Coleção UAB–UFSCar

Sistemas de Informação

Delano Medeiros Beder

Introdução à Programação Orientada a Objetos em Java

Introdução à Programação Orientada a Objetos em Java

Reitor
Targino de Araújo Filho
Vice-Reitor
Adilson J. A. de Oliveira
Pró-Reitora de Graduação
Claudia Raimundo Reyes

Secretária Geral de Educação a Distância - SEaD
Aline Maria de Medeiros Rodrigues Reali
Coordenação SEaD-UFSCar
Glauber Lúcio Alves Santiago
Marcia Rozenfeld G. de Oliveira
Sandra Abib
Coordenação UAB-UFSCar
Sandra Abib

Coordenadora do Curso de Sistemas de Informação
Vânia Neris

UAB-UFSCar
Universidade Federal de São Carlos
Rodovia Washington Luís, km 235
13565-905 - São Carlos, SP, Brasil
Telefax (16) 3351-8420
www.uab.ufscar.br
uab@ufscar.br

Delano Medeiros Beder

Introdução à Programação Orientada a Objetos em Java

1ª edição
São Carlos, SP
2014

© 2016, Delano Medeiros Beder

Supervisão
Douglas Henrique Perez Pino

Revisão Linguística
Clarissa Galvão Bengtson
Daniel William Ferreira de Camargo

Diagramação
Izis Cavalcanti

Capa e Projeto Gráfico
Luís Gustavo Sousa Sguissardi

Todos os direitos reservados. Nenhuma parte desta obra pode ser reproduzida ou transmitida por qualquer forma e/ou quaisquer meios (eletrônicos ou mecânicos, incluindo fotocópia e gravação) ou arquivada em qualquer sistema de banco de dados sem permissão escrita do titular do direito autoral.

Sumário

APRESENTAÇÃO 11

1 Introdução 23

1.1 Primeiras palavras 25

1.2 Problematizando o tema 25

1.3 Paradigmas de programação 25

1.3.1 Paradigma orientado a objetos: breve histórico 27

1.4 Afinal de contas, o que significa ser orientado a objetos? 28

1.4.1 Abstração de dados 29

1.4.2 Encapsulamento 29

1.4.3 Ocultação de informações e implementações 30

1.4.4 Retenção de estado 31

1.4.5 Identidade de objeto 31

1.4.5.1 Referências para objetos 32

1.4.6 Mensagens 33

1.4.7 Classes/Objetos 34

1.4.8 Herança 34

1.4.9 Classes abstratas, classes concretas e interfaces 36

1.4.10 Polimorfismo 37

1.4.11 Relacionamento entre classes/objetos 38

1.5 Considerações finais 40

1.6 Estudos complementares 40

2 Introdução à linguagem Java 41

2.1 Primeiras palavras 43

2.2 Problematizando o tema 43

2.3 O que é Java? 43

2.3.1 Breve histórico 43

2.3.2 Máquina Virtual Java 44

2.3.3 Ambiente de desenvolvimento 45

2.3.3.1 Configuração do ambiente de desenvolvimento . . . 46

2.3.4 Coletor de lixo 47

2.3.5 Primeiro programa Java 48

2.3.5.1 Compilando o programa 48

2.3.5.2 Executando o programa 48

2.4 Tipos primitivos & classes *Wrappers* 49

2.5 Controle de fluxo 50

2.5.1 Comando de seleção: `if-else` 50

2.5.2 Comando de seleção: `switch` 52

2.5.3 Comando de repetição: `while` 53

2.5.4 Comando de repetição: `do..while` 54

2.5.5 Comando de repetição: `for` 55

2.5.6 Comando `break` 55

2.5.7 Comando `continue` 56

2.6 Orientação a objetos em Java: conceitos básicos 57

2.6.1 Atributos 58

2.6.2 Métodos 59

2.6.3 Modificadores de acesso 61
2.6.4 *Getters* e *Setters* 62
2.6.5 Construtores de classes 63
2.6.6 Instanciando classes 64
2.6.7 Atributos e métodos estáticos 65
2.6.8 Palavra reservada `final` 66
2.7 Considerações finais 67
2.8 Estudos complementares 68

3 Conceitos avançados de orientação a objetos em Java 69

3.1 Primeiras palavras 71
3.2 Problematizando o tema 71
3.3 Arrays 71
3.3.1 Percorrendo um *array* 72
3.3.2 Métodos com argumentos variáveis 73
3.4 Pacotes 74
3.4.1 Pacote `java.lang` 75
3.4.1.1 `String` 76
3.4.1.2 `String`, `StringBuider` e `StringBuffer` 77
3.5 Associação, Composição e Agregação 78
3.6 Herança 79
3.7 Classes abstratas 81
3.7.1 Classe `ObraDeArte` 82
3.7.2 Classe `Pintura` 83
3.7.3 Classe `Escultura` 84
3.7.4 Classe `Museu` 85
3.8 Interfaces 87

3.8.1 Interface **Figura2D** 87
3.8.2 Classe **Retângulo** 88
3.8.3 Classe **Triângulo** 89
3.8.4 Classe **Equilátero** 90
3.9 Polimorfismo paramétrico 91
3.10 Exceções 93
3.10.1 Tratamento de exceções 93
3.10.2 Erro e **RuntimeException** 95
3.10.3 Propagando exceções 96
3.10.4 Definindo e sinalizando exceções 96
3.11 Considerações finais 97
3.12 Estudos complementares 98

4 Entrada e saída em Java **99**
4.1 Primeiras palavras 101
4.2 Problematizando o tema 101
4.3 Pacote **java.io** 101
4.4 Manipulação de arquivos 102
4.4.1 Acesso não sequencial 104
4.5 Leitura e escrita sequencial de bytes 106
4.5.1 Leitura de *streams* de bytes 106
4.5.2 Escrita de *streams* de bytes 109
4.5.3 Classes **Scanner** e **PrintStream** 111
4.5.4 Serialização de objetos 112
4.6 Leitura e escrita sequencial de caracteres 115
4.6.1 Leitura de *streams* de caracteres 115
4.6.2 Escrita de *streams* de caracteres 117

4.7 Considerações finais 118
4.8 Estudos complementares 118

5 Coleções em Java 119

5.1 Primeiras palavras 121
5.2 Problematizando o tema 121
5.3 *Framework* de coleções 121
5.3.1 Principais interfaces do *framework* de coleções 123
5.4 Listas 125
5.4.1 Relacionamento *para muitos* entre classes 127
5.4.2 Iterando sobre coleções com `java.util.Iterator` 129
5.5 Ordenação 130
5.6 Conjuntos 132
5.6.1 Sistema de gerenciamento de obras de arte 134
5.7 Filas 136
5.7.1 Filas de prioridades 138
5.8 Mapas 140
5.8.1 Frequência de letras em um texto 143
5.9 Algoritmos 145
5.10 Considerações finais 146
5.11 Estudos complementares 146

6 Interface gráfica em Java 147

6.1 Primeiras palavras 149
6.2 Problematizando o tema 149
6.3 AWT – *Abstract Windowing Toolkit* 149
6.3.1 Interfaces gráficas com usuários 150
6.3.2 Componentes gráficos 150

6.3.3 Gerenciadores de leiautes 155
6.3.4 Eventos 159
6.4 Swing 162
6.4.1 Componentes gráficos 163
6.4.2 Editor de texto 168
6.5 Considerações finais 173
6.6 Estudos complementares 174

7 Acesso a banco de dados em Java 175

7.1 Primeiras palavras 177
7.2 Problematizando o tema 177
7.3 Modelo de dados relacional 178
7.3.1 `SQL` 180
7.3.2 JDBC 184
7.3.2.1 *Drivers* `JDBC` 184
7.3.2.2 Conexão com banco de dados 185
7.3.2.3 Execução da consulta 185
7.3.2.4 Aplicação `ListaObrasDeArte` 187
7.4 Mapeamento objeto-relacional 189
7.4.1 *Data Access Object* 189
7.4.2 Sistema de gerenciamento de obras de arte 190
7.4.2.1 Classe `ConnectionFactory` – Fábrica de conexões . 191
7.4.2.2 Classe `ObjectDTO` 192
7.4.2.3 Interface `IGenericDAO` 192
7.4.2.4 Interface `IObraDeArteDAO` 193
7.4.2.5 Classe `GenericJDBCDAO` 194
7.4.2.6 Classe `ObraDeArteJDBCDAO` 196

7.4.2.7 Fábrica de *DAOs* 200
7.4.2.8 Aplicação **ImprimeObrasDeArte** 201
7.5 Considerações finais 202
7.6 Estudos complementares 202

8 Desenvolvimento Web em Java 203

8.1 Primeiras palavras 205
8.2 Problematizando o tema 205
8.3 Desenvolvimento Web 205
8.3.1 Protocolo **HTTP** 206
8.3.2 **HTML** 207
8.3.3 Páginas dinâmicas 209
8.3.4 Configuração do ambiente de desenvolvimento 210
8.4 *Applets* 211
8.4.1 Ciclo de vida de um *applet* 212
8.4.2 O primeiro exemplo de um *applet* 213
8.4.3 Carregando um *applet* 214
8.4.4 Classe **JApplet** 215
8.5 *Servlets* 217
8.5.1 Classe **HttpServlet** 217
8.5.2 Ciclo de vida de um *servlet* 218
8.5.3 Interface **HttpServletRequest** 219
8.5.4 Atributos 220
8.5.5 Interface **HttpServletResponse** 221
8.5.6 Exemplo: Conversão de temperaturas 222
8.5.6.1 Arquivo **index.html** 222
8.5.6.2 Arquivo **web.xml** 223

8.5.6.3 Classe **ServletConversão** 224

8.5.6.4 Classe **ServletCelsiusFahrenheit** 226

8.5.6.5 Classe **ServletCelsiusKelvin** 227

8.6 *Java Server Pages* 228

8.6.1 Ciclo de vida de um *JSP* 229

8.6.2 Objetos implícitos 230

8.6.3 Exemplo: conversão de temperaturas 231

8.6.3.1 Arquivo **index.html** 231

8.6.3.2 Arquivo **web.xml** 232

8.6.3.3 Página **JSP** – **conversão.jsp** 233

8.6.3.4 Página **JSP** – **fahrenheit.jsp** 234

8.6.3.5 Página **JSP** – **kelvin.jsp** 235

8.7 Considerações finais 236

8.8 Estudos complementares 236

APRESENTAÇÃO

É indiscutível que orientação a objetos é o paradigma de programação mais utilizado na atualidade e que a linguagem Java é, entre as linguagens de programação que adotam esse paradigma, uma das mais populares.

Nesse contexto, este livro foi escrito com o objetivo de lhe oferecer uma introdução à linguagem Java ressaltando como os principais conceitos inerentes ao paradigma orientado a objetos foram definidos nessa linguagem. O livro é composto de oito unidades, conforme descrito a seguir:

- A Unidade 1 apresenta uma breve revisão dos conceitos inerentes ao paradigma orientado a objetos.
- A Unidade 2 apresenta uma breve introdução à linguagem de programação Java.
- A Unidade 3 discute como conceitos mais avançados do paradigma orientado a objetos, tais como classe abstrata, interface, pacotes e tratamento de exceções, são definidos na linguagem de programação Java.
- A Unidade 4 apresenta os conceitos relacionados às operações de entrada e saída na linguagem de programação Java.
- A Unidade 5 apresenta os conceitos relacionados ao *framework* de coleções presente na linguagem de programação Java.
- A Unidade 6 apresenta os conceitos relacionados ao desenvolvimento de interfaces gráficas (*Desktop*) na linguagem de programação Java.
- A Unidade 7 apresenta os conceitos relacionados às operações de acesso a banco de dados na linguagem de programação Java.
- E, por fim, a Unidade 8 apresenta os conceitos básicos relacionados ao desenvolvimento de aplicações Web na linguagem de programação Java.

Lista de Figuras

1.1 Paradigmas. 27
1.2 Diagrama de classes UML: classe Conta. 29
1.3 Ocultação de informações e implementações. 30
1.4 Identidade de objetos. 32
1.5 Identidade de objetos: objeto inacessível. 33
1.6 Mensagens. 33
1.7 Diagrama de classes UML: classe ContaChequeEspecial. 34
1.8 Diagrama de classes UML: Hierarquia de classes. 36
1.9 Polimorfismo. 38
1.10 Diagrama de classes UML: relacionamento de associação. 39
1.11 Diagrama de classes UML: relacionamento de composição. 39
1.12 Diagrama de classes UML: relacionamento de agregação. 39

2.1 Compilando o primeiro programa Java. 48
2.2 Executando o primeiro programa Java. 49
2.3 *Autoboxing* e *auto-unboxing* em Java. 50
2.4 Exemplo da execução da classe **`ExemploIf`**. 51
2.5 Exemplo da execução da classe **`ExemploSwitch`**. 52
2.6 Exemplo da execução da classe **`ExemploWhile`**. 53
2.7 Exemplo da execução da classe **`ExemploDoWhile`**. 54

2.8 Exemplo da execução da classe `ExemploFor`. 55

3.1 Gerenciamento de obras de arte de um museu. 82

3.2 Interface `Figura2D` e classes que a implementam. 87

5.1 Principais interfaces do *framework* de coleções Java. 123

5.2 Listas – hierarquia de classes e interfaces. 126

5.3 Execução da classe `Pessoa`. 129

5.4 Conjuntos – hierarquia de classes e interfaces. 132

5.5 Execução da classe `Museu`. 135

5.6 Filas – hierarquia de classes e interfaces. 137

5.7 Fila de prioridades. 139

5.8 Mapas – hierarquia de classes e interfaces. 140

6.1 Hierarquia de componentes gráficos presentes na biblioteca `AWT`. . . . 151

6.2 Componentes gráficos `AWT` – classe `Button`. 152

6.3 Componentes gráficos `AWT` – classe `List`. 153

6.4 Componentes gráficos `AWT` – classe `Checkbox`. 153

6.5 Componentes gráficos `AWT` – classe `CheckboxGroup`. 153

6.6 Componentes gráficos `AWT` – campos de texto. 154

6.7 Componentes gráficos `AWT` – menus. 155

6.8 Gerenciador de leiaute `BorderLayout`. 157

6.9 Gerenciador de leiaute `CardLayout`. 157

6.10 Tratamento de eventos `AWT`. 161

6.11 Hierarquia de componentes gráficos presentes na biblioteca `Swing`. . . 163

6.12 Componentes gráficos `Swing` – classe `JButton`. 165

6.13 Componentes gráficos `Swing` – classe `JComboBox`. 165

6.14 Componentes gráficos `Swing` – classe `JCheckBox`. 165

6.15 Componentes gráficos **Swing** – classe **JRadioButton**. 166

6.16 Componentes gráficos **Swing** – campos de texto. 167

6.17 Componentes gráficos **Swing** – menus. 168

6.18 Aplicação gráfica **Swing** – editor de texto. 170

7.1 Tuplas de um banco de dados relacional. 178

8.1 Distribuições do NetBeans IDE 7.4. 210

8.2 Ciclo de vida de um *applet*. 212

8.3 Execução do **AppletDemo** no navegador Web. 214

8.4 Execução do **JAppletDemo** no navegador Web. 215

8.5 Ciclo de vida dos *servlets*. 218

8.6 Invocação do *servlet* com dados inválidos. 225

8.7 Conversão de temperaturas (°C para °F). 226

8.8 Conversão de temperaturas (°C para °K). 227

8.9 Ciclo de vida de páginas *JSP*. 230

Lista de Tabelas

2.1 Tipos primitivos de Java e classes *Wrappers*. 49

2.2 Palavras reservadas em Java. 58

2.3 Valores *default* dos atributos. 63

7.1 Saída da execução da classe `ListaObrasDeArte`. 187

Lista de Códigos

2.1 Primeiro programa Java. 48

2.2 Exemplo do comando de seleção `if-else`. 51

2.3 Exemplo do comando de seleção `switch`. 52

2.4 Exemplo do comando de repetição `while`. 53

2.5 Exemplo do comando de repetição `do..while`. 54

2.6 Exemplo do comando de repetição `for`. 55

2.7 Exemplo do comando `break`. 56

2.8 Exemplo do comando `continue`. 56

2.9 Classe `Conta` – apenas definição da classe. 59

2.10 Classe `Conta` com atributos e métodos. 61

2.11 Classe `AcessaConta` que manipula objetos da classe `Conta`. 64

2.12 Classe `Conta` – Implementação final. 65

2.13 Exemplo: classe final. 66

2.14 Exemplo: método final. 66

2.15 Exemplo: atributos finais. 67

3.1 Percorrendo *array*: utilizando o atributo `length`. 73

3.2 Percorrendo *array*: **enhanced-for**. 73

3.3 Método com argumentos variáveis. 73

3.4 Classes `Carro` e `Pessoa`. 79

3.5 Classe `ContaChequeEspecial` com atributos e métodos. 80

3.6 Classe abstrata `ObraDeArte`. 83
3.7 Classe `Pintura`. 84
3.8 Classe `Escultura`. 85
3.9 Classe `Museu`. 86
3.10 Interface `Figura2D`. 88
3.11 Classe `Retângulo`. 89
3.12 Classe `Triângulo`. 90
3.13 Classe `Equilátero`. 91
3.14 Polimorfismo paramétrico – listas genéricas. 92
3.15 Tratamento de exceções em Java. 94
3.16 Definindo e sinalizando exceções. 97
4.1 Lista de arquivos e diretórios de um diretório. 104
4.2 Acesso não sequencial a arquivos. 105
4.3 Leitura de um *stream* de bytes. 108
4.4 Leitura e escrita de *streams* de bytes. 110
4.5 Classe `Scanner` – leitura pelo teclado (ou arquivo). 111
4.6 Classe `PrintStream` – escrita para arquivo (ou console). 112
4.7 Serialização e desserialização de objetos. 114
4.8 Leitura de um *stream* de caracteres. 116
4.9 Leitura e escrita de *streams* de caracteres. 118
5.1 Classe `Carro`. 127
5.2 Classe `Pessoa`. 128
5.3 Classe `Carro` – implementando a interface `Comparable`. 130
5.4 Método `imprimeCarros()` – impressão em ordem alfabética. 131
5.5 Classe `ObraDeArte` – implementando a interface `Comparable`. 135
5.6 Classe `Museu` – impressão em ordem alfabética. 136

5.7 Classe **Processo** – escalonamento de processos. 138

5.8 Mapas – frequência de letras. 144

5.9 Classe **Collections** – métodos polimórficos. 145

6.1 Classes manipuladoras de eventos. 162

6.2 Classe *handler* de eventos do teclado. 167

6.3 Classe **Editor**. 169

6.4 Classe **FileUtil**. 171

6.5 Classe **Handler**. 172

7.1 Comandos SQL. 183

7.2 Classe **ListaObrasDeArte**. 188

7.3 Classe **ConnectionFactory** – Fábrica de conexões. 191

7.4 Classe **ObjectDTO**. 192

7.5 Interface **IGenericDAO**. 193

7.6 Interface **IObraDeArteDAO**. 194

7.7 Classe **GenericJDBCDAO**. 195

7.8 Classe **ObraDeArteJDBCDAO**. 198

7.9 Classe **ObraDeArteJDBCDAO** – Continuação. 199

7.10 Classe **DAOFactory**. 200

7.11 Classe **JDBCDAOFactory**. 200

7.12 Classe **ImprimeObrasDeArte**. 201

8.1 Classe **AppletDemo**. 213

8.2 **AppletDemo.html**. 215

8.3 Classe **JAppletDemo**. 216

8.4 Arquivo **index.html**. 222

8.5 Arquivo **web.xml**. 223

8.6 Classe **ServletConversão**. 225

8.7 Classe `ServletCelsiusFahrenheit`. 226
8.8 Classe `ServletCelsiusKelvin`. 227
8.9 Arquivo `index.html`. 232
8.10 Arquivo `web.xml`. 232
8.11 Página `JSP` – `conversão.jsp`. 233
8.12 Página `JSP` – `fahrenheit.jsp`. 234
8.13 Página `JSP` – `kelvin.jsp`. 235

UNIDADE 1

Introdução

1.1 Primeiras palavras

É indiscutível que orientação a objetos é o paradigma de programação mais utilizado na atualidade e que a linguagem Java é, entre as linguagens de programação que adotam esse paradigma, uma das mais populares.

Nesse contexto, este livro foi escrito com o objetivo de lhe oferecer uma introdução à linguagem Java ressaltando como os principais conceitos inerentes ao paradigma orientado a objetos foram definidos nessa linguagem.

É importante deixar uma ressalva ao leitor. Este livro foi escrito para ser adotado como material de apoio à disciplina de *Programação Orientada a Objetos II* do curso de Bacharelado de Sistemas de Informação da Universidade Federal de São Carlos. Dessa forma, é pressuposto que o leitor já tem algum conhecimento sobre o paradigma orientado a objetos e que, apesar de esta unidade apresentar uma discussão sobre os principais conceitos inerentes a esse paradigma, este livro não deveria ser considerado um material de referência sobre o paradigma. Para mais detalhes sobre tais conceitos, sugere-se que o leitor consulte as referências listadas na Seção 1.6.

1.2 Problematizando o tema

Ao final desta unidade, espera-se que o leitor seja capaz de reconhecer, distinguir e definir precisamente os conceitos básicos relacionados ao paradigma orientado a objetos. Dessa forma, esta unidade pretende discutir as seguintes questões:

- O que é um paradigma de programação?
- Quais as principais características do paradigma orientado a objetos?

