

PAMPLONA, Vitor Fernando. **Tutorial Java 3: orientação a objetos**. Disponível em: <<http://javafree.uol.com.br/artigo/871497/Tutorial-Java-3-Orientacao-a-Objetos.html>>. Acesso em: 12 out.2011.

Currículo dos professores-autores



Victorio Albani

Técnico em Processamento de Dados pela Escola Técnica Federal do Espírito Santo (1998), graduado em Ciência da Computação pela UFES (2003) e mestre em Informática pela UFES (2006). É professor do Departamento de Informática do Instituto Federal de Educação, Ciência e Tecnologia do Espírito Santo – Campus Colatina, onde trabalha com os cursos Técnico em Informática, na modalidade presencial e a distância, e Superior de Tecnologia em Redes de Computadores.

Antes de ingressar no IFES, atuou em várias empresas públicas e privadas, dentre as quais se destacam Xerox, Unisys e Cesan, sempre exercendo funções relacionadas ao processo de desenvolvimento de *software*, como programador, analista/projetista de sistemas e líder de equipe de desenvolvimento.



Giovany Frossard Teixeira

Graduado em Ciência da Computação (2004) e mestre em Informática (2006) pela Universidade Federal do Espírito Santo. Atualmente é professor do Instituto Federal de Educação, Ciência e Tecnologia do Espírito Santo – Campus Colatina, nos cursos Técnico em Informática, Tecnólogo em Redes de Computadores e Bacharel em Sistemas de Informação. Tem experiência na área de Ciência da Computação, com ênfase em Linguagens de Programação, Otimização e *Software* Básico.



ISBN 978-85-62934-38-4