1.3 Paradigmas de programação

Um paradigma de programação fornece e determina a visão que o programador possui sobre a estruturação e execução do programa. Ou seja, um paradigma de programação é um modelo, padrão ou estilo de programação suportado por linguagens que agrupam certas características comuns (SEBESTA, 2003).

De maneira bastante simplista, podemos definir os paradigmas de programação clássicos da seguinte forma:

- **Imperativo**: o computador é uma calculadora programável.
  - Linguagens representativas: C e Pascal.
- **Orientado a objetos (OO)**: o mundo consiste de objetos, e o software simula o mundo real.
  - Linguagens representativas: C++, Java, Eiffel e Smalltalk.
- **Funcional**: o computador calcula funções matemáticas.
  - Linguagens representativas: LISP e Haskell.
- **Lógica**: o computador entende a lógica formal (fatos e regras).
  - Linguagens representativas: Prolog e Godel.

A Figura 1.1 apresenta um breve histórico da evolução das linguagens de programação clássicas, bem como sua classificação de acordo com os paradigmas de programação clássicos. Vale a pena salientar que esse histórico não apresenta as linguagens de programação modernas (criadas a partir do ano 2000), pois estas geralmente suportam múltiplos paradigmas. Como exemplos de linguagens de programação modernas que suportam múltiplos paradigmas, podemos citar a linguagem Scala[1], uma linguagem orientada a objetos e funcional, e a linguagem Xtend[2], que suporta três dos paradigmas de programação clássicos: imperativo, orientado a objetos e funcional. Ambas as linguagens executam sobre a Máquina Virtual Java[3], que será discutida na Seção 2.3.2.

Dessa forma, o relacionamento entre paradigmas de programação e linguagens de programação pode ser complexo pelo fato de linguagens de programação poderem suportar mais de um paradigma. Assim como diferentes grupos em engenharia de software propõem diferentes metodologias, diferentes linguagens de programação propõem diferentes paradigmas de programação. Algumas linguagens foram desenvolvidas para suportar um paradigma específico (Smalltalk suporta o paradigma orientado a objetos, enquanto Haskell suporta o paradigma funcional), enquanto outras linguagens suportam múltiplos paradigmas (como o Scala e Xtend).

[1] http://www.scala-lang.org.
[2] http://www.eclipse.org/xtend.
[3] Em inglês: Java Virtual Machine (JVM).

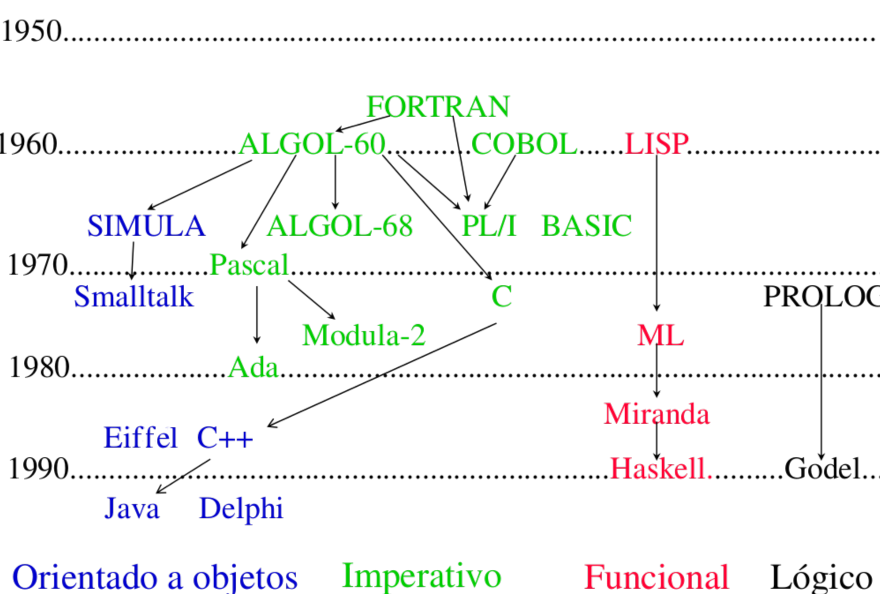


Figura 1.1 Paradigmas.

1.3.1 Paradigma orientado a objetos: breve histórico

A orientação a objetos tem sua origem nos anos 1960 na Noruega, com Kristen Nygaard e Ole-Johan Dahl, no Centro Norueguês de Computação. Através da linguagem Simula 67 (DAHL; MYHRHAUG; NYGAARD, 1968), foram introduzidos os conceitos de classe e herança.

A orientação a objetos foi mais bem conceituada no laboratório da Xerox, em Palo Alto, sendo refinada numa sequência de protótipos da linguagem Smalltalk (KAY, 1996). O líder desse projeto foi Alan Kay, considerado um dos criadores do termo "programação orientada a objetos".

Alan Kay começou a programar em Simula 67 e, a partir dos conceitos desse sistema, como também dos seus conhecimentos em Biologia e Matemática[4], formulou sua analogia "algébrico-biológica" (FELDMAN, 2004). Ele lançou o postulado de que o computador ideal deveria funcionar como um organismo vivo, isto é, cada "célula" comportar-se-ia relacionando-se com outras células a fim de alcançar um objetivo, entretanto, funcionando de forma autônoma. As células poderiam também reagrupar-se para resolver outros problemas ou desempenhar outras funções, trocando mensagens "químicas" entre elas.

[4] Ele é bacharel em matemática e biologia molecular pela Universidade do Colorado.

Baseando-se nessas ideias, Alan Kay pensou em como construir um sistema de software a partir de agentes autônomos que interagissem entre si, estabelecendo os seguintes princípios da orientação a objetos (KAY, 1996):

- Qualquer coisa é um objeto.
- Objetos realizam tarefas através da requisição de serviços.
- Cada objeto pertence a uma determinada classe.
- Uma classe agrupa objetos similares.
- Um classe possui comportamentos associados ao objeto.
- Classes são organizadas em hierarquias.

Portanto, depois de reunir conceitos de diversas áreas do conhecimento e com base em sua experiência como pesquisador, Alan Kay criou o paradigma de orientação a objetos. Após sua definição inicial, os conceitos definidos por Alan Kay foram estendidos e consolidados pela comunidade científica nos últimos anos e tornaram-se o paradigma de análise e programação mais eficiente da atualidade.

1.4 Afinal de contas, o que significa ser orientado a objetos?

Esta seção discute os onze conceitos fundamentais ao paradigma orientado a objetos. A discussão aqui apresentada baseia-se na discussão encontrada em Page-Jones (2001) e Sebesta (2003).

A melhor forma de definir esses conceitos é através de um exemplo simples da classe `Conta`, que representa contas bancárias e cuja representação UML (BOOCH; RUMBAUGH; JACOBSON, 2000) encontra-se na Figura 1.2. Pela figura, pode-se observar as seguintes características da classe `Conta`:

- O atributo `saldo` é responsável por armazenar o saldo da conta bancária;
- O método `getSaldo()` é responsável por retornar o saldo da conta corrente. Ou seja, esse método retorna o valor do atributo `saldo`;
- Os métodos `saque()` e `deposito()` são responsáveis, respectivamente, por realizar as operações de saque e depósito na conta bancária. Ou seja, esses métodos atualizam o valor do atributo `saldo`.

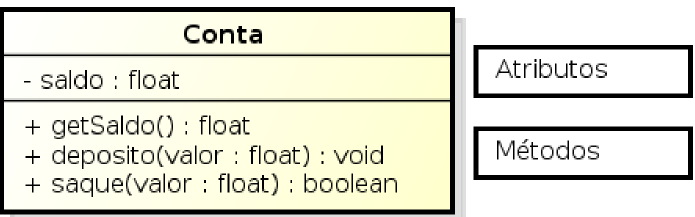


Figura 1.2 Diagrama de classes UML: classe Conta.

1.4.1 Abstração de dados

Uma abstração é uma visualização ou representação de uma entidade que inclui somente as características de importância em um contexto particular (SEBESTA, 2003). No contexto do paradigma orientado a objetos, a *abstração de dados* é obtida através da definição de um Tipo Abstrato de Dados (TAD) que consiste, colocando de maneira simples, em um encapsulamento (Seção 1.4.2) o qual combina dados e operações (sobre esses dados) em um elemento único. O conceito de tipos abstratos de dados encontra-se presente em linguagens não orientadas a objetos, tais como Ada (BARNES, 2006), porém o paradigma orientado a objetos estende esse conceito ao permitir, por exemplo, a herança (Seção 1.4.8) de tipos abstratos de dados.

- *Classe*: definição do tipo abstrato de dados.
- *Objeto*: cada instância criada a partir da classe.

No caso do exemplo das contas bancárias, a classe **`Conta`** pode ser considerada um tipo abstrato de dados, pois combina os dados (atributo **`saldo`**) e as operações (métodos **`getSaldo()`**, **`saque()`** e **`deposito()`**) em um elemento único (ver Figura 1.2).

1.4.2 Encapsulamento

Quando o tamanho do sistema se estende para além de milhares de linhas, um problema prático aparece. Do ponto de vista do programador, manter esse programa de tal forma que ele apareça como uma coleção coesa de subprogramas não é uma tarefa intelectualmente simples.

Dessa forma, *encapsulamento*, que vem de encapsular, no paradigma orientado a objetos significa separar o programa em partes, as mais isoladas possíveis (SCOTT, 2009). Ou seja, encapsular no contexto do paradigma orientado a objetos significa

separar o programa em classes que combinam dados e operações (sobre esses dados) em um elemento único, sendo os dados denominados como *atributos* da classe e as operações denominadas como *métodos* da classe (ver Figura 1.2). A ideia é tornar o software mais flexível, fácil de modificar e de criar novas implementações.

1.4.3 Ocultação de informações e implementações

Page-Jones define os conceitos de *ocultação de informações e implementações* da seguinte maneira: "A ocultação de informações e implementações é a utilização do encapsulamento para restringir a visibilidade externa de certos detalhes das informações ou implementações, os quais são internos à estrutura do encapsulamento" (PAGE-JONES, 2001, p. 13).

Ou seja, o termo *ocultação de informações* implica que as informações contidas em um objeto não podem ser acessadas por entidades externas a ele, enquanto o termo *ocultação de implementações* implica que os detalhes de implementações contidas em um objeto também não podem ser acessados externamente.

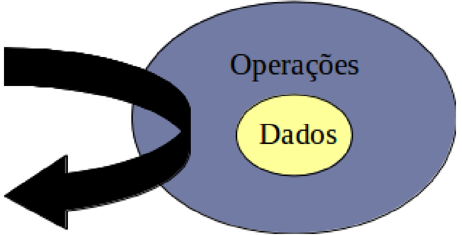


Figura 1.3 Ocultação de informações e implementações.

A ocultação de informações e implementações é uma técnica poderosa para lidar com a complexidade existente em software. Isso significa que um objeto parece com uma caixa preta para um observador externo (ver Figura 1.3). Em outras palavras, o observador externo tem pleno conhecimento do *que* o objeto pode fazer (ou seja, conhece seus métodos), mas não tem conhecimento de *como* ele faz isso (isto é, não tem conhecimento acerca da implementação dos métodos) ou de *como* ele é construído internamente.

1.4.4 Retenção de estado

O quarto conceito inerente ao paradigma orientado a objetos pertence à habilidade de um objeto reter seu *estado*, e o estado consiste do conjunto de valores (de seus atributos) que um objeto consegue manter consigo. Todavia, como discutido na Seção 1.4.3, a maneira *como* o objeto escolhe reter esse conhecimento corresponde a uma atividade interna, exclusiva dele.

Ao reter seu estado, um objeto é ciente de seu passado (operações que foram executadas previamente). Além disso, o estado influencia o comportamento do objeto[5]. Por exemplo, o método **`getSaldo()`** retorna valores diferentes antes e após a execução de um saque (método **`saque()`**) ou de um depósito (método **`deposito()`**).

A abstração de dados (Seção 1.4.1), o encapsulamento (Seção 1.4.2), a ocultação de informações e implementações (Seção 1.4.3), assim como a retenção de estado estão no cerne do paradigma orientado a objetos. Porém, essas não são ideias novas, elas já se encontravam presentes na definição original do conceito de tipo abstrato de dados (SEBESTA, 2003). Entretanto, o paradigma orientado a objetos vai além do conceito de tipo abstrato de dados, como a discussão dos próximos sete conceitos inerentes ao paradigma orientado a objetos revelará.

1.4.5 Identidade de objeto

Page-Jones define os conceitos de *identidade de objeto* da seguinte maneira: "A identidade de objeto é a propriedade pela qual cada objeto (independentemente de sua classe ou seu estado) pode ser identificado e tratado como uma entidade distinta de software" (PAGE-JONES, 2001, p. 16).

A implementação desse conceito em uma linguagem orientada a objetos depende fortemente da presença de um mecanismo de *identificação de objetos*. Um identificador é uma identidade anexada a um objeto quando este é criado. Duas regras aplicam-se a identificadores:

- O mesmo identificador permanece com o objeto por toda sua vida, independentemente do que possa acontecer ao objeto durante esse período;
- Dois objetos nunca podem ter o mesmo identificador. Dois objetos podem ter o mesmo "estado" (Seção 1.4.4), porém seus identificadores são distintos.

[5] O comportamento do objeto é regido pelo comportamento de seus métodos.

A Figura 1.4 apresenta dois objetos da classe **Conta** que possuem o mesmo estado (ou seja, saldos iguais), porém identificadores distintos – 123758 e 167893. Esses identificadores foram atribuídos aos objetos no momento de sua criação e permanecem com eles durante toda a vida desses objetos.

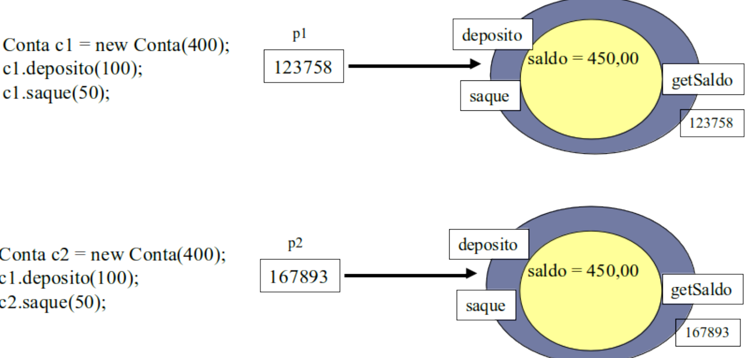


Figura 1.4 Identidade de objetos.

A Figura 1.4 também ilustra que um objeto é ciente de seu passado – operações que foram executadas previamente (ver discussão na Seção 1.4.4). O atributo **saldo** possui o valor R$ 450,00 devido aos seguintes fatos:

- Os objetos foram criados com saldo de R$ 400,00;
- Logo após, houve uma operação de depósito (método **deposito()**) de R$ 100,00. Dessa forma, o atributo **saldo** torna a possuir o valor de R$ 500,00;
- E, por fim, houve uma operação de saque (método **saque()**) de R$ 50,00, que atualizou o valor do atributo **saldo** para R$ 450,00.

1.4.5.1 Referências para objetos

Pela Figura 1.4, pode-se observar que a operação de atribuição (=) faz com que as variáveis **c1** e **c2** retenham o identificador do objeto criado no lado direito do comando de atribuição. Nesse ponto, é importante salientar que, na maioria das linguagens orientadas a objetos, se uma variável retém o identificador de um objeto implica que essa variável "aponta" para esse objeto.

Por outro lado, pela Figura 1.5, pode-se observar que o comando de atribuição (**c2 = c1**) faz com que ambas as variáveis retenham o mesmo objeto identificador. Ter duas variáveis apontando para o mesmo objeto geralmente não é útil[6].

[6] Esse é um erro típico de programadores com pouca experiência no paradigma orientado a objetos.

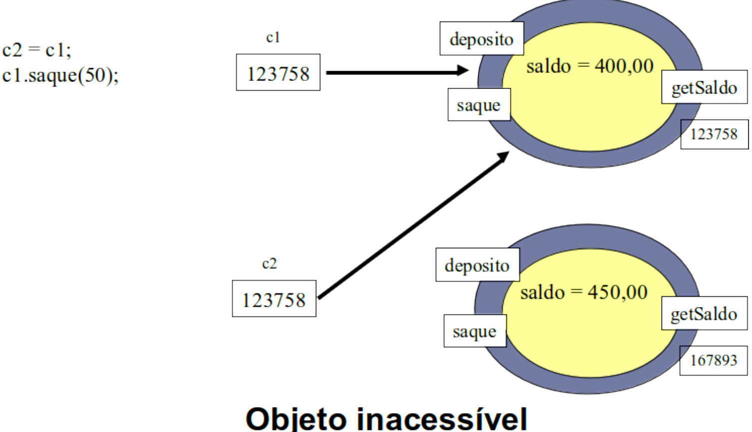


Figura 1.5 Identidade de objetos: objeto inacessível.

Para piorar a situação, não há mais meios de acessar o segundo objeto, pois não existem mais variáveis que retenham o seu identificador. Nessa situação, a maioria das linguagens orientadas a objetos chamaria um coletor de lixo para remover o objeto da memória[7].

1.4.6 Mensagens

Um objeto se comunica com outro, geralmente solicitando que este execute algum método, através da troca de mensagens. Page-Jones define o conceito de *mensagem* da seguinte maneira: "Uma mensagem é o veículo pelo qual um objeto remetente **obj1** transmite a um objeto destinatário **obj2** um pedido para o **obj2** executar um de seus métodos" (PAGE-JONES, 2001, p. 20).

A Figura 1.6 ilustra o envio de uma mensagem para um objeto da classe `Conta` solicitando que este execute o método `getSaldo()`.

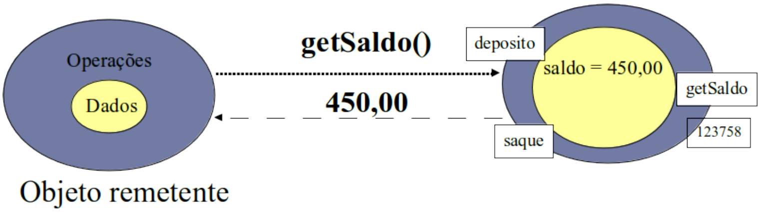


Figura 1.6 Mensagens.

[7] O coletor de lixo da linguagem Java será discutido no Capítulo 2.

1.4.7 Classes/Objetos

No paradigma orientado a objetos, uma *classe* é uma estrutura que abstrai um conjunto de *objetos* com características similares. Uma classe define o comportamento de seus objetos através da definição de *métodos* e os estados (ver Seção 1.4.4) possíveis desses objetos através da definição de *atributos*. Em outras palavras, uma classe descreve as funcionalidades providas por seus objetos e quais informações eles podem armazenar.

- `Pessoa`, `Automóvel` e `Conta` são exemplos de classes;
- **João** e **Maria** são dois objetos (instâncias) da classe `Pessoa`.

À primeira vista, pode parecer confusa a distinção entre uma classe e um objeto. Page-Jones argumenta que o modo mais simples de distinguir esses dois conceitos é lembrar que "A classe é que você projeta e programa, enquanto o objeto é o que você cria (a partir de uma classe) em tempo de execução" (PAGE-JONES, 2001, p. 28).

1.4.8 Herança

A herança define o relacionamento "é um" entre classes no qual uma classe, denominada *subclasse*, herda todos os comportamentos (métodos) e estados (atributos) de outra classe, denominada *superclasse*. Em outras palavras, a herança é um mecanismo para derivar novas classes a partir das classes já existentes. Existem três possibilidades: (a) a subclasse adicionar novos atributos; (b) a subclasse adicionar novos métodos; e (c) a subclasse redefinir a implementação dos métodos herdados.

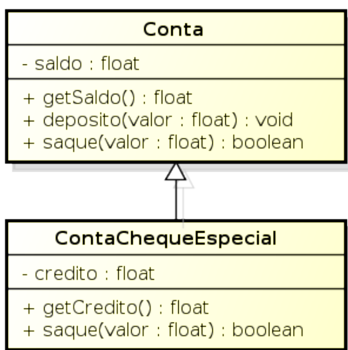


Figura 1.7 Diagrama de classes UML: classe ContaChequeEspecial.

Como exemplo do relacionamento de herança, observe na Figura 1.7 que a classe `ContaChequeEspecial` é subclasse da classe `Conta`. A classe `ContaChequeEspecial` é um tipo especial de conta em que a instituição bancária disponibiliza ao correntista um crédito, caso necessário[8]. Como se pode observar, a classe `ContaChequeEspecial`:

(a) Adiciona o atributo `credito`, que é responsável por armazenar o valor do crédito disponível ao correntista;

(b) Adiciona o método `getCredito()`, que é responsável por retornar o valor do crédito disponível. Ou seja, esse método retorna o valor do atributo `credito`;

(c) Redefine o método `saque()`, para levar em consideração o crédito disponível ao correntista. Ou seja, saques serão permitidos mesmo que o saldo fique negativo, contanto que não ultrapasse o crédito disponibilizado pelo cheque especial.

O mecanismo de herança permite construir uma hierarquia Generalização/Especialização de classes baseada no relacionamento "é um", em que as classes mais genéricas se encontram nos níveis mais altos da hierarquia, enquanto as classes mais especializadas se encontram nos níveis mais baixos da hierarquia.

Por exemplo, a Figura 1.8 ilustra uma hierarquia incompleta de classes que representam seres vivos. Por essa hierarquia, pode-se afirmar que um **animal** "é um" **ser vivo**, que um **mamífero** "é um" **animal** e que um **homem** "é um" **mamífero**. Nesse ponto, é importante ressaltar que o relacionamento "é um" é transitivo. Dessa forma, pode-se afirmar que um **homem** "é um" **ser vivo**.

O mecanismo de herança possibilita a adoção de uma abordagem incremental de desenvolvimento de software com as seguintes características:

- Primeiro construa classes para lidar com o caso mais geral (classes que se encontram nos níveis mais altos da hierarquia);
- Em seguida, com o objetivo de tratar os casos especiais, acrescente classes mais especializadas – herdadas das primeiras classes. Essas novas classes são aptas a utilizar todos os métodos e atributos das classes originais (superclasses).

[8] Cheque especial é o nome dado, no sistema financeiro brasileiro, ao crédito automático que o banco disponibiliza ao correntista para caso ele necessite efetuar pagamentos, transferências ou saques em sua conta e não haja saldo disponível.

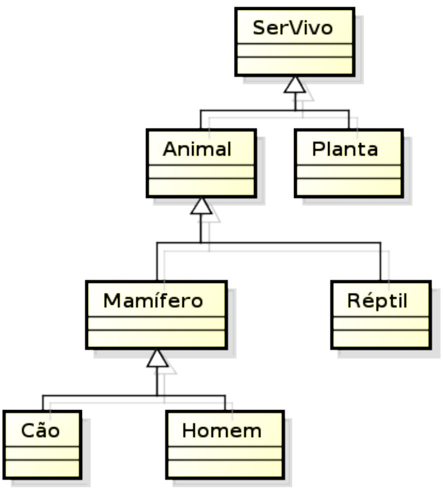


Figura 1.8 Diagrama de classes UML: Hierarquia de classes.

E, por fim, finalizando a discussão sobre o mecanismo de herança, é importante mencionar que esse mecanismo possibilita a substituição da superclasse pelas subclasses em qualquer programa que exija a superclasse. Ou seja, basicamente isso significa que, se você escrever um programa, supondo que utilizará a classe `SerVivo`, então pode usar livremente qualquer subclasse de `SerVivo` (`Animal`, `Homem`, etc.) em substituição desta.

1.4.9 Classes abstratas, classes concretas e interfaces

Uma *classe abstrata* representa entidades e conceitos abstratos. A classe abstrata é sempre uma superclasse que não possui instâncias. Na hierarquia da Figura 1.8, pode-se considerar que as classes `SerVivo`, `Animal`, `Planta`, `Mamífero` e `Réptil` são classes abstratas, pois não existem instâncias destas, apenas de suas subclasses mais especializadas (classes `Cão` e `Homem`).

Uma classe abstrata define um modelo para uma funcionalidade e fornece uma implementação incompleta – a parte genérica dessa funcionalidade –, que é compartilhada pelo grupo de classes derivadas (subclasses). Cada uma das classes derivadas completa a funcionalidade da classe abstrata adicionando um comportamento específico.

Uma classe abstrata geralmente possui métodos abstratos. Esses métodos devem ser implementados nas suas classes derivadas concretas com o objetivo de definir o comportamento específico. O método abstrato apenas define a assinatura (nome, parâmetros e retorno) do método e, portanto, não contém código.

Por outro lado, as *classes concretas* implementam todos os seus métodos e permitem a criação de instâncias. Uma classe concreta não possui métodos abstratos e, geralmente, são classes derivadas de uma classe abstrata.

Já uma *interface* é uma coleção de declarações de métodos sem dados (sem atributos) e sem corpo. Ou seja, os métodos de uma interface são sempre vazios – são simples assinaturas de métodos. Dessa forma, pode-se considerar que uma interface é um contrato entre a classe (que a implementa) e o mundo externo. Quando uma classe concreta implementa uma interface, ela deve implementar todos os métodos declarados nessa interface. Dessa forma, pode-se dizer que a interface impõe que a classe que a implementa tenha métodos com assinaturas específicas.

No primeiro momento, parece que os conceitos de classes abstratas e interfaces são similares devido às semelhanças encontradas (não podem ser instanciadas, presença de métodos vazios, etc.). Porém, é importante destacar as principais diferenças entre esses dois conceitos:

- Interfaces contêm apenas assinaturas de métodos sem sua respectiva implementação. Classes abstratas podem conter a implementação de alguns de seus métodos. Nesse caso, a classe especializada (subclasse) decide se redefine ou não esses métodos.
- Ao criar uma subclasse, as definições de atributos e métodos da superclasse são herdados e fazem parte da subclasse. Todavia, ao definir que uma classe implementa uma interface, essa classe precisa, de fato, implementar todos os métodos do contrato definido pela interface.

Se para o leitor continua confusa a distinção entre os conceitos de classes abstratas e interfaces, a Unidade 3 revisita esses conceitos no contexto da linguagem Java. Provavelmente torna-se-á mais claro o entendimento desses conceitos com a utilização de exemplos práticos implementados na linguagem Java.

1.4.10 Polimorfismo

Polimorfismo, no contexto do paradigma orientado a objetos, é o mecanismo que possibilita que uma única operação (método) seja definida e assuma diferentes implementações em diferentes classes – um método polimórfico é um método implementado por uma superclasse e reimplementada (redefinida) pelas subclasses.

Uma linguagem orientada a objetos geralmente implementa polimorfismo por meio do mecanismo de ligação dinâmica[9], que permite que o método a ser executado seja apenas determinado em tempo de execução. A Figura 1.7 contém o método polimórfico **saque()**. Esse método é polimórfico pois ele foi definido na classe **Conta** e redefinido na subclasse **ContaChequeEspecial**.

Suponha a situação apresentada na Figura 1.9, em que o correntista é indagado, no momento da criação da conta bancária, se deseja ou não o cheque especial. O método **saque()** a ser executado depende da resposta dada pelo correntista. Se o correntista aceita o cheque especial, então o método **saque()** a ser executado encontra-se implementado na classe **ContaChequeEspecial**. Caso o correntista não deseje o cheque especial, então o método a ser executado encontra-se implementado na classe **Conta**.

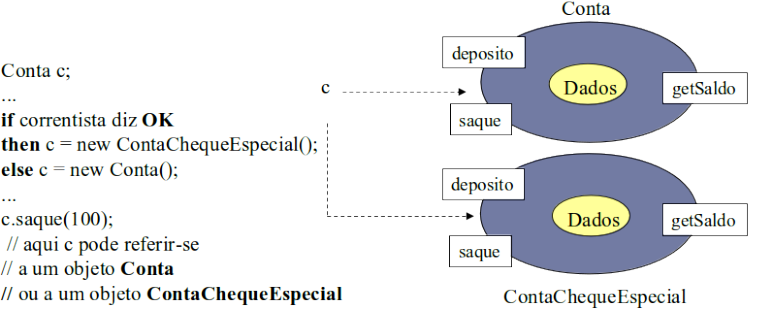


Figura 1.9 Polimorfismo.

1.4.11 Relacionamento entre classes/objetos

Além do relacionamento "é um", definido pelo mecanismo de herança (Seção 1.4.8), é possível outros três tipos de relacionamentos entre classes: associação, composição e agregação.

A *associação* simples é um vínculo que permite que objetos de uma ou mais classes se relacionem. Através desses vínculos é possível que um objeto invoque comportamentos (ou seja, envie mensagens – Seção 1.4.6) e estados de outros objetos. Uma associação simples nos permite capturar relacionamentos básicos que existem entre conjuntos de objetos.

[9] Em inglês: dynamic binding.

A Figura 1.10 ilustra uma associação simples entre as classes `Pessoa` e `Carro` a qual indica que uma pessoa pode possuir um ou mais carros. Porém, o carro pertence a apenas uma pessoa.

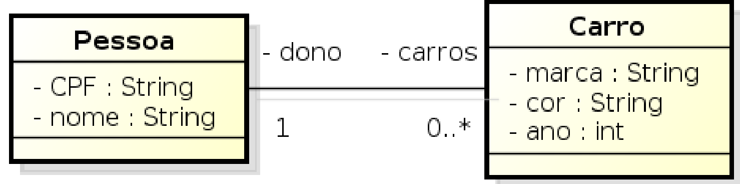


Figura 1.10 Diagrama de classes UML: relacionamento de associação.

Os outros dois tipos de relacionamentos – composição e agregação – têm características "todo-parte", em que o objeto "todo" é composto de objetos "parte".

A *composição* é um relacionamento com características "todo-parte", em que existe um alto grau de coesão entre o **todo** e as **partes**. Ou seja, a existência dos objetos "parte" não faz sentido se o objeto "todo" não existir. Por exemplo, considere a Figura 1.11, que ilustra uma composição em que um pedido é composto de um ou mais itens de pedido, mas um item de pedido não existe se o pedido não existir.

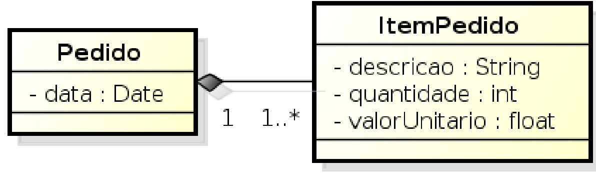


Figura 1.11 Diagrama de classes UML: relacionamento de composição.

A *agregação* é um relacionamento com características "todo-parte", em que existe um grau de coesão entre o **todo** e as **partes** menos intenso, podendo haver um certo grau de independência entre eles. Ou seja, a existência dos objetos "parte" faz sentido, mesmo não existindo o objeto "todo". Por exemplo, considere a Figura 1.12, que ilustra uma agregação em que um time é composto de um ou mais atletas. Ou seja, os atletas são parte integrante de um time, mas os atletas existem independentemente de um time existir.

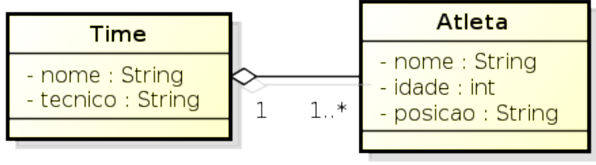


Figura 1.12 Diagrama de classes UML: relacionamento de agregação.

1.5 Considerações finais

Esta unidade apresentou os conceitos inerentes ao paradigma orientado a objetos. As próximas unidades oferecem uma introdução à linguagem Java, ressaltando como os conceitos discutidos nessa unidade são definidos nessa linguagem.

1.6 Estudos complementares

Para estudos complementares sobre os tópicos abordados nesta unidade, o leitor interessado pode consultar as seguintes referências:

NOONAN, R.; TUCKER, A. *Linguagens de Programação* – Princípios e Paradigmas. Porto Alegre: McGraw-Hill Artmed, 2009.

PAGE-JONES, M. *Fundamentos do Desenho Orientado a Objeto com UML*. São Paulo: MAKRON *Books* Ltda., 2001.

SCHACH, S. R. *Engenharia de Software*: Os Paradigmas Clássico e Orientado a Objetos. Porto Alegre: McGraw-Hill Artmed, 2009.

SCOTT, M. L. *Programming Language Pragmatics*. 3. ed. Burlington: Morgan Kaufmann, 2009.

SEBESTA, R. W. *Conceitos de Linguagens de Programação*. 5. ed. Porto Alegre: Bookman Companhia, 2003.

UNIDADE 2

Introdução à linguagem Java

2.1 Primeiras palavras

Na unidade anterior, foram discutidas as principais características do paradigma orientado a objetos. Esta unidade apresenta a linguagem de programação Java, ressaltando como os conceitos básicos inerentes a esse paradigma foram definidos nessa linguagem.

É importante deixar uma ressalva ao leitor. Este livro não deveria ser considerado um material de referência sobre a linguagem Java. Para mais detalhes sobre as funcionalidades presentes nessa linguagem, sugere-se que o leitor consulte as referências listadas na Seção 2.8.

2.2 Problematizando o tema

Ao final desta unidade, espera-se que o leitor seja capaz de reconhecer e definir precisamente os conceitos básicos (classes, objetos, etc.) relacionados ao paradigma orientado a objetos na linguagem de programação Java. Dessa forma, esta unidade pretende discutir as seguintes questões:

- Quais as principais características da linguagem de programação Java?
- Como são definidos os conceitos básicos (classes, objetos, etc.) inerentes ao paradigma orientado a objetos na linguagem de programação Java?

2.3 O que é Java?

Java é uma linguagem de programação orientada a objetos, fácil de aprender e que possui como maior vantagem ser portável entre as diversas plataformas existentes.

2.3.1 Breve histórico

Esta seção apresenta um breve histórico da linguagem Java. O conteúdo aqui apresentado baseia-se no apresentado em Caelum (2014a).

Em 1991, a Sun Microsystem iniciou o *Green Project*, o berço da linguagem Java, que tinha como mentores James Gosling, Patrick Naughton e Mike Sheridan. A princípio, o objetivo desse projeto não era criar uma nova linguagem, e sim proporcionar meios para a convergência de computadores com outros tipos de equipamentos – eles

acreditavam que, em algum momento, haveria uma convergência dos computadores com equipamentos e eletrodomésticos comumente utilizados pelas pessoas no seu cotidiano. Porém, a ideia não deu certo. Eles não tiveram êxito em arrumar parceiros comerciais tais como grandes fabricantes de equipamentos e eletrodomésticos. A ideia que o *Green Project* tentava vender, atualmente, é realidade na televisão digital. Ou seja, era uma ideia certa em uma época errada.

Porém, com o advento da Web, a Sun Microsystem percebeu que poderia utilizar essa ideia para criar pequenas aplicações (*applets*) dentro do navegador. A semelhança era que na internet havia uma grande quantidade de sistemas operacionais e navegadores, e com isso seria grande vantagem ter a possibilidade de programar em uma única linguagem, independentemente da plataforma.

Dessa forma, em 1995 a Sun Microsystem lançou a plataforma Java no mercado. Ela é constituída pela linguagem de programação Java, a Máquina Virtual Java (Seção 2.3.2) e diversas *APIs* (*Application Programming Interface*).

Sob o slogan "*write once, run anywhere*", que referencia sua portabilidade e indica que um programa Java pode ser executado em qualquer plataforma desde que esta possua a Máquina Virtual Java, a linguagem Java se consolidou como uma linguagem de programação multiplataforma e tornou-se popular pelo seu uso na internet. Atualmente, possui seu ambiente de execução presente em navegadores, *mainframes*, celulares, etc. Em 2009, a Oracle comprou a Sun Microsystem, com o objetivo de fortalecer a marca.

Para saber mais detalhes sobre a história da linguagem Java, o leitor interessado pode consultar o seguinte link: http://oracle.com.edgesuite.net/timeline/java/.

2.3.2 Máquina Virtual Java

Uma máquina virtual representa a abstração de uma máquina real, em que um software simula um computador e executa vários programas. Ela é um ambiente operacional independente da máquina ou dispositivo em que está hospedada e sendo executada. Dessa forma, ela constitui uma plataforma, e o sistema operacional, a memória, o processador e seus demais recursos são virtuais.

A máquina virtual permite realizar a interpretação de um código intermediário, o qual é gerado como resultado da tradução de uma linguagem de alto nível. Com isso, garante-se a portabilidade do código do programa. Isso porque, como o código está

numa linguagem intermediária independente da arquitetura de um computador real, só é necessário que a máquina virtual esteja presente no computador onde o programa será executado, uma vez que ela será a responsável pela interpretação do código para a linguagem de máquina do computador em questão.

Dentre as vantagens proporcionadas pela máquina virtual, pode-se citar o fato de que os programas intermediários são compactos e podem ser executados em qualquer plataforma na qual a máquina virtual esteja presente. Além disso, a velocidade de execução é superior àquela de um interpretador puro. Este último garante o quesito portabilidade tal como uma máquina virtual. No entanto, a velocidade de execução ainda é inferior a de um compilador nativo.

A Máquina Virtual Java[1] (LINDHOLM; YELLIN, 1999) caracteriza a portabilidade de Java, permitindo que um programa seja executado em diferentes plataformas, independentemente da arquitetura, sistema operacional ou processador. Isso se deve ao fato de a compilação do código do programa ser feita para *bytecodes*, ou seja, para uma máquina genérica, a Máquina Virtual Java. Esta, por sua vez, traduz o código intermediário gerado pela compilação (*bytecodes*) para a linguagem de máquina característica da plataforma em que o programa e a Máquina Virtual Java executarão.

Além de ser responsável pela interpretação dos *bytecodes*, a Máquina Virtual Java verifica se estes se adequam às suas especificações e se não violam a integridade e segurança do sistema. Ela também é responsável por carregar o programa de forma segura.

2.3.3 Ambiente de desenvolvimento

Um programa escrito na linguagem Java é compilado (compilador **javac**), gerando os *bytecodes* do programa. Estes são armazenados na memória e depois verificados se não violam restrições de segurança do sistema. Após essa primeira etapa, a máquina virtual (Seção 2.3.2) realiza a execução do programa, traduzindo os *bytecodes* para a linguagem de máquina nativa.

Os *bytecodes* representam uma linguagem intermediária entre Java e a linguagem de máquina. Eles são instruções em código de máquina para a máquina virtual Java, podendo ser compilados e executados em diferentes plataformas, desde que estas suportem Java.

[1] Em inglês: *Java Virtual Machine* (JVM).

O processo de execução de um programa em Java foi aperfeiçoado com o decorrer do tempo. Isso porque, inicialmente, a máquina virtual era simplesmente um interpretador de *bytecodes*, interpretando e executando um *bytecode* por vez. No entanto, atualmente, há uma combinação de interpretação com compilação *Just-in-Time* (JIT) para executar os *bytecodes*. Dessa forma, os *bytecodes* são analisados à medida que são interpretados. Essa análise permite identificar as partes que são executadas com maior frequência, sendo estas nomeadas *hot spots*. A partir disso, os *bytecodes* dessas partes são traduzidos em linguagem de máquina pelo compilador JIT.

Assim, as duas etapas de compilação de um programa permitem identificar o caráter multiplataforma de Java. Isso porque, na segunda etapa, os *bytecodes* são traduzidos para linguagem de máquina, dependendo, assim, da arquitetura do computador ou dispositivo em que o programa será executado. Entretanto, na primeira etapa, o código Java é traduzido em *bytecodes*, que independem da plataforma em questão, caracterizando, assim, a portabilidade da máquina virtual em relação a diferentes arquiteturas.

2.3.3.1 Configuração do ambiente de desenvolvimento

Esta seção apresenta algumas dicas importantes no processo de instalação e configuração do ambiente de desenvolvimento necessário para compilar e executar os exemplos discutidos neste material.

O Java Development Kit (JDK) será necessário para compilar e executar os exemplos apresentados nesse material. A última versão do JDK pode ser obtida em http://java.com/pt_BR/download/index.jsp. No momento da escrita deste material, a última versão do JDK é a 1.7.0_65. Para obter dicas em como instalar o JDK, o leitor interessado pode consultar a seguinte referência: Caelum (2014a).

É importante salientar a diferença entre a JDK e a JRE:

- JRE **(Java Runtime Environment)**: é o ambiente de execução Java, formado pela JVM e bibliotecas. Ou seja, tudo que você precisa para executar uma aplicação Java. No entanto, no contexto deste material, apenas executar programas Java não é suficiente. Por exemplo, será necessário também compilar programas Java. Dessa forma, é necessário o JDK.

- JDK **(Java Development Kit)**: formado pela JRE adicionada a ferramentas, tais como o compilador.

Na realidade, Java não exige a utilização de um IDE. Todos os comandos necessários ao desenvolvimento poderiam ser feitos em um terminal de comando. No entanto, assim como em outras linguagens de programação, o IDE torna ágil o processo de desenvolvimento ao integrar diferentes funcionalidades (edição, compilação, execução, etc.) e abstrair a sintaxe dos comandos necessários relacionados a essas atividades. Dessa forma, esse material sugere e indica alguns links que podem auxiliar na tarefa de configurar o IDE para que possa ser utilizada no desenvolvimento de sistemas Java.

- Netbeans: https://netbeans.org/kb/docs/java/quickstart_pt_BR.html.
- Eclipse: http://www.caelum.com.br/apostila-java-orientacao-objetos/eclipse-ide/.

2.3.4 Coletor de lixo

Coletor de lixo[2] (JONES; LINS, 1999) é um processo usado para a automação do gerenciamento de memória. Com ele, é possível recuperar uma área de memória inutilizada por um programa, o que pode evitar problemas de vazamento de memória, resultando no esgotamento da memória livre para alocação.

Esse sistema contrasta com o gerenciamento manual de memória, em que o programador deve especificar explicitamente quando e quais objetos devem ser desalocados e retornados ao sistema.

Uma das vantagens da linguagem Java, em relação a outras linguagens orientadas a objetos, é a presença do coletor de lixo. Ou seja, o coletor de lixo gerencia "quantas" referências estão "apontando" para aquele trecho de memória. Quando esse contador atinge valor "zero", o objeto é candidato a ser desalocado da memória. Em um momento oportuno, o coletor de lixo desalocará esse espaço de memória, dando lugar para que novos objetos o utilizem.

[2] Em inglês: *Garbage Collector*, ou o acrônimo GC.

2.3.5 Primeiro programa Java

Nosso primeiro programa Java imprimirá uma linha simples. Para imprimir uma linha, pode-se utilizar:

```
System.out.println("O primeiro programa da disciplina POO2.");
```

Porém, esse código não seria aceito pelo compilador Java. O Java é uma linguagem burocrática e necessita de mais do que isso para iniciar sua execução. O mínimo que precisaríamos escrever é algo semelhante ao apresentado no Código 2.1.

```java
public class PrimeiroPrograma {
    public static void main(String[] args) {
      System.out.println("O primeiro programa da disciplina POO2.");
    }
}
```

Código 2.1 Primeiro programa Java.

2.3.5.1 Compilando o programa

Após digitar o código acima, deve-se salvar como **PrimeiroPrograma.java** em algum diretório. Para compilar, deve-se pedir ao compilador de Java, chamado **javac**, que gere os *bytecodes* correspondentes ao seu programa. Quando o sistema operacional listar os arquivos contidos no diretório atual, pode-se observar que um arquivo **.class** foi gerado, com o mesmo nome da sua classe Java (Figura 2.1).

```
delano@jequie: ~/UFSCar/livro/POO/codigo/Capitulo2
delano@jequie:~/UFSCar/livro/POO/codigo/Capitulo2$ javac PrimeiroPrograma.java
delano@jequie:~/UFSCar/livro/POO/codigo/Capitulo2$ ls -l
total 8
-rw-rw-r-- 1 delano delano 465 Ago 23 10:20 PrimeiroPrograma.class
-rw-rw-r-- 1 delano delano 227 Ago 23 10:12 PrimeiroPrograma.java
delano@jequie:~/UFSCar/livro/POO/codigo/Capitulo2$
```

Figura 2.1 Compilando o primeiro programa Java.

2.3.5.2 Executando o programa

Para executar o seu primeiro programa, basta invocar a Máquina Virtual Java (JVM) para interpretar seu programa. Vale a pena ressaltar que o comando **javac** invoca o compilador Java, enquanto o comando **java** invoca a JVM (Figura 2.2).

```
delano@jequie: ~/UFSCar/livro/POO/codigo/Capitulo2
delano@jequie:~/UFSCar/livro/POO/codigo/Capitulo2$ java PrimeiroPrograma
O primeiro programa da disciplina POO2.
delano@jequie:~/UFSCar/livro/POO/codigo/Capitulo2$
```

Figura 2.2 Executando o primeiro programa Java.

A linha 3 do Código 2.1 é a que será executada quando invocada a Máquina Virtual Java. A expressão `System.out.println` faz com que o conteúdo entre aspas seja impresso na saída padrão (nesse caso, o monitor do computador). Por enquanto, todas as demais linhas, em que há a declaração de uma classe e a de um método, são irrelevantes. Porém, é importante mencionar nesse momento que toda a aplicação Java tem um ponto de entrada, e esse ponto de entrada é o método `main` (linhas 2-4). A declaração de classes e métodos na linguagem Java serão discutidos na Seção 2.6.

2.4 Tipos primitivos & classes *Wrappers*

Java não é uma linguagem orientada a objetos pura. Ou seja, existem tipos de dados em Java que não são objetos, os chamados **tipos primitivos** (Tabela 2.1). Como se pode observar, os caracteres em Java são baseados na especificação de caracteres unicode[3]. Por exemplo, o caractere 'a' é representado por '\u0061' (97).

Tipo	Tamanho (bytes)	Faixa de valores	classe *Wrapper*
`byte`	1	-128 a 127	`Byte`
`short`	2	-32768 a 32767	`Short`
`int`	4	-2147483648 a 2147483647	`Integer`
`long`	8	-9223372036854775808 a 9223372036854775807	`Long`
`float`	4	single-precision 32-bit IEEE 754 floating point	`Float`
`double`	8	double-precision 64-bit IEEE 754 floating point	`Double`
`char`	2	Caracteres unicode '\u0000' (0) a '\uffff' (65.535)	`Character`
`boolean`	Não definido	Assume `true` ou `false`	`Boolean`

Tabela 2.1 Tipos primitivos de Java e classes *Wrappers*.

No entanto, situações em que se devem utilizar objetos ao invés de tipos primitivos são bastante comuns. Por exemplo, as classes Java que implementam estruturas de dados (Pilha, Fila, etc.) manipulam objetos. Para contornar essas situações, foram criadas as classes *Wrappers*, que permitem encapsular um tipo primitivo em um objeto que o represente. Essas classes possuem métodos que permitem a conversão (*wrapping*) entre os tipos primitivos e objetos bem como a operação inversa (*unwrapping*).

[3] http://www.unicode.org/.

A tarefa de *wrapping/unwrapping* é entediante. A partir da versão 1.5[4], a linguagem Java traz um recurso denominado **autoboxing**, que realiza essa tarefa automaticamente, custando legibilidade (Figura 2.3). No Java 1.4 (e versões anteriores), esse código seria inválido. É importante salientar que isso não quer dizer que tipos primitivos e objetos sejam do mesmo tipo. É apenas uma funcionalidade da linguagem que facilita a codificação.

```
// autoboxing
Integer x = 10; // atribui um valor primitivo a um objeto
// auto-unboxing
int y = x; // obtém valor de uma classe wrapper
```

Figura 2.3 *Autoboxing* e *auto-unboxing* em Java.

2.5 Controle de fluxo

A maioria dos programas toma decisões que afetam seu fluxo. As declarações que tomam essas decisões são chamadas de declarações de controle.

O controle de fluxo na linguagem Java pode ser utilizado tanto por sentenças condicionais quanto por controles de estruturas de repetição. O controle de fluxo em Java é similar ao encontrado nas linguagens C e C++.

2.5.1 Comando de seleção: `if-else`

O comando de seleção `if-else` permite especificar dois comandos (ou dois blocos de comandos delimitados por chaves "{" e "}") que serão alternativas de execução. Um bloco será executado caso a condição seja verdadeira (`true`) e o outro, caso a condição seja falsa (`false`). Quando o bloco de comandos é composto de um único comando, as chaves que delimitam o bloco podem ser omitidas. A cláusula `else` é opcional. Nesse caso, o comando especifica apenas um bloco de comandos que deve ser executado apenas quando a condição for verdadeira.

O Código 2.2 ilustra um programa Java que utiliza o comando `if-else` e que, ao receber três valores inteiros, imprime o maior deles. É importante salientar que esse programa utiliza uma funcionalidade presente na linguagem Java que consiste

[4] Versão lançada em setembro de 2004.

```java
public class ExemploIf {
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        int c = Integer.parseInt(args[2]);
        System.out.println("Valores: " + a + "," + b + "," + c);
        int maior;
        if (a > b & a > c) {
            maior = a;
        } else if (b > c) {
            maior = b;
        } else {
            maior = c;
        }
        System.out.println("Maior é " + maior);
    }
}
```

Código 2.2 Exemplo do comando de seleção `if-else`.

na possibilidade de passar argumentos, pela linha de comando, para o programa a ser executado. O argumento **`args`** é um *array* de *strings* (Seção 3.4.1.1) em que são atribuídos valores, caso os argumentos sejam passados na linha de comando. A Figura 2.4 apresenta um exemplo da execução da classe **`ExemploIf`**.

```
> java ExemploIf 10 4 15
Dentro do método main, a variável args terá os seguintes valores:
args[0] = 10
args[1] = 4
args[2] = 15
e a saída será:
Valores: 10,4,15
Maior é 15
```

Figura 2.4 Exemplo da execução da classe `ExemploIf`.

Sendo **`args`** um *array* de *strings*, deve-se converter os valores para inteiros para serem utilizados no programa. O método **`parseInt()`** da classe **`Integer`**, utilizado nesse programa, realiza a conversão de *strings* para inteiros.

2.5.2 Comando de seleção: **switch**

O comando de seleção **switch** é utilizado quando existem execuções diferenciadas para determinados valores de uma única variável. Ou seja, o comando **switch** também expressa alternativas de execução, porém, nesse caso, as condições estão restritas à comparação de uma variável do tipo **int** (ou **char**) com valores constantes.

```
import java.util.Scanner;

public class ExemploSwitch {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Digite um número: ");
        int n = scanner.nextInt();
        switch(n) {
            case 1: {
                System.out.println("Domingo"); break;
            }
            case 2:  {
                System.out.println("Segunda"); break;
            }
            case 3: {
                System.out.println("Terça"); break;
            }
            case 4: {
                System.out.println("Quarta"); break;
            }
            case 5: {
                System.out.println("Quinta"); break;
            }
            case 6: {
                System.out.println("Sexta"); break;
            }
            case 7: {
                System.out.println("Sábado"); break;
            }
            default: System.out.println("Opção Inválida");
        }
    }
}
```

Código 2.3 Exemplo do comando de seleção `switch`.

O Código 2.3 ilustra um programa Java que utiliza o comando **switch** e que, dado um inteiro n (digitado pelo usuário no teclado), imprime o dia da semana correspondente. A Figura 2.5 apresenta um exemplo da execução da classe **ExemploSwitch**.

> **java** ExemploSwitch
Dentro do método main, o programa solicitará que digite um número
Digite um número: 5
e a saída será:
Quinta

Figura 2.5 Exemplo da execução da classe `ExemploSwitch`.

É importante salientar que esse programa utiliza uma funcionalidade, provida pela classe **Scanner**, a qual será apresentada na Seção 4.5.3. Ela consiste na possibilidade de entrada de dados pelo teclado.

2.5.3 Comando de repetição: **while**

O **while** é um comando usado para executar um laço (*loop*). Isto é, repetir um trecho de código algumas vezes. A ideia é que esse trecho de código seja repetido enquanto uma determinada condição seja verdadeira (**true**).

```
import java.util.Scanner;

public class ExemploWhile {
    public static void main(String[] args) {
        // Encontrar a 1a potência de 2 que seja maior ou igual que um número digitado (positivo)
        Scanner scanner = new Scanner(System.in);
        System.out.print("Digite um número: ");
        int n = scanner.nextInt();
        int x = 1;
        while (x < n) {
            x = x * 2;
        }
        System.out.println("A 1a potência de 2 maior ou igual que " + n + " é " + x);
    }
}
```

Código 2.4 Exemplo do comando de repetição **while**.

O Código 2.4 ilustra um programa Java que utiliza o comando **while** e que, dado um inteiro n (digitado pelo usuário no teclado), imprime a primeira potência de 2 que seja maior ou igual ao inteiro n digitado. A Figura 2.6 apresenta um exemplo da execução da classe **ExemploWhile**.

> **java** ExemploWhile
Dentro do método main, o programa solicitará que digite um número
Digite um número: 5
e a saída será:
A 1a potência de 2 maior ou igual que 5 é 8

Figura 2.6 Exemplo da execução da classe **ExemploWhile**.

2.5.4 Comando de repetição: `do..while`

O comando `do..while` é uma pequena variação do comando `while`. No comando `while`, a condição é testada sempre antes da execução do corpo de comandos que compõe o laço. Já o comando `do..while` tem a condição testada apenas no final. Consequentemente, no caso do `do..while`, existe a garantia de que o conteúdo no interior do laço será executado pelo menos uma vez, enquanto no `while`, ele pode nunca ser executado. Na prática, a existência desses dois tipos de laços é uma mera conveniência sintática, já que um pode ser facilmente substituído pelo outro.

```
import java.util.Scanner;

public class ExemploDoWhile {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        byte n;
        do {
            System.out.print("Digite um número entre 1 e 10: ");
            n = scanner.nextByte();
        } while (n < 1 || n > 10);
    }
}
```

Código 2.5 Exemplo do comando de repetição `do..while`.

O Código 2.5 ilustra um programa Java que utiliza o comando `do..while` e que solicita que o usuário digite um número inteiro n. O laço é repetido enquanto a condição $1 \leq n \leq 10$ não for satisfeita. A Figura 2.7 apresenta um exemplo da execução da classe `ExemploDoWhile`.

> **java** ExemploDoWhile
Dentro do método main, o programa solicitará que digite um número n
Digite um número: 45
O número n **não satisfaz a condição** $1 \leq n \leq 10$
Digite um número: 5

Figura 2.7 Exemplo da execução da classe `ExemploDoWhile`.

2.5.5 Comando de repetição: **for**

Outro comando de repetição extremamente utilizado é o comando **for**. A ideia é a mesma do comando **while** – fazer um trecho de código ser repetido enquanto uma condição continuar verdadeira. Mas, além disso, o comando **for** isola também um espaço para inicialização de variáveis e o incremento dessas variáveis. Isso faz com que as variáveis, as quais são relacionadas ao laço, fiquem mais legíveis.

```java
import java.util.Scanner;

public class ExemploFor {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Digite um número: ");
        int n = scanner.nextInt();
        int soma = 0;
        for (int i = 1; i <= n; i++) {
            soma += i;
        }

        System.out.println("A soma dos " + n + " primeiros números inteiros positivos é " + soma);
    }
}
```

Código 2.6 Exemplo do comando de repetição **for**.

O Código 2.6 ilustra um programa Java que utiliza o comando **for** e que, dado um inteiro n (digitado pelo usuário no teclado), imprime a soma dos n primeiros números inteiros positivos. A Figura 2.8 apresenta um exemplo da execução da classe **ExemploFor**.

> **java** ExemploFor
Dentro do método main, o programa solicitará que digite um número
Digite um número: 5
e a saída será:
A soma dos 5 primeiros números inteiros positivos é 15

Figura 2.8 Exemplo da execução da classe **ExemploFor**.

2.5.6 Comando **break**

Apesar de termos condições booleanas nos laços, em algum momento, pode-se decidir interromper (através do comando **break**) o laço, por algum motivo especial, sem que o resto do laço seja executado.

O Código 2.7 ilustra um programa Java que utiliza o comando **break** para interromper o laço do comando **for**. Esse código vai somar os números de 1 a n e parar quando a variável $i > n$. Ou seja, apresenta o mesmo comportamento do Código 2.6, que realiza a soma dos n números inteiros positivos.

```java
import java.util.Scanner;

public class ExemploBreak {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Digite um número: ");
        int n = scanner.nextInt();
        int soma = 0;
        for (int i = 1; ; i++) {
            if (i > n) break;
            soma += i;
        }

        System.out.println("A soma dos " + n + " primeiros números inteiros positivos é " + soma);
    }
}
```

Código 2.7 Exemplo do comando **break**.

2.5.7 Comando **continue**

Da mesma maneira, é possível obrigar o laço a interromper a iteração corrente e executar a próxima iteração no laço. Para isso, pode-se utilizar a palavra-chave **continue**.

```java
public class ExemploContinue {
    public static void main(String[] args) {
        for (int i = 1; i <= 100; i++) {
            if (i % 2 != 0) continue; // número ímpar
            System.out.println(i);
        }
    }
}
```

Código 2.8 Exemplo do comando **continue**.

O Código 2.8 ilustra um programa Java que utiliza o comando **continue**. Esse código imprime apenas os números pares.

2.6 Orientação a objetos em Java: conceitos básicos

Conforme discutido anteriormente, todo programa Java precisa ser compilado no intuito de que o compilador Java gere os *bytecodes* correspondentes ao programa compilado. Dessa forma, para que o processo de compilação seja realizado com sucesso, espera-se que o código do programa siga certas regras básicas de sintaxe, apresentadas a seguir:

- Uma classe Java é sempre declarada com a palavra reservada `class` seguida do nome da classe. O nome da classe não pode conter espaços e deve ser iniciado por uma letra[5]. Nomes de classes não podem ser iguais às palavras reservadas de Java (apresentadas na Tabela 2.2). Caracteres maiúsculos e minúsculos são diferenciados em Java. Ou seja, as palavras **Class** e **class** são consideradas diferentes, e somente a última pode ser usada para declarar uma classe.
  - Tradicionalmente, os nomes de classes começam com caracteres maiúsculos e alternam entre palavras. Nomes de classes que seguem essa convenção são `ContaCorrente`, `TimeDeFutebol`, `SerVivo` e `ItemPedido`.
  - É aconselhável que os arquivos criados em editores de texto contenham somente uma classe, e que os nomes dos arquivos sejam compostos dos nomes das classes com a extensão .java. Dessa forma, a classe `SerVivo` deve ser salva no arquivo **SerVivo.java**.
- O conteúdo das classes é delimitado pelas chaves (caracteres { e }) – todos os atributos e métodos da classe devem estar entre esses caracteres. Blocos de código correspondentes a métodos também devem estar entre esses caracteres. A cada caractere { que abre um bloco, deve haver um caractere } correspondente que fecha o bloco, caso contrário erros de compilação acontecerão. Um exemplo de classe sintaticamente válida em Java é mostrado no Código 2.9. Conforme pode ser observado, a classe `Conta` não possui atributos nem métodos. Os atributos e métodos dessa classe serão adicionados adiante nesta unidade.

Existem três tipos de comentários em Java, que serão ignorados pelo compilador, mas podem ser bastante úteis para programadores com acesso ao código-fonte da classe.

[5] Para nomes de classes, atributos e métodos, os caracteres _ e $ são considerados letras.

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	false	final	finally
float	for	goto	if	implements	import
instanceof	int	interface	long	native	new
null	package	private	protected	public	return
short	static	strictfp	super	switch	synchronized
this	throw	throws	transient	true	try
void	volatile	while			

Tabela 2.2 Palavras reservadas em Java.

- Comentários de uma única linha começam com duas barras inclinadas (//) e terminam ao final da linha – servem para especificar comentários sobre uma região próxima do código, geralmente a mesma linha, como nas linhas 7 e 13 do Código 2.9.
- Comentários em bloco são delimitados pelos conjuntos de caracteres /* no início e */ no fim – tudo entre esses dois conjuntos de caracteres será considerado como comentários pelo compilador. Um exemplo pode ser visto entre as linhas 8 e 11 do Código 2.9.
- Comentários em bloco para documentação são similares aos comentários em bloco comuns, exceto que são iniciados pela sequência /** ao invés de /*. Esses comentários podem ser analisados pela ferramenta Javadoc para criação automática de documentação para a classe. Um exemplo pode ser visto entre as linhas 1 e 6 do Código 2.9.
  - Javadoc[6]é um gerador de documentação para documentar a *API* dos programas em Java, a partir do código-fonte. O resultado é expresso em HTML. É constituído, basicamente, por algumas marcações muito simples inseridas nos comentários do programa.

2.6.1 Atributos

Atributos e métodos em Java devem ser declarados dentro do corpo da classe (a parte entre as chaves { e }).

A declaração dos atributos em Java é simples: basta declarar o tipo de dado, seguido dos nomes dos atributos que serão daquele tipo. Em Java, o tipo de dado

[6] http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html.

```java
/**
 * A classe Conta, que não possui atributos nem métodos, mesmo assim pode ser usada para
 * exemplificar as regras sintáticas básicas de Java, podendo até mesmo ser compilada.
 *
 * @author Delano Medeiros Beder
 */
class Conta { // esta é a declaração da classe !
/*
 * Se houvessem atributos ou métodos para a classe Conta, eles deveriam ser declarados aqui.
 * Os atributos e métodos dessa classe serão adicionados adiante nesta seção.
 */

} // fim da classe Conta
```

Código 2.9 Classe `Conta` – apenas definição da classe.

pode ser um tipo primitivo (discutido na Seção 2.4) ou uma outra classe. Ou seja, é possível declarar atributos como sendo referências a instâncias de outras classes existentes (Seção 3.5).

Algumas regras e informações sobre a declaração de atributos em Java são mostradas a seguir:

- Nomes de atributos seguem quase todas as mesmas regras de nomes de classes: devem ser iniciados por uma letra (incluindo _ e $), devem ser compostos de uma única palavra (sem espaços, vírgulas, etc.) e podem ser compostos de letras e números.
- Dois atributos de uma mesma classe não podem ter o mesmo nome. Nomes de atributos não podem ser iguais a nenhuma das palavras reservadas mostradas na Tabela 2.2.
- Modificadores de acesso (por exemplo, `private`) podem ser utilizados na declaração de atributos. Modificadores de acesso serão vistos na Seção 2.6.3.

2.6.2 Métodos

Analogamente, a declaração dos métodos em Java é simples: basta declarar o tipo de retorno, seguido do nome e a lista de parâmetros (se houver) do método.

Algumas regras e informações sobre a declaração de métodos em Java são mostradas a seguir:

- Nomes de métodos seguem quase todas as mesmas regras de nomes de atributos: devem ser iniciados por uma letra (incluindo _ e $), devem ser compostos

de uma única palavra (sem espaços, vírgulas, etc.) e podem ser compostos de letras e números. Nomes de métodos não podem ser iguais a nenhuma das palavras reservadas mostradas na Tabela 2.2.

- Métodos não podem ser criados dentro de outros métodos, nem fora da classe a qual pertencem.
- Cada método deve ter, na sua declaração, um tipo ou classe de retorno, correspondente ao valor que o método deve retornar. Caso o método não retorne nada, isto é, caso ele somente execute alguma operação sem precisar retornar valores, o valor de retorno deverá ser `void`, um tipo de retorno especial que indica que o retorno deve ser desconsiderado.
- Métodos que retornam algum valor diferente de `void` devem ter, em seu corpo, a palavra reservada `return` seguida de uma constante ou variável. É importante salientar que essa constante ou variável deve ser do tipo (ou classe) que foi declarado como sendo a de retorno do método. Métodos que retornam `void` não precisam ter a palavra reservada `return` no seu corpo, e, se tiverem, esta não deve ser seguida de nenhuma constante ou variável.
- Métodos podem ter uma lista de argumentos, ou seja, variáveis contendo valores que podem ser usados pelos métodos para efetuar suas operações.
- Modificadores de acesso (por exemplo, `private`) podem ser utilizados na declaração de métodos. Modificadores de acesso serão vistos na Seção 2.6.3.

O Código 2.10 apresenta a implementação da classe `Conta` (Figura 1.2), em que atributos e métodos são declarados.

- O atributo `saldo` é responsável por armazenar o saldo da conta bancária;
- O método `getSaldo()` é responsável por retornar o saldo da conta corrente, ou seja, esse método retorna o valor do atributo `saldo`;
- Os métodos `saque()` e `deposito()` são responsáveis, respectivamente, por realizar as operações de saque e depósito na conta bancária. Ou seja, esses métodos atualizam o valor do atributo `saldo`.
- Pode-se observar que dois construtores foram declarados. Construtores de classes serão discutidos na Seção 2.6.5.

```java
/**
 * Classe Conta Corrente
 *
 * @author Delano Medeiros Beder
 */
public class Conta {
    /*
     * Declaração dos atributos da classe
     */
    protected float saldo; // Atributo da classe que representa o saldo da conta

    /*
     * Declaração dos construtores da classe
     */
    public Conta() {
        this.saldo = 0;
    }

    public Conta(float saldo) {
        this.saldo = saldo;
    }

    /*
     * Declaração dos métodos da classe
     */
    public float getSaldo() {
        return saldo;
    }

    public void deposito (float valor) {
        saldo += valor; // saldo = saldo + valor
    }

    public boolean saque(float valor) {
        boolean ok = false;
        if (saldo >= valor) {
            saldo -= valor; // saldo = saldo - valor
            ok = true;
        }
        return ok;
    }
}
```

Código 2.10 Classe `Conta` com atributos e métodos.

2.6.3 Modificadores de acesso

Uma das principais características do paradigma orientado a objetos é a possibilidade de encapsular atributos e métodos capazes de manipular esses atributos, conforme discutido na Seção 1.4.2. É desejável que os atributos das classes sejam ocultos, para evitar que os dados sejam manipulados diretamente ao invés de por meio dos métodos das classes. A linguagem Java permite a restrição ao acesso a atributos e métodos em classes através de modificadores de acesso que são declarados antes dos métodos e campos. Existem quatro modificadores de acesso, descritos a seguir:

- Atributos e métodos declarados com o modificador `private` só podem ser acessados, modificados ou executados por métodos da mesma classe. Atributos

ou métodos que devam ser ocultos totalmente de usuários da classe devem ser declarados com o modificador `private`.

- Atributos e métodos podem ser declarados sem modificadores. Nesse caso, eles serão considerados como sendo da categoria *package* ou *friendly*, significando que seus atributos e métodos serão visíveis (podendo ser acessados) para todas as classes pertencentes a um mesmo pacote. Pacotes de classes serão discutidos na Seção 3.4.

- O modificador `protected` funciona como o modificador da categoria *package*, exceto que classes derivadas, através do mecanismo de herança de classes, também terão acesso ao atributo ou método declarado com esse modificador. Ou seja, atributos e métodos declarados com o modificador `protected` podem ser acessados por todas as classes do mesmo pacote e por todas as classes que o estendam, mesmo que estas não estejam no mesmo pacote. O mecanismo de herança de classes em Java será discutido na Seção 3.6.

- O modificador `public` garante que o atributo ou método da classe declarado com esse modificador poderá ser acessado ou executado a partir de qualquer outra classe. Atributos e métodos que devam ser acessados (e modificados, no caso de atributos) devem ser declarados com o modificador `public`. A classe `Conta`, mostrada no Código 2.10, encapsula os valores (atributos privados) para representar uma conta bancária, fazendo com que os dados encapsulados só possam ser acessados via seus métodos públicos.

2.6.4 *Getters* e *Setters*

Conforme discutido, é desejável, na medida do possível, que os atributos das classes sejam ocultos para evitar que os dados sejam manipulados diretamente. Porém, com essa política, surge um problema: como fazer para imprimir o valor de um atributo, se não é possível acessá-lo para leitura?

Para permitir o acesso aos atributos de uma maneira controlada, a prática mais comum é, para cada atributo, criar dois métodos – um que retorna o valor e outro que altera o valor do atributo.

A convenção para esses métodos é colocar a palavra `get` ou `set` antes do nome do atributo. Por exemplo, a classe `Conta` com o atributo `saldo` teria dois métodos: `getSaldo()` e `setSaldo()`.

Porém é uma má prática criar uma classe e, logo em seguida, criar *getters* e *setters* para todos os atributos. Esses métodos deveriam apenas ser criados se houver real necessidade. Por exemplo, na classe `Conta`, o método `setSaldo()` não foi criado, pois é recomendado que os usuários dessa classe utilizem os métodos `saque()` e `deposito()` para atualizar o valor do atributo `saldo`.

2.6.5 Construtores de classes

Quando uma classe é criada, pode ser necessário atribuir um valor inicial a alguns dos atributos. Isso pode ser feito através de um **construtor**, o qual é automaticamente executado toda vez que a classe é instanciada.

Em Java, os construtores têm o mesmo nome da classe da qual são membros. O construtor não tem tipo de retorno, isso porque o tipo implícito de retorno é a instância da própria classe.

É importante lembrar que não é obrigatório criar um construtor. Caso não exista a declaração de um construtor, o compilador gera automaticamente um construtor *default* – sem nenhum parâmetro e com implementação vazia. Nesse caso, os atributos serão inicializados com os valores *default* (Tabela 2.3). Porém, a partir do momento que o programador declara um construtor, o construtor *default* não é mais fornecido.

Uma mesma classe pode ter vários construtores, desde que eles tenham quantidade diferente de parâmetros ou parâmetros de tipos diferentes. Na classe `Conta` (Código 2.10), foram definidos dois construtores:

- O primeiro construtor inicializa o atributo `saldo` com o valor zero;
- O segundo construtor inicializa o atributo `saldo` com o valor passado como parâmetro.

Tipo	Valor *default*
`byte`, `short`, `int` e `long`	0
`float`	0.0f
`double`	0.0d
`char`	'\u0000' (0)
`boolean`	`false`
Qualquer classe Java	`null`

Tabela 2.3 Valores *default* dos atributos.

Nos dois construtores, pode-se observar a utilização do operador **this**. A linguagem Java inclui um valor de referência especial, chamado **this**, que é usado dentro de qualquer método/construtor para referir-se ao objeto corrente. O valor de **this** refere-se ao objeto do qual o método/construtor corrente foi invocado. Dessa forma, é permitido que as variáveis locais (ou parâmetros) de um método/construtor tenham o mesmo nome de atributos da classe. Para definir que se está acessando um atributo, utiliza-se o operador **this** (observe os construtores da classe **Conta** – Código 2.10).

2.6.6 Instanciando classes

Em um programa Java, os objetos são criados (as classes são instanciadas) através do operador **new**. A execução do comando **new** cria um objeto de uma determinada classe e retorna uma referência a esse objeto.

Por exemplo, considere o Código 2.11, em que dois objetos da classe **Conta** são criados. É importante ressaltar que, nas linhas 8 e 9, as instâncias da classe **Conta** são criadas através da invocação de diferentes construtores. Ou seja, o objeto **c1** é criado com **saldo** R$ 0,00 (primeiro construtor), enquanto o objeto **c2** é criado com **saldo** R$ 400,00 (segundo construtor).

```java
/**                                                                    1
 * Classe que manipula objetos da classe Conta                         2
 *                                                                     3
 * @author Delano Medeiros Beder                                       4
 */                                                                    5
public class AcessaConta {                                             6
   public static void main(String args[]) {                           7
       Conta c1 = new Conta(); // Invoca construtor sem parâmetro      8
       Conta c2 = new Conta(400); // Invoca construtor com parâmetro saldo   9
       c1.deposito(200);                                               10
       System.out.println("Conta c1: " + c1.getSaldo());               11
       c2.saque(100);                                                  12
       System.out.println("Conta c2: " + c2.getSaldo());               13
   }                                                                   14
}                                                                      15
```

Código 2.11 Classe **AcessaConta** que manipula objetos da classe **Conta**.

Após criar um objeto, é possível manipular seus dados acessando os atributos e métodos desse objeto através do operador ponto (**.**). Considere novamente o Código 2.11, que, após criar os dois objetos, modifica e imprime os seus respectivos saldos (linhas 10 a 13).

Fica como exercício para o leitor: quais valores serão impressos ?

2.6.7 Atributos e métodos estáticos

Até o presente momento, apenas foi discutido como definir atributos e métodos de instâncias. Cada instância (objeto) tem seus próprios atributos, e uma modificação nos atributos de um objeto não afeta os atributos de outros objetos.

Porém, suponha que o sistema de gerenciamento de contas correntes queira controlar a quantidade de contas existentes. Seria interessante que houvesse a possibilidade de declararmos uma variável, para armazenar esse valor, que fosse única e compartilhada por todas as instâncias da classe `Conta`. Ou seja, esse atributo seria o mesmo para todos os objetos (instâncias da classe), e uma mudança nesse atributo seria visível para todos os objetos instanciados.

Dessa forma, a solução mais indicada seria utilizar um atributo **`static`** para armazenar a quantidade de contas criadas. Na linguagem Java, ao declarar um atributo como **`static`**, este passa a não ser mais um atributo de cada objeto, e sim um atributo da classe, a informação fica guardada pela classe, não é mais individual para cada objeto. Analogamente, é possível definir métodos **`static`** que só podem operar sobre atributos **`static`**.

```java
/**
 * Classe Conta Corrente
 *
 * @author Delano Medeiros Beder
 */
public class Conta {

    protected float saldo; // Atributo da classe que representa o saldo da conta

    private static int quantidade; // Armazena a quantidade de contas criadas

    /*
     * Declaração dos construtores da classe
     */
    public Conta() {
        Conta.quantidade++;
        this.saldo = 0;
    }

    public Conta(float saldo) {
        Conta.quantidade++;
        this.saldo = saldo;
    }

    // Métodos omitidos -- Ver Código 2.11

    public static int getQuantidade() {
        return Conta.quantidade;
    }
}
```

Código 2.12 Classe `Conta` – Implementação final.

O Código 2.12 apresenta a implementação da classe **Conta** em que atributos e métodos estáticos foram adicionados. É importante mencionar que, para acessar um atributo ou chamar um método estático, não se utiliza a palavra **this** (ou uma referência para uma instância da classe Conta), e sim o nome da classe.

2.6.8 Palavra reservada **final**

Na linguagem de programação Java, a palavra reservada **final** é utilizada em vários contextos diferentes, conforme discutido a seguir.

Uma classe final não pode ter subclasses. Isso é feito geralmente por razões de segurança e eficiência. Muitas das classes da biblioteca padrão do Java são finais, por exemplo **java.lang.System** e **java.lang.String**. Todos os métodos em uma classe final são implicitamente finais. O Código 2.13 apresenta um exemplo de uma classe final que não pode ter subclasses. O compilador apresentará um erro de compilação ao tentar definir uma subclasse para a classe final.

```
public class final FinalClass {...}

// Erro de compilação: classe final não pode ter subclasses
public class Wrong extends FinalClass {...}
```

Código 2.13 Exemplo: classe final.

Um método final não pode ser reimplementado pelas subclasses. Isso é usado para evitar que uma subclasse altere inesperadamente o comportamento de um método que pode ser crucial para a funcionalidade provida pela superclasse. O Código 2.14 apresenta um exemplo da classe **Base** que define um método final **m2()**. A classe **Derivada** é subclasse da classe **Base** e tenta reimplementar o método final **m2()**. Novamente o compilador apresentará um erro de compilação – um método final não pode ser reimplementado.

```
public class Base {
    public void m1() {...}
    public final void m2() {...}
}

public class Derivada extends Base {
    public void m1() {...} // Reimplementação OK !
    public void m2() {...}  // Erro de compilação: método final não pode ser reimplementado
}
```

Código 2.14 Exemplo: método final.

Um atributo final não pode ter seu valor modificado, ou seja, define valores constantes. O valor do atributo deve ser definido no momento da declaração (ou no construtor da classe), pois não é permitida nenhuma atribuição em outro momento. Permitir que o valor do atributo final seja definido no construtor introduz um maior grau de flexibilidade na definição de constantes para objetos de uma classe, uma vez que estas podem depender de parâmetros passados para o construtor. O Código 2.15 apresenta a classe `Circulo`, que possui atributos finais os quais são inicializados ou na declaração ou no construtor da classe.

```
public class Circulo {

    public static final double PI = 3.14; // Atributo final com valor atribuído na declaração

    public final double r;
    public final double x;
    public final double y;

    Circulo(double r, double x, double y) {
        this.r = r;
        this.x = x; // Atributos finais com valores atribuídos no construtor
        this.y = y;
    }

    [...]
}
```

Código 2.15 Exemplo: atributos finais.

A utilização de `final` para uma referência a objetos é permitida. Como no caso de constantes, a definição do valor (ou seja, a criação do objeto) também deve ser especificada no momento da declaração. No entanto, é preciso ressaltar que o conteúdo do objeto em geral pode ser modificado (através da chamada de métodos) – apenas a referência é fixa. O mesmo é válido para *arrays*. Argumentos de um método que não devem ser modificados podem ser declarados como final, também, na própria lista de parâmetros.

2.7 Considerações finais

Esta unidade apresentou a linguagem de programação Java ressaltando como os conceitos básicos inerentes ao paradigma orientado a objetos foram definidos nessa linguagem.

A próxima unidade discute como conceitos mais avançados do paradigma orientado a objetos, tais como classe abstrata, interface, pacotes e tratamento de exceções, são definidos na linguagem de programação Java.

2.8 Estudos complementares

Para estudos complementares sobre os tópicos abordados nesta unidade, o leitor interessado pode consultar as seguintes referências:

ARNOLD, K.; GOSLING, J.; HOLMES, D. *The Java Programming Language*. 4. ed. Boston: Addison-Wesley, 2005.

CAELUM. *Apostila do curso FJ-11* – Java e Orientação a Objetos. 2014. Disponível em: http://www.caelum.com.br/apostila-java-orientacao-objetos. Acesso em: 12 ago. 2014.

CAMARÃO, C.; FIGUEIREDO, L. *Programação de Computadores em Java*. 1. ed. São Paulo: LTC, 2003.

DEITEL, P.; DEITEL, H. *Java*: Como programar. 8. ed. São Paulo: Pearson Brasil, 2010.

GOLDMAN, A.; KON, F.; SILVA, P. J. *Introdução à Ciência da Computação com Java e Orientação a Objetos*. 1. ed. São Paulo: IME-USP, 2006. Disponível em: http://ccsl.ime.usp.br/files/books/intro-java-cc.pdf. Acesso em: 12 ago. 2014.

SANTOS, R. *Introdução à Programação Orientada a Objetos usando Java*. Rio de Janeiro: Campus, 2003.

UNIDADE 3

Conceitos avançados de orientação a objetos em Java

3.1 Primeiras palavras

Na unidade anterior, foram dados os primeiros passos no estudo da programação orientada a objetos em Java ao ressaltar como os conceitos básicos do paradigma orientado a objetos são definidos na linguagem de programação Java. No entanto, para sistemas mais complexos, apenas os conceitos discutidos na unidade anterior não são suficientes.

Dessa forma, esta unidade tem como objetivo aprofundar o estudo sobre a programação orientada a objetos ao apresentar como conceitos mais avançados do paradigma orientado a objetos, tais como classes abstratas, interfaces, pacotes e tratamento de exceções, são definidos na linguagem de programação Java.

3.2 Problematizando o tema

Ao final desta unidade, espera-se que o leitor seja capaz de reconhecer e definir precisamente os conceitos relacionados ao paradigma orientado a objetos na linguagem de programação Java. Dessa forma, esta unidade pretende discutir as seguintes questões:

- Como são definidos os conceitos (classes, objetos, interfaces, herança, etc.) inerentes ao paradigma orientado a objetos na linguagem de programação Java?
- O que são pacotes? Para que servem?
- Como *arrays* são definidos na linguagem de programação Java?
- Quais são as semelhanças e diferenças entre esses conceitos: classe concreta, classe abstrata e interface?
- Quais as principais características do mecanismo de tratamento de exceções presente na linguagem de programação Java?

3.3 Arrays

Um *array* em Java é tratado como um objeto, afinal, ele advém da classe `Array`. Eles são agregados homogêneos que podem guardar valores primitivos ou referências para objetos. É importante mencionar que *arrays* não podem mudar de tamanho. A partir do momento que um *array* foi criado, ele não pode mudar de tamanho. Se for

necessário mais espaço, é obrigatório criar um novo *array* e, antes de se referir a ele, copiar os elementos do *array* antigo.

A declaração de um *array*, como

```
int[] a;
```

apenas cria uma referência para um *array* de inteiros – porém, o *array* não foi criado. Assim como objetos, *arrays* são criados com a palavra reservada **`new`**:

```
a = new int[10];
```

Esse comando cria espaço para armazenar 10 inteiros no *array* **a**. Porém, os dois comandos poderiam ter sido combinados em um único comando, incluindo a declaração e a criação de espaço:

```
int[] a = new int[10];
```

Alternativamente, *arrays* podem ser criados com a especificação de algum conteúdo:

```
int[] b = {1, 3, 5, 7, 9, 2, 4, 6, 8};
```

O acesso a elementos individuais de um *array* é especificado através de um índice inteiro. O elemento inicial, assim como em C e C++, tem índice 0. Assim, do exemplo acima, a expressão **b[3]** acessa o conteúdo da quarta posição. O acesso a elementos do *array*, além do último índice permitido – por exemplo, **b[9]** –, gera um erro em tempo de execução ou uma exceção (Seção 3.10).

3.3.1 Percorrendo um *array*

Todo *array* em Java tem um atributo que se chama **`length`**, e pode-se acessá-lo para saber o tamanho do *array* ao qual se está referenciando naquele momento:

A versão 1.5 da plataforma Java trouxe uma nova sintaxe para percorrer *arrays* (e coleções, que serão discutidas na Unidade 5) – o **enhanced-for**.

```
void imprimeArray(int[] a) {
    for (int i = 0; i < a.length; i++) {
      System.out.println(a[i]);
    }
}
```

Código 3.1 Percorrendo *array*: utilizando o atributo `length`.

```
void imprimeArray(int[] a) {
    for (int x : a) {
      System.out.println(x);
    }
}
```

Código 3.2 Percorrendo *array*: **enhanced-for**.

3.3.2 Métodos com argumentos variáveis

A versão 1.5 da plataforma Java trouxe um recurso muito comum em diversas outras linguagens: número variável de argumentos na chamada de métodos. A sintaxe é simples e intuitiva: `Tipo... Nome`. Em que `Tipo` é o tipo do argumento (primitivo ou uma classe) e `Nome` é o nome do argumento. No corpo do método, o parâmetro de tamanho variável é tratado como um *array*.

```
1  public class VarArgs {
2
3      public static int soma(int... valor) {
4          int soma = 0;
5          for (int v : valor) {
6              soma += v;
7          }
8          return soma;
9      }
10
11     public static void main(String[] args) {
12         System.out.println(VarArgs.soma(1, 3, 5, 7));
13         System.out.println(VarArgs.soma(2, 4, 6));
14     }
15 }
```

Código 3.3 Método com argumentos variáveis.

O Código 3.3 apresenta o método `soma(int... valor)`, que retorna a soma dos valores inteiros passados como parâmetro. As linhas 12 e 13 apresentam duas invocações desse método com diferentes números de parâmetros.

3.4 Pacotes

Pacotes são utilizados para organizar as classes de funcionalidades similares ou relacionadas. Pacotes, a grosso modo, são apenas pastas ou diretórios do sistema operacional onde ficam armazenados os arquivos-fonte de Java e são essenciais para o conceito de encapsulamento, no qual são dados níveis de acesso às classes. Java possui um pacote padrão (*default*), que é utilizado quando não se define nenhum pacote, embora não seja recomendado usá-lo.

Definindo Pacotes. Para se definir a qual pacote uma classe pertence, pode-se utilizar a palavra reservada `package`, que obrigatoriamente deve ser a primeira linha de comando da classe a ser compilada.

Importando Pacotes. Java possui vários pacotes com outros pacotes internos e várias classes já prontas para serem utilizadas.

Dentre os pacotes Java, podemos determinar dois grandes pacotes: o pacote `java`, que possui as classes padrões da linguagem de programação Java; e o pacote `javax`, que possui pacotes de extensão os quais fornecem classes e objetos que implementam ainda mais o pacote `java`.

Exemplo: o pacote `AWT` (*Abstract Windowing Toolkit*) possui as classes necessárias para se criar um ambiente gráfico *Desktop* (Janelas) e está fortemente ligado às funções nativas do sistema operacional, ou seja, ele pertence ao pacote `java`. Mas o pacote `Swing` não é ligado fortemente às funções nativas do sistema operacional, mas sim às funcionalidades do `AWT`, ou seja, `Swing` complementa o `AWT`; portanto, `Swing` faz parte do pacote de extensão `javax`.

Para utilizar as milhares de classes contidas nos inúmeros pacotes de Java, devemos ou nos referenciar diretamente à classe ou importá-la. Para importar um pacote, usamos o comando `import`. Para separar um pacote de seu subpacote, usamos um ponto (como no acesso a membros de classe). Ao importarmos um pacote, podemos utilizar um coringa, o asterisco (*). O asterisco serve para importar todos os subpacotes e classes do pacote que está definido.

```
import java.awt.*; // Isso importará todos os subpacotes pertencentes ao pacote AWT.
```

Ou podemos definir diretamente qual pacote desejamos.

```
import javax.swing.JOptionPane; // Isso importará apenas a classe JOptionPane do pacote Swing.
```

A diferença entre as duas formas de importação de pacotes é o consumo de recursos do computador. Como o asterisco importa todos os subpacotes, o consumo de memória será alto, e, muito provavelmente, não usaremos todas as classes de todos os pacotes importados. Por isso, o recomendado é sempre importar apenas as classes que serão utilizadas.

3.4.1 Pacote `java.lang`

O pacote `java.lang` contém as classes que constituem recursos básicos da linguagem, necessários à execução de qualquer programa Java. O pacote `java.lang` é o único automaticamente importado pelo compilador. Ou seja, as classes pertencentes a esse pacote não precisam ser importadas.

Dentre as classes desse pacote, destacam-se:

`Object` – expressa o conjunto de funcionalidades comuns a todos os objetos Java. É a raiz da hierarquia de classes da linguagem Java;

`Class` e `ClassLoader` – representam classes Java e o mecanismo para carregá-las dinamicamente;

`String`, `StringBuffer` e `StringBuilder` – permitem a representação e a manipulação de *strings* (Seção 3.4.1.1), fixas ou modificáveis;

`Math` – contém a definição de métodos para cálculo de funções matemáticas (trigonométricas, logarítmicas, exponenciais, etc.) e de constantes;

`Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float` e `Double` – classes *wrappers* (Seção 2.4) que permitem a manipulação de valores dos tipos primitivos da linguagem como se fossem objetos;

`System`, `Runtime` e `Process` – são classes que permitem interação da aplicação com o ambiente de execução;

`Thread`, `Runnable` e `ThreadGroup` – são classes que dão suporte à execução de múltiplas linhas de execução;

`Throwable`, `Error` e `Exception` – são classes que permitem a definição e manipulação de situações de erros e condições inesperadas de execução (Seção 3.10), tais como `OutOfMemoryError`, `ArithmeticException` (por exemplo, divisão inteira por zero) e `ArrayIndexOutOfBoundsException` (acesso a elemento de *array* além da última posição ou antes da primeira posição).

3.4.1.1 String

Ao contrário do que ocorre nas linguagens C e C++, *strings* em Java não são tratadas como sequências de caracteres terminados em NULL. *Strings* são objetos, instâncias da classe **java.lang.String**.

Uma *string* pode ser criada como ilustrado abaixo. O operador +[1]pode ser utilizado para concatenar *strings*. Se, em uma mesma expressão, o operador + combinar *strings* e valores numéricos, os valores numéricos serão convertidos para *strings* e, então, concatenados.

```
int i = 9;
String s1 = "abc";
String s2 = "def";
String s3 = s1 + s2 + i;
```

Além disso, existem alguns métodos definidos na classe **String**. Dentre eles, podem-se citar:

char charAt(int index) – devolve o caractere na posição **index**. Os índices em uma *string* vão de zero ao seu tamanho menos um.

String toUpperCase() – devolve a *string* convertida para letras maiúsculas.

```
String s = "caneta";
System.out.println(s.charAt(0)); // imprime c
System.out.println(s.charAt(4)); // imprime t
System.out.println(s.toUpperCase()); // imprime CANETA
```

boolean endsWith(String suffix) – verifica se a *string* acaba com o sufixo dado. É usado, entre outras coisas, para verificar as extensões dos arquivos.

int indexOf(char ch) – devolve o índice da primeira ocorrência do caractere **ch** na *string* ou -1, caso o caractere não ocorra na *string*.

int length() – devolve o tamanho da *string*.

```
String s1 = "Programa.java";
System.out.println(s1.endsWith(".java")); // imprime true
System.out.println(s1.indexOf('.')); // imprime 8
System.out.println(s1.indexOf('s')); // imprime -1
System.out.println(s1.length()); // imprime 13
```

[1] Analogamente, o método **String concat(String str)** pode ser usado para concatenar *strings*.

int compareTo(String outra) – compara a *string* corrente com o parâmetro **outra**. Devolve um número positivo se o parâmetro **outra** for menor, 0 se forem iguais, e um negativo caso contrário.

```
String s1 = "mesa";
String s2 = "caneta";
System.out.println(s1.compareTo(s2)); // imprime 10
```

Para saber mais detalhes sobre a classe **java.lang.String**, o leitor interessado pode consultar o link: http://docs.oracle.com/javase/7/docs/api/java/lang/String.html.

3.4.1.2 String, StringBuider e StringBuffer

As três classes **String**, **StringBuilder** e **StringBuffer** são usadas para armazenar uma *string*, mas apresentam algumas diferenças. As classes **StringBuilder** e **StringBuffer** vieram para solucionar e/ou dar uma alternativa para alguns aspectos da classe **String** (como imutabilidade).

Aparentemente, objetos do tipo **String** são como qualquer outro objeto, mas eles são imutáveis. Ou seja, após ser atribuído um valor ao objeto, aquele nunca mais será modificado. Isto é válido para o valor do objeto e não para a referência do objeto. A variável de referência pode referenciar outra instância da classe **String**, a qualquer momento.

Observe o trecho de código abaixo. O valor impresso na linha 3 será **ca**, pois, como os objetos da classe **String** são imutáveis, o objeto referenciado por **s1** não modificou o seu valor após a execução do método **concat**. Porém, o valor impresso na linha 6 será **mesa**, pois, ao executar **s2 = s2.concat("sa")**, uma nova instância da classe **String** foi criada na memória (*heap*), e este novo objeto foi atribuído à variavel de referência **s2**.

```
1. String s1 = "ca";
2. s1.concat("sa"); // concatena, porém não altera o valor de s1
3. System.out.println(s1); // imprime ca
4. String s2 = "me";
5. s2 = s2.concat("sa"); // novo objeto é criado. s2 aponta para o novo objeto criado
6. System.out.println(s2); // imprime mesa
```

Observação: o outro objeto com o valor **me** foi para um local especial da memória, chamado *String constant pool*. No momento da criação de um objeto da classe `String`, o compilador verifica a existência no *String constant pool* de um objeto `String` idêntico. Se existir, a variável de referência apontará para esse objeto `String` já existente, e não será criado um novo objeto na memória.

Como discutido, objetos `String` são imutáveis. Dessa forma, vários objetos `String` podem ser abandonados no *String constant pool* como consequência de muitas modificações a um objeto `String`. As classes `StringBuilder` e `StringBuffer` foram projetadas para solucionar esse problema. Objetos dessas classes são mutáveis. Ou seja, podem ser modificados, e não será criado um novo objeto a cada modificação, mas o valor do objeto será realmente modificado.

```
StringBuilder s = new StringBuilder("ca");
s.append("sa"); // concatena e altera o valor de s
System.out.println(s); // imprime casa
```

As classes `StringBuilder` e `StringBuffer` são similares. A única diferença é que os métodos da classe `StringBuilder` não são **synchronized**. Ou seja, a classe `StringBuilder` é indicada quando não é necessário ter o controle sobre as *threads* (a maioria dos casos) que irão modificar o conteúdo da instância.

3.5 Associação, Composição e Agregação

Conforme discutido, além do relacionamento "é um" definido pelo mecanismo de herança (a ser discutido na Seção 3.6), é possível outros três tipos de relacionamento entre classes: associação, composição e agregação.

A associação *para um* pode ser armazenada em um atributo na classe de origem da associação, e seu tipo deve ser a classe de destino. Como exemplo, considere a associação *para um* entre as classes `Carro` e `Pessoa` apresentada na Figura 1.10. Essa associação pode ser armazenada no atributo `dono` da classe `Carro` (Código 3.4).

A associação *para muitos* corresponde à implementação de uma estrutura de dados que representa uma coleção. Como exemplo, considere a associação *para muitos* entre as classes `Pessoa` e `Carro` apresentada na Figura 1.10. A escolha mais óbvia seria implementar a coleção como um *array* de carros, conforme apresentado no Código 3.4. No entanto, a Unidade 5 discute o *framework* de coleções Java que poderia ser utilizado na implementação da coleção de carros.

```java
/**
 * Classe Carro
 *
 * @author Delano Medeiros Beder
 */
public class Carro {

    /*
     * Declaração dos atributos da classe
     */
    private String marca, cor;
    private int ano;
    private Pessoa dono;

    /* Construtor e métodos omitidos */
}

/**
 * Classe Pessoa
 *
 * @author Delano Medeiros Beder
 */
public class Pessoa {

    /*
     * Declaração dos atributos da classe
     */
    private String CPF, nome;
    private Carro[] carros;

    /* Construtor e métodos omitidos */
}
```

Código 3.4 Classes `Carro` e `Pessoa`.

A implementação dos relacionamentos de agregação e composição é bastante semelhante à implementação do relacionamento de associação *para muitos*. O que diferencia é a semântica inerente aos relacionamentos de agregação e composição. Por exemplo, na implementação do relacionamento de composição, é necessária a presença de métodos que destroem os objetos "parte" quando o objeto "todo" for destruído, visto que a existência dos objetos "parte" não faz sentido se o objeto "todo" não existir.

Para mais detalhes sobre a implementação desses relacionamentos, sugere-se que o leitor consulte Wazlawick (2011), que consta na Seção 3.12.

3.6 Herança

Conforme discutido na Seção 1.4.8, o mecanismo de herança define o relacionamento "é um" entre classes no qual uma classe, denominada *subclasse*, herda todos os comportamentos (métodos) e estados (atributos) de outra classe, denominada *superclasse*.

A forma básica de herança em Java é a extensão simples entre uma superclasse e sua classe derivada. Para tanto, utiliza-se na definição da classe derivada a palavra reservada **extends** seguida pelo nome da superclasse.

A hierarquia de classes de Java tem como raiz uma classe básica, `Object`. Quando não for especificada uma superclasse na definição de uma classe, o compilador assume que a superclasse é `Object`.

É por esse motivo que todos os objetos podem invocar os métodos da classe `Object`, tais como **equals()**, **hashCode()** e **toString()**. O método **equals()** permite comparar objetos por seus conteúdos. A implementação padrão desse método realiza uma comparação de conteúdo bit a bit; se um comportamento distinto for desejado para uma classe definida pelo programador, o método deve ser redefinido. O método **hashCode()** retorna um código *hash* de um objeto. Sempre que se sobrescrever o método **equals()**, sobrescreve-se também o método **hashCode()**. Ele é usado normalmente para agilizar a busca em coleções Java (Unidade 5). O método **toString()** permite converter uma representação interna do objeto em uma *string* que pode ser apresentada ao usuário.

```java
/**
 * Classe Conta Cheque Especial
 * @author Delano Medeiros Beder
 */
public class ContaChequeEspecial extends Conta{
    /*
     * Declaração dos atributos da classe
     */
    private float credito; // Atributo da classe que representa o crédito disponível

    /*
     * Construtor da classe
     */

    public ContaChequeEspecial(float saldo, float credito) {
        super(saldo); // invoca o construtor Conta(float saldo) da classe Conta
        this.credito = credito;
    }

    /*
     * Declaração dos métodos da classe
     */

    public boolean saque (float valor) {
        boolean ok = false;
        if (saldo + credito >= valor) {
            saldo -= valor; // saldo = saldo - valor
            ok = true;
        }
        return ok;
    }
}
```

Código 3.5 Classe `ContaChequeEspecial` com atributos e métodos.

Como exemplo do mecanismo de herança em Java, esta seção apresenta a implementação da classe `ContaChequeEspecial` (Figura 1.7), que é subclasse da classe `Conta` (Código 2.10).

Como se pode observar, a classe `ContaChequeEspecial` (Código 3.5):

(a) define um construtor que invoca, através da palavra reservada `super`, um construtor específico da superclasse. A invocação do construtor da superclasse, se presente, deve ser o primeiro comando do construtor, pois o início da construção do objeto é a construção da parte da superclasse. Caso não esteja presente, está implícita uma invocação para o construtor padrão, sem argumentos, da superclasse, equivalente à forma `super()`.

(b) adiciona o atributo `credito`, que é responsável por armazenar o valor do crédito disponível ao correntista;

(c) adiciona o método `getCredito()`, que é responsável por retornar o valor do crédito disponível. Ou seja, esse método retorna o valor do atributo `credito`;

(d) redefine o método `saque()` para levar em consideração o crédito disponível ao correntista. Ou seja, saques serão permitidos mesmo que o saldo fique negativo, contanto que não ultrapasse o crédito disponível. É importante salientar que, visto que o atributo `saldo` da classe `Conta` foi definido com o modificador `protected`, ele pode ser acessado pela classe `ContaChequeEspecial`.

3.7 Classes abstratas

Conforme discutido na Seção 1.4.9, uma classe abstrata representa entidades e conceitos abstratos e é sempre uma superclasse que não possui instâncias. Com o intuito de ilustrar melhor esse conceito, esta seção apresenta um exemplo hipotético do desenvolvimento de um sistema de gerenciamento de obras de arte de um museu.

Suponha que você faz parte de uma equipe de desenvolvimento que foi contratada para desenvolver um sistema de gerenciamento de obras de arte de um museu. Após a fase de levantamento de requisitos do sistema, a equipe identificou que as obras de arte são apenas de duas categorias – pinturas e esculturas – e que compartilham algumas características em comum: título, artista, material (tela, madeira, papel, etc.) e ano de criação. Dessa forma, a modelagem orientada a objetos do sistema é composta das 4 classes (Figura 3.1) discutidas nas próximas seções.

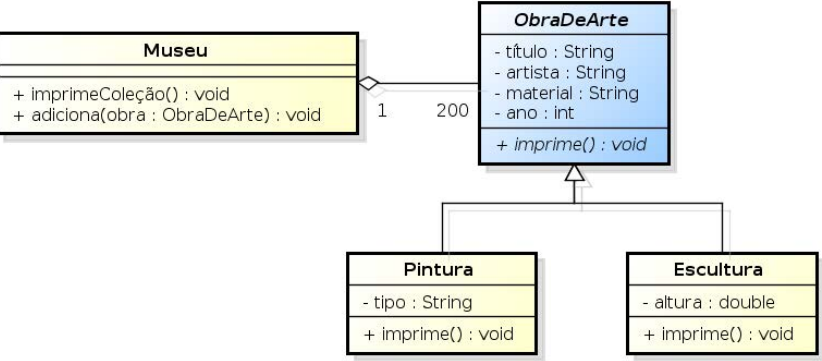


Figura 3.1 Gerenciamento de obras de arte de um museu.

3.7.1 Classe `ObraDeArte`

A classe abstrata `ObraDeArte` representa obras de arte, é a raiz da hierarquia de obras de artes e possui alguns atributos para armazenar informações sobre as obras de arte: título, artista, material (tela, madeira, papel, etc.) e ano de criação. Pelo Código 3.6, pode-se observar que a classe `ObraDeArte`:

- é uma classe abstrata. A palavra reservada `abstract` foi utilizada para declarar que a classe `ObraDeArte` é abstrata (Código 3.6, linha 6).
- define um construtor que inicializa as informações sobre os obras de arte: título, artista, material e ano de criação.
- define um método abstrato `imprime()` que deve ser implementado pelas subclasses para apresentar as informações sobre as obras de arte. A palavra reservada `abstract` foi utilizada novamente para declarar que o método é abstrato (Código 3.6, linha 27).

```
/**
 * Classe ObraDeArte
 *
 * @author Delano Medeiros Beder
 */
public abstract class ObraDeArte {

    /*
     * Declaração dos atributos da classe
     */
    private String título, artista, material;
    private int ano;

    /*
     * Declaração do construtor da classe
     */
    public ObraDeArte(String título, String artista, String material, int ano) {
        this.título = título;
        this.artista = artista;
        this.material = material;
        this.ano = ano;
    }

    /*
     * Declaração dos métodos da classe
     */
    public abstract void imprime();

    /* Métodos getters (getTítulo(), getArtista(), getMaterial() e getAno()) omitidos */

}
```

Código 3.6 Classe abstrata `ObraDeArte`.

3.7.2 Classe `Pintura`

A classe **`Pintura`** representa pinturas e é subclasse da classe **`ObjetoDeArte`**. Essa classe define o seguinte atributo: tipo (óleo, aquarela, etc.). Pelo Código 3.7, pode-se observar que a classe **`Pintura`**:

- define um construtor que inicializa as informações sobre pinturas. É importante salientar que esse construtor invoca, através da palavra reservada **`super`**, o construtor da superclasse com o objetivo de inicializar os atributos da superclasse.

- implementa o método **`imprime()`**, responsável por apresentar as informações sobre as pinturas. Vale a pena mencionar que esse método invoca os métodos *getters* definidos pela classe **`ObraDeArte`** para acessar as características comuns a todas as obras de arte: título, artista, material e ano de criação.

```java
/**
 * Classe Pintura
 *
 * @author Delano Medeiros Beder
 */
public class Pintura extends ObraDeArte {

    /*
     * Declaração dos atributos da classe
     */
    private String tipo;

    /*
     * Declaração do construtor da classe
     */
    public Pintura(String título, String artista, String material, int ano, String tipo) {
        super(título, artista, material, ano);
        this.tipo = tipo;
    }

    /*
     * Declaração dos métodos da classe
     */
    public void imprime() {
        System.out.println("Título    : " + this.getTítulo());
        System.out.println("Artista   : " + this.getArtista());
        System.out.println("Material  : " + this.getMaterial());
        System.out.println("Ano       : " + this.getAno());
        System.out.println("Tipo      : " + this.tipo);
    }
}
```

Código 3.7 Classe `Pintura`.

3.7.3 Classe `Escultura`

A classe **Escultura** representa esculturas e é subclasse da classe **ObjetoDeArte**. Essa classe define o seguinte atributo: altura. Pelo Código 3.8, pode-se observar que a classe `Escultura`:

- define um construtor que inicializa as informações sobre esculturas. Análogo ao construtor da classe `Pintura`, esse construtor invoca o construtor da classe `ObraDeArte`.
- implementa o método `imprime()`, responsável por apresentar as informações sobre as esculturas. Análogo ao método `imprime()` da classe `Pintura`, esse método também invoca os métodos *getters* definidos pela classe `ObraDeArte`.

```java
/**
 * Classe Escultura
 *
 * @author Delano Medeiros Beder
 */
public class Escultura extends ObraDeArte {

    /*
     * Declaração dos atributos da classe
     */
    private double altura;

    /*
     * Declaração do construtor da classe
     */
    public Escultura(String título, String artista, String material, int ano, double altura) {
        super(título, artista, material, ano);
        this.altura = altura;
    }

    /*
     * Declaração dos métodos da classe
     */
    @Override
    public void imprime() {
        System.out.println("Título    : " + this.getTítulo());
        System.out.println("Artista   : " + this.getArtista());
        System.out.println("Material  : " + this.getMaterial());
        System.out.println("Ano       : " + this.getAno());
        System.out.println("Altura    : " + this.altura);
    }
}
```

Código 3.8 Classe `Escultura`.

3.7.4 Classe `Museu`

A classe **Museu** representa museus e, dessa forma, armazena uma coleção de obras de arte (instâncias das subclasses de `ObraDeArte`) presentes no acervo do museu. Pelo Códígo 3.9, pode-se observar que a classe `Museu`:

- define o atributo `obras`, que é responsável por armazenar uma coleção de obras de arte presentes no acervo do museu.

  Nessa primeira implementação, utiliza-se um *array* de instâncias da classe `ObraDeArte`. A principal deficiência dessa implementação é que não é possível redimensionar o tamanho de *arrays* após estes serem criados. Dessa forma, essa implementação adota o tamanho máximo de 200 obras de arte. A Unidade 5 discute a *API* de classes e interfaces conhecida como *framework* de coleções, que auxilia a manipulação de coleções (listas, conjuntos, etc.) e outras estruturas de dados.

- define o atributo `quantidade`, que é responsável por armazenar a quantidade de obras de arte presentes no acervo do museu.

- define o método **adicionaObra()**, que é responsável por incluir, respeitando o limite de 200 obras, uma nova obra de arte no acervo do museu.

  É importante salientar que esse método utiliza o princípio de substituição, discutido na Seção 1.4.8. Esse princípio consiste na possibilidade de substituir a superclasse pelas subclasses em qualquer trecho de código que exija a superclasse. No caso desse método, o argumento é uma instância de **ObraDeArte**. Porém, a classe **ObraDeArte** é abstrata e, portanto, não possui instâncias. Dessa forma, as obras de arte a serem adicionadas no acervo do museu serão instâncias das classes concretas (**Pintura** e **Escultura**) – subclasses de **ObraDeArte**.

- define o método **imprimeColeção()**, que é responsável por apresentar as informações relacionadas às obras de arte presentes no acervo do museu. Como se pode observar, esse método invoca o método **imprime()** dos objetos que representam as obras de arte – instâncias das classes **Pintura** e **Escultura**.

```
/**
 * Classe Museu
 *
 * @author Delano Medeiros Beder
 */
public class Museu {

    /*
     * Declaração dos atributos da classe
     */
    private ObraDeArte[] obras;

    private int quantidade;

    /*
     * Declaração do construtor da classe
     */
    public Museu() {
        obras = new ObraDeArte[200];
        quantidade = 0;
    }

    /*
     * Declaração dos métodos da classe
     */
    public void adicionaObra(ObraDeArte obra) {
        if (quantidade < 200) {
            obras[quantidade++] = obra;
        }
    }

    public void imprimeColeção() {
        for (int i = 0; i < quantidade; i++) {
            obras[i].imprime();
        }
    }
}
```

Código 3.9 Classe Museu.

3.8 Interfaces

Conforme discutido na Seção 1.4.9, uma *interface* é uma coleção de declarações de métodos sem dados (sem atributos) e sem corpo. Ou seja, os métodos de uma interface são sempre vazios – são simples assinaturas de métodos.

Com o intuito de ilustrar melhor esse conceito, essa seção apresenta um exemplo da implementação de uma hierarquia de classes que representam figuras geométricas de duas dimensões: retângulo, triângulo e triângulo equilátero. Essas classes implementam a interface **Figura2D**, conforme a Figura 3.2.

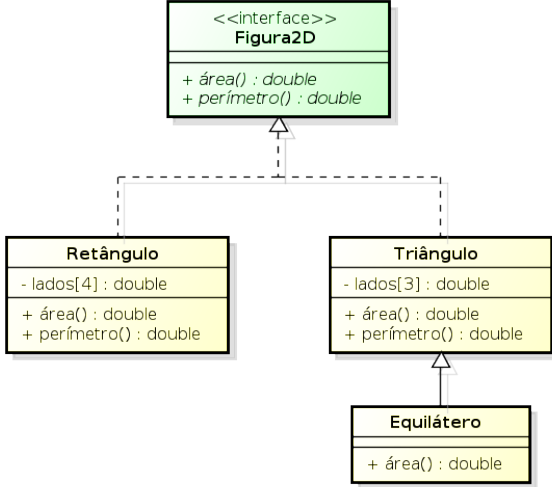


Figura 3.2 Interface **Figura2D** e classes que a implementam.

3.8.1 Interface Figura2D

Uma interface Java é sempre declarada com a palavra reservada **interface** seguida do nome da interface. O nome da interface não pode conter espaços e deve ser iniciado por uma letra. Nomes de interfaces não podem ser iguais às palavras reservadas de Java (apresentadas na Tabela 2.2).

O Código 3.10 apresenta a implementação da interface **Figura2D**, que declara os seguintes métodos: **double área()** e **double perímetro()**. Pode-se observar que os métodos declarados são sempre vazios – são simples assinaturas de métodos.

```
/**
 * Interface Figura2D
 *
 * @author Delano Medeiros Beder
 */
public interface Figura2D {

    /*
     * Definição dos métodos da interface
     */
    double área();

    double perímetro();
}
```

Código 3.10 Interface `Figura2D`.

3.8.2 Classe `Retângulo`

A classe **`Retângulo`** representa retângulos e implementa (através da palavra reservada **`implements`**) a interface **`Figura2D`**. Conforme discutido anteriormente, as classes concretas que implementam uma interface devem obrigatoriamente implementar todos os métodos declarados nela. Pelo Código 3.11, pode-se observar que a classe **`Retângulo`**:

- define um construtor que verifica se os valores passados como parâmetros representam um retângulo (4 lados, $l_1 = l_3$ e $l_2 = l_4$). É importante salientar que o construtor recebe um número variável de parâmetros, conforme discutido na Seção 3.3.2. Caso uma dessas condições não for satisfeita, o construtor levanta uma exceção (Seção 3.10).
- implementa o método **`double área()`**, que retorna a área de um retângulo. A fórmula para o cálculo da área de um retângulo é $l_1 * l_2$. Sendo l_1 a base, e l_2 a altura do retângulo.
- implementa o método **`double perímetro()`**, que retorna o perímetro de um retângulo. Ou seja, a soma do tamanho dos quatro lados. Desde que $l_1 = l_3$ e $l_2 = l_4$, a fórmula para o cálculo do perímetro de um retângulo é $2 * l_1 + 2 * l_2$.

```java
/**
 * Classe Retângulo
 *
 * @author Delano Medeiros Beder
 */
public class Retângulo implements Figura2D {

    private double[] lado;

    public Retângulo(double... l) {
        if (l.length != 4 || l[0] != l[2] || l[1] != l[3]) {
            throw new RuntimeException("Não é Retângulo");
        }

        this.lado = l;
    }

    public double área() {
        return lado[0] * lado[1];
    }

    public double perímetro() {
        return 2 * lado[0] + 2 * lado[1];
    }
}
```

Código 3.11 Classe `Retângulo`.

3.8.3 Classe `Triângulo`

A classe **`Triângulo`** representa triângulos e implementa (através da palavra reservada **`implements`**) a interface **`Figura2D`**. Pelo Código 3.12, pode-se observar que a classe **`Triângulo`**:

- define um construtor que verifica se os valores passados como parâmetros representam um triângulo (tem 3 lados e o tamanho de qualquer um dos lados é menor que a soma dos tamanhos dos outros dois). É importante salientar que o construtor recebe um número variável de parâmetros, conforme discutido na Seção 3.3.2. Caso uma dessas condições não seja satisfeita, o construtor levanta uma exceção (Seção 3.10).
- implementa o método **`double área()`**, que retorna a área de um triângulo. A fórmula para o cálculo da área de um triângulo é $\sqrt{s*(s-l_1)*(s-l_2)*(s-l_3)}$, em que o semiperímetro é $s = \frac{l_1+l_2+l_3}{2}$.
- implementa o método **`double perímetro()`**, que retorna o perímetro de um triângulo, ou seja, a soma do tamanho dos três lados.

```java
/**
 * Classe Triângulo
 *
 * @author Delano Medeiros Beder
 */
public class Triângulo implements Figura2D {
    protected double[] lado;

    public Triângulo(double... l) {
        if (l.length != 3 || l[0] > l[1] + l[2] || l[1] > l[0] + l[2] || l[2] > l[0] + l[1]) {
            throw new RuntimeException("Não é Triângulo");
        }
        this.lado = l;
    }

    public double área() {
        double s = perímetro() / 2;
        return Math.sqrt(s * (s - lado[0]) * (s - lado[1]) * (s - lado[2]));
    }

    public double perímetro() {
        return lado[0] + lado[1] + lado[2];
    }
}
```

Código 3.12 Classe `Triângulo`.

3.8.4 Classe `Equilátero`

A classe **`Equilátero`**, subclasse da classe **`Triângulo`**, representa triângulos equiláteros. Pelo Código 3.13, pode-se observar que a classe **`Equilatéro`**:

- define um construtor que invoca o construtor do pai, através da chamada **`super(l)`**, para verificar se os valores passados como parâmetro representam um triângulo. Logo após, ele verifica se os três lados são iguais (condição para ser um triângulo equilátero). Caso uma dessas condições não seja satisfeita, o construtor levanta uma exceção (Seção 3.10).
- implementa o método **`double área()`**, que retorna a área de um triângulo equilátero. A fórmula para o cálculo da área de um triângulo equilátero é $\frac{l^2 * \sqrt{3}}{4}$.

Vale a pena salientar que a classe **`Equilátero`** apenas redefine o método **`double área()`**, pois o comportamento é diferente da superclasse. Em relação ao método **`double perímetro()`**, ele herda e utiliza a implementação provida pela superclasse.

```java
/**
 * Classe Equilátero
 * @author Delano Medeiros Beder
 */
public class Equilátero extends Triângulo {

    public Equilátero(double... l) {
        super(l);

        if (l[0] != l[1] || l[0] != l[1] || l[1] != l[2]) {
            throw new RuntimeException("Não é Triângulo Equilátero");
        }
    }

    public double área() {
        return this.lado[0] * this.lado[0] * Math.sqrt(3) / 4;
    }
}
```

Código 3.13 Classe `Equilátero`.

3.9 Polimorfismo paramétrico

O polimorfismo paramétrico ou genérico atua em um tipo que é um parâmetro. Sob um ponto de vista mais amplo, esse tipo de polimorfismo permite ampliar os níveis de abstração da linguagem. Imagine um módulo que deva ser autocontido o suficiente para se comportar de forma distinta diante de parâmetros distintos, permitindo que a inclusão, alteração ou remoção de dados seja feita não importando qual a característica do dado. Esse tipo de abstração é conseguido com o polimorfismo paramétrico.

Normalmente, o polimorfismo paramétrico é muito usado na criação de estruturas genéricas o suficiente para que possam ser reaproveitadas para armazenar os mais diversos tipos de objetos possíveis.

Por exemplo, é bastante comum a necessidade de estruturas de dados flexíveis o bastante para se adaptarem a qualquer tipo. Uma vez definido o tipo, a estrutura de dados passa a aceitar somente objetos desse tipo. Este é o exemplo da estrutura de dados **Lista**, que, não importando o tipo (inteiros, caracteres, objetos, etc.), mantém operações de inclusão, remoção, busca, etc.

A versão 1.5 da plataforma Java trouxe a implementação de tipos genéricos, que permitem a implementação do polimorfismo paramétrico. Considere o exemplo (Código 3.14) em que se utilizam coleções genéricas (Unidade 5). Nesse código, os dois primeiros comandos (linhas 1 e 2) são responsáveis pela criação de duas listas genéricas (lista de instâncias da classe `String` e lista de instâncias da classe `Object`). É importante salientar que o objeto a ser criado é parametrizado com o tipo (`String` ou `Object`).

Logo após, há 4 comandos (linhas 4-14), que são responsáveis por adicionar elementos nas listas genéricas.

- O primeiro comando de adição (linhas 4 e 5) é responsável por adicionar uma *string* em uma lista de instâncias da classe **`Object`**. Visto que a classe **`String`** é subclasse de **`Object`**, o comando é aceito pelo compilador.

- O segundo comando de adição (linhas 7 e 8) é responsável por adicionar uma *string* em uma lista de instâncias da classe **`String`**. O comando obviamente é aceito pelo compilador.

- O terceiro comando de adição (linhas 10 e 11) é responsável por adicionar um inteiro em uma lista de instâncias da classe **`Object`**. Visto que o número inteiro **1** é convertido automaticamente em uma instância da classe **`Integer`** e a classe **`Integer`** é subclasse de **`Object`**, o comando é aceito pelo compilador.

- O segundo comando de adição (linhas 13 e 14) é responsável por adicionar um inteiro em uma lista de instâncias da classe **`String`**. De maneira análoga ao comando anterior, o número inteiro **1** é convertido automaticamente em uma instância da classe **`Integer`**. Porém, as classes **`Integer`** e **`String`** são incompatíveis. Ou seja, não há nenhum relacionamento de herança entre essas classes. Dessa forma, esse comando não é aceito pelo compilador.

```
LinkedList<Object> lista1 = new LinkedList<Object>(); // Criando uma lista de Object     1
LinkedList<String> lista2 = new LinkedList<String>(); // Criando uma lista de String     2
                                                                                          3
// Adição na lista de Object OK ! "Teste" é uma String. E toda String "é um" Object       4
lista1.add("Teste");                                                                      5
                                                                                          6
// Adição na lista de String OK ! "Teste" é uma String.                                   7
lista2.add("Teste");                                                                      8
                                                                                          9
 // Adição na lista de Object OK ! Auto-boxing int => Integer. E todo Integer "é um" Object 10
lista1.add(1);                                                                            11
                                                                                          12
// Erro de compilação ! Auto-boxing int => Integer. Um Integer "não é uma" String         13
lista2.add(1);                                                                            14
```

Código 3.14 Polimorfismo paramétrico – listas genéricas.

Para saber mais detalhes sobre classes genéricas em Java, o leitor interessado pode consultar o seguinte link: http://docs.oracle.com/javase/tutorial/java/generics/.

3.10 Exceções

Uma exceção é um sinal indicativo de que algum tipo de condição excepcional ocorreu durante a execução do programa. Assim, exceções estão associadas a condições de erro que não tinham como ser verificadas durante a compilação do programa.

As duas atividades associadas à manipulação de uma exceção são:

- Geração: a sinalização de que uma condição excepcional (por exemplo, um erro) ocorreu;
- Captura: a manipulação (tratamento) da situação excepcional, em que as ações necessárias para a recuperação da situação de erro são definidas.

Para cada exceção que pode ocorrer durante a execução do código, um bloco de ações de tratamento (um *exception handler*) deve ser especificado. O compilador Java verifica e obriga que toda exceção não trivial tenha um bloco de tratamento associado. O mecanismo de tratamento de exceções em Java, embora apresente suas particularidades, teve seu projeto inspirado no mecanismo equivalente da linguagem C++. É um mecanismo adequado ao tratamento de exceções síncronas, para situações em que a recuperação do erro é possível.

A sinalização da exceção é propagada a partir do bloco de código em que ela ocorreu através de toda a pilha de invocações de métodos, até que a exceção seja capturada por um bloco de ações de tratamento (*exception handler*). Eventualmente, se tal bloco não existir em nenhum ponto da pilha de invocações de métodos, a sinalização da exceção atinge o método `main()`, fazendo com que a máquina virtual Java apresente uma mensagem de erro e aborte sua execução.

3.10.1 Tratamento de exceções

A captura e o tratamento de exceções em Java se dão através da especificação de blocos `try`, `catch` e `finally`, definidos através destas mesmas palavras reservadas da linguagem. A estruturação desses blocos obedece a sintaxe apresentada no Código 3.15.

`XException` e `YException` deveriam ser substituídos pelo nome do tipo de exceção. Os blocos não podem ser separados por outros comandos — um erro de sintaxe seria detectado pelo compilador Java nesse caso. Cada bloco `try` pode ser

seguido por zero ou mais blocos **catch**, e cada bloco **catch** refere-se a uma única exceção.

```
try {
    // código que inclui comandos/invocações de métodos que podem gerar uma situação de exceção.
}
catch (XException ex) {
    // bloco de tratamento associado à condição de exceção XException ou a qualquer uma de suas
    // subclasses, identificada aqui pelo objeto ex
}
catch (YException ey) {
    // bloco de tratamento para a situação de exceção YException ou a qualquer uma de suas subclasses
}
finally {
    // bloco de código que sempre será executado após o bloco try, independentemente de sua conclusão
    // ter ocorrido normalmente ou ter sido interrompida
}
```

Código 3.15 Tratamento de exceções em Java.

O bloco **finally**, quando presente, é sempre executado. Em geral, ele inclui comandos os quais liberam recursos que eventualmente podem ter sido alocados durante o processamento do bloco **try** e que podem ser liberados, independentemente de a execução ter sido encerrada com sucesso ou ter sido interrompida por uma condição de exceção. A presença desse bloco é opcional.

Alguns exemplos de exceções já definidas no pacote **java.lang** incluem:

ArithmeticException: indica situações de erros em processamento aritmético, tal como uma divisão inteira por 0. A divisão de um valor real por 0 não gera uma exceção (o resultado é o valor infinito);

NumberFormatException: indica que se tentou a conversão de uma *string* para um formato numérico, mas seu conteúdo não representava adequadamente um número para aquele formato. É uma subclasse de **IllegalArgumentException**;

IndexOutOfBounds: indica a tentativa de acesso a um elemento de um agregado aquém ou além dos limites válidos. É a superclasse de **StringIndexOutOfBounds**, para *strings*, e de **ArrayIndexOutOfBoundsException**, para *arrays*;

NullPointerException: indica que a aplicação tentou usar uma referência, a um objeto, que ainda não foi definida;

ClassNotFoundException: indica que a máquina virtual Java tentou carregar uma classe, mas não foi possível encontrá-la durante a execução da aplicação.

Além disso, outros pacotes especificam suas exceções, referentes às suas funcionalidades. Por exemplo, no pacote **java.io**, define-se **IOException**, que indica a ocorrência de algum tipo de erro em operações de entrada e saída (Unidade 4). É a

superclasse das classes que representam condições de exceção mais específicas, tais como `EOFException` (fim de arquivo ou *stream*), `FileNotFoundException` (arquivo especificado não foi encontrado) e `InterruptedIOException` (operação de entrada ou saída foi interrompida).

Uma exceção contém pelo menos uma *string* que a descreve, que pode ser obtida pela invocação do método `getMessage()`, mas pode eventualmente conter outras informações. Por exemplo, `InterruptedIOException` inclui o atributo público do tipo inteiro, chamado `bytesTransferred`, que indica quantos *bytes* foram transferidos antes de a interrupção da operação ocorrer. Outra informação que pode sempre ser obtida de uma exceção é a sequência de métodos no momento da exceção, que pode ser obtida a partir da invocação do método `printStackTrace()`.

Como exceções fazem parte de uma hierarquia de classes, exceções mais genéricas (mais próximas do topo da hierarquia) englobam aquelas que são mais específicas. Assim, a forma mais genérica de um bloco `try-catch` é

```
try {
    ...
} catch (Exception e) {
    // código responsável pelo tratamento da exceção
}
```

pois todas as exceções são derivadas de `Exception`. Se dois blocos `catch` especificam exceções, o tratamento da exceção derivada (ex.: `FileNotFoundException`) deve preceder o da mais genérica (ex.: `IOException`).

3.10.2 Erro e `RuntimeException`

Exceções em Java consistem em um caso particular de um objeto da classe `Throwable`. Apenas objetos dessa classe ou de suas classes derivadas podem ser gerados, propagados e capturados através do mecanismo de tratamento de exceções.

Além de `Exception`, outra classe derivada de `Throwable` é a classe `Error`, a qual é a raiz das classes que indicam situações as quais a aplicação não tem como ou não deve tentar tratar. Usualmente indica situações anormais, que não deveriam ocorrer. Dentre os erros definidos em Java, no pacote `java.lang`, estão `StackOverflowError` e `OutOfMemoryError`. São situações em que não é possível

uma correção a partir de um tratamento realizado pelo próprio programa que está executando.

Há também exceções que não precisam ser explicitamente capturadas e tratadas. São aquelas derivadas de **RuntimeException**, uma classe derivada diretamente de **Exception**. São exceções que podem ser geradas durante a operação normal de uma aplicação, para as quais o compilador Java não irá exigir que o programador realize algum tratamento (ou que propague a exceção, como escrito na Seção 3.10.3). Nelas se incluem: **ArithmeticException**, **IllegalArgumentException**, **IndexOutOfBoundsException** e **NullPointerException**.

3.10.3 Propagando exceções

Embora toda exceção que não seja derivada de **RuntimeException** deva ser tratada, nem sempre é possível tratar uma exceção no mesmo escopo do método cuja invocação gerou a exceção. Nessas situações, é possível propagar a exceção para um nível acima na pilha de invocações.

Para tanto, o método que está deixando de capturar e tratar a exceção faz uso da cláusula **throws** em sua declaração:

```
void metodoQueNaoTrataExcecao() throws Exception {
    metodoQuePodeSinalizarExcecao();
}
```

Nesse caso, **metodoQueNaoTrataExcecao()** reconhece que em seu corpo há a possibilidade da sinalização de uma exceção, mas não se preocupa em realizar o tratamento dessa exceção em seu escopo. Em vez disso, ele repassa essa responsabilidade para o método anterior (o método que o invocou) na pilha de chamadas.

Eventualmente, também o outro método pode repassar a exceção adiante. Porém, pelo menos no método **main()**, as exceções deverão ser tratadas ou o programa terá sua interpretação interrompida.

3.10.4 Definindo e sinalizando exceções

Exceções são classes. Assim, é possível que uma aplicação defina suas próprias exceções através do mecanismo de definição de classes.

Por exemplo, considere que fosse importante para uma aplicação diferenciar a condição de divisão por zero de outras condições de exceções aritméticas. Nesse

caso, uma classe **DivideByZeroException** (Código 3.16) poderia ser criada. O argumento para o construtor da superclasse especifica a mensagem que seria impressa quando o método **getMessage()** for invocado para essa exceção.

Para levantar uma exceção durante a execução de um método, um objeto dessa classe deve ser criado e, através do comando **throw**, propagado para os métodos anteriores na pilha de execução (Código 3.16, linhas 10-15).

```
1  public class DivideByZeroException extends ArithmeticException {
2
3      public DivideByZeroException() {
4          super("O denominador na divisão inteira tem valor zero");
5      }
6  }
7
8  public class TesteDivisao {
9
10     public double divisao(double num, int den) throws DivideByZeroException {
11         if (den == 0) {
12             throw new DivideByZeroException();
13         }
14         return num / den;
15     }
16
17     public void usaDivisao(double x, int y) throws DivideByZeroException {
18         try {
19             ... // código anterior à invocação de divisão
20             divisao(x, y);
21             ... // código posterior à invocação de divisão
22         } catch (DivideByZeroException dbze) {
23             ... // tratamento parcial da exceção
24             throw dbze;
25         }
26     }
27 }
```

Código 3.16 Definindo e sinalizando exceções.

O mesmo comando **throw** pode ser utilizado para repassar uma exceção após sua captura – por exemplo, após um tratamento parcial da exceção (Código 3.16, linhas 17-26). Nesse caso, a informação associada à exceção (como o seu ponto de origem, registrado internamente no atributo do objeto que contém a pilha de invocações) é preservada.

3.11 Considerações finais

Esta unidade discutiu como conceitos mais avançados do paradigma orientado a objetos, tais como classes abstratas, interfaces, pacotes e tratamento de exceções, são definidos na linguagem de programação Java.

A próxima unidade discute como os conceitos relacionados às operações de entrada e saída são definidos na linguagem de programação Java.

3.12 Estudos complementares

Para estudos complementares sobre os tópicos abordados nesta unidade, o leitor interessado pode consultar as seguintes referências:

ARNOLD, K.; GOSLING, J.; HOLMES, D. *The Java Programming Language*. 4. ed. Boston: Addison-Wesley, 2005.

CAELUM. *Apostila do curso FJ-11* – Java e Orientação a Objetos. 2014. Disponível em: http://www.caelum.com.br/apostila-java-orientacao-objetos. Acesso em: 12 ago. 2014.

CAMARÃO, C.; FIGUEIREDO, L. *Programação de Computadores em Java*. 1. ed. São Paulo: LTC, 2003.

DEITEL, P.; DEITEL, H. *Java*: Como programar. 8. ed. São Paulo: Pearson Brasil, 2010.

GOLDMAN, A.; KON, F.; SILVA, P. J. *Introdução à Ciência da Computação com Java e Orientação a Objetos*. 1. ed. São Paulo: IME-USP, 2006. Disponível em: http://ccsl.ime.usp.br/files/books/intro-java-cc.pdf. Acesso em: 12 ago. 2014.

WAZLAWICK, R. S. *Análise e Projeto de Sistemas de Informação Orientados a Objetos*. 2. ed. Rio de Janeiro: Elsevier, 2011.

UNIDADE 4

Entrada e saída em Java

4.1 Primeiras palavras

Esta unidade tem como objetivo apresentar como os conceitos inerentes às operações de entrada e saída são definidos na linguagem de programação Java. Análogo aos demais pacotes presentes na linguagem Java, o pacote `java.io`, responsável pela definição dos conceitos inerentes às operações de entrada e saída, é orientado a objetos e utiliza os conceitos discutidos anteriormente: classes, classes abstratas, interfaces, polimorfismo, etc.

4.2 Problematizando o tema

Ao final desta unidade, espera-se que o leitor seja capaz de reconhecer e definir precisamente os conceitos inerentes às operações de entrada e saída presentes na linguagem Java. Dessa forma, esta unidade pretende discutir as seguintes questões:

- O que representa um *stream* na linguagem Java?
- Como arquivos são manipulados na linguagem Java?
- Como são realizados acessos sequenciais e não sequenciais a arquivos na linguagem Java?
- Quais são as semelhanças e as diferenças entre a manipulação de *streams* de dados e a manipulação de *streams* de caracteres?

4.3 Pacote `java.io`

Por entrada e saída, subentende-se o conjunto de mecanismos oferecidos para que um programa executando em um computador consiga respectivamente obter e fornecer informação de dispositivos externos ao ambiente de execução, composto de processador e memória principal.

Os dados de entrada podem ser provenientes de um arquivo em disco, de um teclado ou de uma conexão de rede. Da mesma maneira, o destino de saída pode ser um arquivo em disco ou uma conexão em rede. A linguagem Java permite lidar com todos os tipos de entrada e saída através de uma abstração conhecida como *stream*. Um *stream* é tanto uma fonte de dados como também um destino para dados. O pacote `java.io` define um grande número de classes para ler e escrever *streams*. As classes do pacote `java.io` oferecem as funcionalidades para manipular a entrada e

saída de bytes, adequadas para a transferência de dados binários, e para manipular a entrada e saída de caracteres, adequadas para a transferência de textos.

Assim como os demais pacotes da linguagem Java, a parte de controle de entrada e saída de dados é orientada a objetos e usa os conceitos discutidos nas unidades anteriores: interfaces, classes abstratas, polimorfismo, etc.

As classes abstratas **`InputStream`** e **`OutputStream`** definem, respectivamente, o comportamento padrão para a leitura e a escrita de *streams* de bytes. Ou seja, fontes de dados manipuladas como sequências de bytes são tratadas em Java pela classe **`InputStream`** e suas classes derivadas. Similarmente, destinos de dados, manipulados como sequências de bytes, são tratados pela classe **`OutputStream`** e suas classes derivadas.

Aplicações típicas de entrada e saída envolvem a manipulação de arquivos contendo caracteres. As classes abstratas **`Reader`** e **`Writer`** definem, respectivamente, o comportamento padrão para leitura e escrita de *streams* de caracteres.

As **`streams`** em Java são unidirecionais, ou seja, você pode ler de uma **`stream`** de entrada (ex. **`InputStream`**), mas não pode escrever nela. Analogamente, você pode escrever de uma *stream* de saída (ex. **`OutputStream`**), mas não pode ler dela.

4.4 Manipulação de arquivos

Um dispositivo de entrada e saída de vital importância é o disco (discos rígidos ou discos portáteis tais como um *pen drive*) manipulado pelo sistema operacional e por linguagens de programação através do conceito de arquivos. Um arquivo é uma abstração utilizada por sistemas operacionais para uniformizar a interação entre o ambiente de execução e os dispositivos externos a ele em um computador. Tipicamente, a interação de um programa com um dispositivo através de arquivos passa por três etapas: (1) abertura ou criação de um arquivo; (2) transferência de dados; e (3) fechamento do arquivo.

Em Java, a classe **`File`** permite representar arquivos nesse nível de abstração. Um dos construtores dessa classe recebe como argumento uma *string* que identifica, por exemplo, o nome de um arquivo em disco. Os métodos dessa classe permitem obter informações sobre o arquivo:

- O método `createNewFile()` cria um novo (vazio) arquivo se e somente se não existe um arquivo com mesmo nome;

Observe que o nome do novo arquivo é a *string* passada como argumento na construção da instância da classe **`File`**.

- O método **`exists()`** permite verificar se o arquivo especificado existe ou não;
- Os métodos **`canRead()`** e **`canWrite()`** verificam se o arquivo concede a permissão para leitura e escrita, respectivamente;
- O método **`length()`** retorna o tamanho do arquivo;
- O método **`lastModified()`** retorna há quanto tempo ocorreu a última modificação (em milissegundos desde 01 de janeiro de 1970);
- O método **`isFile()`** retorna **`true`** se o objeto representa um arquivo do disco. Analogamente, o método **`isDirectory()`** retorna **`true`** se o objeto representa um diretório do disco;
- Outros métodos permitem ainda realizar operações sobre o arquivo como um todo, tais como o método **`delete()`**, que remove o arquivo ou diretório representado pelo objeto (instância da classe **`File`**).

Para saber mais detalhes sobre os métodos da classe **`java.io.File`**, o leitor interessado pode consultar o link: http://docs.oracle.com/javase/7/docs/api/java/io/File.html.

Um diretório é representado por uma instância da classe **`File`** que contém uma lista de outros arquivos e diretórios. Conforme dito, é possível saber se um objeto **`File`** é um diretório através do método **`isDirectory()`**. Pode-se, então, usar o método **`list()`** para listar arquivos e diretórios contidos nesse diretório. O Código 4.1 apresenta um programa que imprime a lista de arquivos e diretórios de um diretório. O nome do diretório é passado pela linha de comando (argumento **`args`** do método **`main`**) para o programa a ser executado.

Observe que as funcionalidades de transferência sequencial de dados a partir de ou para um arquivo não são suportadas pela classe **`File`**, mas sim pelas classes **`FileInputStream`** (Seção 4.5.1), **`FileOutputStream`** (Seção 4.5.2), **`FileReader`** (Seção 4.6.1) e **`FileWriter`** (Seção 4.6.2). Todas essas classes oferecem pelo menos um construtor, que recebe como argumento um objeto da classe **`File`**, e implementam os métodos básicos de transferência de dados suportados respectivamente pelas classes abstratas **`InputStream`**, **`OutputStream`**, **`Reader`** e **`Writer`**.

```java
import java.io.File;
/**
 * Classe ListaDiretório
 *
 * @author Delano Medeiros Beder
 */
public class ListaDiretório {
    public static void main(String args[]) {

        String nomeDir = args[0];
        File f1 = new File(nomeDir);
        if (f1.isDirectory()) {
            System.out.println("Diretório " + nomeDir);
            String s[] = f1.list();
            for (int i = 0; i < s.length; i++) {
                File f = new File(nomeDir + "/" + s[i]);
                System.out.print(s[i]);
                if (f.isDirectory()) {
                    System.out.println(" <dir> ");
                } else {
                    System.out.println(" <file>");
                }
            }
        } else {
            System.out.println(nomeDir + " não é um diretório.");
        }
    }
}
```

Código 4.1 Lista de arquivos e diretórios de um diretório.

4.4.1 Acesso não sequencial

Para aplicações que necessitam manipular arquivos de forma não sequencial (ou "direta"), envolvendo avanços ou retrocessos arbitrários na posição do arquivo onde ocorrerá a transferência, a linguagem Java oferece a classe **`RandomAccessFile`**. Esta não é subclasse da classe **`File`**, mas oferece um construtor que aceita como argumento de especificação do arquivo um objeto dessa classe. Outro construtor recebe a especificação do arquivo na forma de uma *string*. Para ambos os construtores, um segundo argumento é uma *string* que especifica o modo de operação, "r" para leitura apenas ou "rw" para leitura e escrita. Os métodos para a manipulação da posição corrente do ponteiro no arquivo são:

- O método **`seek(long pos)`**, que seleciona a posição em relação ao início do arquivo para a próxima operação de leitura ou escrita;
- O método **`getFilePointer()`**, que retorna a posição atual do ponteiro do arquivo;
- Além disso, o método **`length()`** retorna o tamanho do arquivo em bytes.

Para a manipulação do conteúdo do arquivo, todos os métodos especificados pelas interfaces **`DataInput`** e **`DataOutput`** são implementados por essa classe. Assim, é

possível por exemplo usar os métodos **readInt()** e **writeInt()** para ler e escrever valores do tipo primitivo **int**[1]. Observe que os arquivos manipulados pela classe **RandomAcessFile** são arquivos binários – uma sequência de bytes, ou seja, uma sequência de oito dígitos (bits) agrupados.

Para maiores detalhes sobre a classe **RandomAcessFile**, o leitor interessado pode consultar o link: http://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html.

```java
import java.io.IOException;
import java.io.RandomAccessFile;

/**
 * Classe Testa RandomAcessFile
 *
 * @author Delano Medeiros Beder
 */
public class TesteRAF {

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

        // Cria um RandomAcessFile e posiciona na posição 0
        RandomAccessFile file = new RandomAccessFile("rand.txt", "rw");

        // Escrevendo no arquivo
        file.writeChar('V'); // um caractere
        file.writeInt(999);  // um inteiro
        file.writeDouble(99.99); // um double

        // Retorna à posição 0 e lê do arquivo na ordem em que foi escrito
        file.seek(0);
        System.out.println(file.readChar()); // imprime caractere 'V'
        System.out.println(file.readInt());  // imprime inteiro 999
        System.out.println(file.readDouble()); // imprime double 99.99

        // Posiciona na posição 2 e lê o segundo item escrito - o inteiro 999
        file.seek(2);
        System.out.println(file.readInt());

        // Posiciona no fim do arquivo e escreve o booleano true no arquivo
        file.seek(file.length());
        file.writeBoolean(true);

        // Posiciona na posição 4 e lê o quarto item escrito - o booleano true
        file.seek(4);
        System.out.println(file.readBoolean());

        // Fecha o arquivo
        file.close();
    }
}
```

Código 4.2 Acesso não sequencial a arquivos.

O Código 4.2 apresenta um programa que utiliza a classe **RandomAcessFile** para realizar algumas operações em um arquivo.

- Nas linhas 13 e 14, uma instância da classe **RandomAcessFile** é criada. Essa operação também posiciona o ponteiro de leitura/escrita no início do arquivo;

[1] Existem os métodos **read<Type>** e **write<Type>** para cada tipo primitivo Java: **int**, **char**, etc.

- Nas linhas 16 a 19, 3 itens (um caractere, um inteiro e um ponto flutuante) são escritos no arquivo;
- Nas linhas 21 a 25, o ponteiro de leitura/escrita é posicionado no início do arquivo (posição 0), e os dados são lidos na ordem em que foram escritos anteriormente;
- Nas linhas 27 a 29, o ponteiro de leitura/escrita é posicionado na posição 2 e ocorre a leitura do segundo item (inteiro 999), escrito anteriormente;
- Nas linhas 31 a 33, o ponteiro de leitura/escrita é posicionado no final do arquivo e ocorre a escrita de um valor booleano (`true`);
- Nas linhas 35 a 37, o ponteiro de leitura/escrita é posicionado na posição 4 e ocorre a leitura do quarto item (booleano `true`), escrito anteriormente;
- Por fim, na linha 40, ocorre a operação de fechamento do arquivo.

4.5 Leitura e escrita sequencial de bytes

Conforme dito, as classes abstratas `InputStream` e `OutputStream` definem, respectivamente, o comportamento padrão para a leitura e a escrita de *streams* de bytes. Ou seja, fontes de dados manipuladas como sequências de bytes são tratadas pela linguagem Java pela classe `InputStream` e suas classes derivadas. Similarmente, destinos de dados manipulados como sequências de bytes são tratados pela classe `OutputStream` e suas classes derivadas.

4.5.1 Leitura de *streams* de bytes

A classe abstrata `InputStream` oferece a funcionalidade básica para a leitura de um byte ou de uma sequência de bytes a partir de algum *stream* de entrada. A classe `InputStream` apenas discrimina as funcionalidades genéricas, tais como os métodos `available()`, que retorna o número de bytes disponíveis para leitura, e `read()`, que retorna o próximo byte do *stream* de entrada. Visto que a classe `InputStream` é uma classe abstrata, não é possível criar diretamente objetos dessa classe. É necessário criar objetos de uma de suas subclasses concretas para ter acesso às funcionalidades especificadas pela classe `InputStream`.

A classe `ByteArrayInputStream` permite ler valores originários de um *array* de bytes, permitindo, assim, tratar uma área de memória como um *stream* de entrada de dados.

Outra funcionalidade associada à transferência de dados está relacionada à conversão de formatos, tipicamente entre texto e o formato interno de dados binários. Essa e outras funcionalidades são suportadas através do oferecimento de filtros, classes derivadas da classe `FilterInputStream`, que podem ser agregados aos objetos que correspondem aos mecanismos elementares de entrada.

A classe `FileInputStream` lê bytes que são originários de um arquivo em disco. Usualmente, um objeto dessa classe é usado em combinação com filtros – subclasses de `FilterInputStream`.

A classe `BufferedInputStream`, subclasse de `FilterInputStream`, lê mais dados que aqueles solicitados no momento da invocação do método `read()` e os coloca em um *buffer* interno. Como a velocidade de operação de dispositivos de entrada e saída é várias ordens de grandeza mais lenta que a velocidade de processamento, *buffers* são tipicamente utilizados para melhorar a eficiência das operações de leitura e escrita.

A classe `DataInputStream`, subclasse de `FilterInputStream`, permite a leitura de representações binárias dos tipos primitivos de Java, oferecendo métodos como `readBoolean()`, `readChar()`, `readDouble()` e `readInt()`. É uma implementação da interface `DataInput`. Essa interface, também implementada pela classe `RandomAccessFile` (Seção 4.4.1), especifica métodos que possibilitam a interpretação de uma sequência de bytes como um tipo primitivo da linguagem Java. Adicionalmente, é também possível interpretar a sequência de bytes como um objeto da classe `String`. Em geral, métodos dessa interface geram a exceção `EOFException` se for solicitada a leitura além do final do arquivo. Em outras situações de erro de leitura, a exceção `IOException` é gerada.

A classe `PushbackInputStream`, subclasse de `FilterInputStream`, oferece o método `unread()`, que permite repor um ou mais bytes de volta ao *stream* de entrada, como se eles não tivessem sido lidos.

E por fim, a classe `SequenceInputStream` oferece a funcionalidade de concatenar duas ou mais instâncias de `InputStream`. O construtor especifica os objetos que serão concatenados e, automaticamente, quando o fim do primeiro objeto é alcançado, os bytes passam a ser obtidos do segundo objeto.

Para saber mais detalhes sobre a classe **InputStream** e suas derivadas, o leitor interessado pode consultar o seguinte link:

http://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html.

```java
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

/**
 * Exemplo de código que utiliza InputStream e suas subclasses para ler a partir de um arquivo
 * de entrada ou teclado. O teclado (System.in) é "mapeado" em java como um InputStream.
 * É importante salientar que leitura, nesse programa, é bufferizada. Dessa forma, é
 * possível ler vários bytes em um único acesso ao dispositivo de entrada.
 *
 * @author Delano Medeiros Beder
 */
public class TesteInputStream {

    public static void readStream (InputStream is) throws IOException{
        BufferedInputStream bis = new BufferedInputStream(is);
        int i = bis.read(); // bis.read() retorna -1 quando não há mais bytes - fim do arquivo
        while (i != -1) {
            System.out.print((char) i); // imprime o que foi lido
            i = bis.read(); // bis.read() retorna -1 quando não há mais bytes - fim do arquivo
        }
        System.out.println();
    }

    public static void main(String[] args) throws IOException {
        InputStream is;
        if (args.length == 0) {
            is = System.in; // Lê do teclado
        } else {
            is = new FileInputStream(args[0]); // Lê de um arquivo
        }
        readStream(is);
        is.close(); // fecha o stream de leitura
    }
}
```

Código 4.3 Leitura de um *stream* de bytes.

O Código 4.3 apresenta um programa que realiza operações de leitura em um *stream* de bytes. A grande vantagem da abstração definida pela classe abstrata **InputStream** pode ser apresentada no método **readStream()** (linhas 12-21) que utiliza um **InputStream** recebido como argumento para ler de um *stream* de entrada. De onde esse método está lendo? Não importa: para ler de um arquivo (**FileInputStream**) ou do teclado (**System.in**), que é uma instância da classe **InputStream**, pode-se utilizar o método **readStream()** visto que ele aceita qualquer subclasse concreta de **InputStream**. O nome do arquivo a ser lido é o argumento **args** passado pela linha de comando. Caso não seja passado nenhum argumento, a leitura é realizada através da digitação dos dados no teclado. A linguagem Java representa o teclado como o atributo **in** da classe **System** – uma instância da classe **InputStream**. É importante salientar que a leitura, nesse programa, é *bufferizada* (**BufferedInputStream**).

4.5.2 Escrita de *streams* de bytes

A classe abstrata **OutputStream** oferece a funcionalidade básica para a transferência sequencial de bytes para algum *stream* de saída. A classe **OutputStream** apenas discrimina as funcionalidades genéricas, tais como os métodos **flush()**, o qual força que todos os bytes do *buffer* sejam escritos no *stream* de saída, e **write()**, o qual escreve um byte ou uma sequência de bytes para o *stream* de saída. Visto que a classe **OutputStream** é uma classe abstrata, não é possível criar diretamente objetos dessa classe. É necessário criar objetos de uma de suas subclasses concretas para ter acesso às funcionalidades especificadas pela classe **OutputStream**.

A classe **ByteArrayOutputStream** oferece facilidades para escrever em um *array* de bytes, permitindo, assim, tratar uma área de memória como um *stream* de saída de dados. Os dados, posteriormente, podem ser acessados através dos métodos **toByteArray()** ou **toString()**.

A classe **FileOutputStream** oferece facilidades para escrever em arquivos em disco. Usualmente, um objeto dessa clase é usado em combinação com filtros – subclasses de **FilterOutputStream**.

A classe **FilterOutputStream** define funcionalidades básicas para a filtragem da saída de dados, implementadas em alguma de suas classes derivadas. A classe **BufferedOutputStream**, subclasse de **FilterOutputStream**, armazena bytes em um *buffer* interno até que este esteja cheio ou até que o método **flush()** seja invocado.

A classe **DataOutputStream**, subclasse de **FilterOutputStream**, é uma implementação da interface **DataOutput** que permite escrever valores de variáveis de tipos primitivos de Java em um formato binário portátil, oferecendo métodos tais como **writeBoolean()**, **writeChar()**, **writeDouble()** e **writeInt()**.

A classe **PrintStream**, subclasse de **FilterOutputStream**, oferece métodos para apresentar representações textuais dos valores de tipos primitivos Java, através das várias assinaturas dos métodos **print()** (impressão sem mudança de linha) e **println()** (impressão com mudança de linha). É importante salientar que a linguagem Java mapeia o console (monitor) como o atributo **out** da classe **System** – uma instância da classe **PrintStream**. Dessa forma, o comando **System.out.println()** consiste na invocação do método **println()** em uma instância da classe **PrintStream**.

E, por fim, as classes **PipedInputStream** e **PipedOutputStream** oferecem um mecanismo de comunicação de bytes entre *threads* de uma máquina virtual Java. A classe **PipedInputStream** oferece a funcionalidade de leitura de um *pipe* de bytes cuja origem está associada a um objeto **PipedOutputStream**. Já a classe **PipedOutputStream** implementa a origem de um *pipe* de bytes, que serão lidos a partir de um objeto **PipedInputStream**.

Para saber mais detalhes sobre a classe **OutputStream** e suas derivadas, o leitor interessado pode consultar o seguinte link:

http://docs.oracle.com/javase/7/docs/api/java/io/OutputStream.html.

O Código 4.4 apresenta um programa que realiza operações de leitura e escrita em *streams* de bytes. Esse programa realiza a cópia de um arquivo de entrada. Ou seja, um arquivo de entrada é lido (através do método **read()** – linhas 25 e 28), e seu conteúdo é escrito (através do método **write()** – linha 27) em um arquivo de saída. Os arquivos de entrada e saída são representados por instâncias das classes **FileInputStream** e **FileOutputStream**, respectivamente. É importante salientar que a leitura e a escrita, nesse programa, são *bufferizadas*. Dessa forma, é possível ler/escrever vários bytes em um único acesso ao dispositivo de entrada/saída.

```java
/**
 * Exemplo de código que utiliza InputStream (e suas subclasses) e OutputStream (e suas
 * subclasses) para realizar a cópia de um arquivo de entrada. Ou seja, um arquivo de entrada é
 * lido, e seu conteúdo é escrito em um arquivo de saída. É importante salientar que a leitura e
 * a escrita, nesse programa, são bufferizadas. Dessa forma, é possível ler/escrever vários bytes
 * em um único acesso ao dispositivo de entrada/saída.
 *
 * @author Delano Medeiros Beder
 */
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

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

        FileInputStream is = new FileInputStream("entrada.txt");
        BufferedInputStream bis = new BufferedInputStream(is);

        FileOutputStream os = new FileOutputStream("saida.txt");
        BufferedOutputStream bos = new BufferedOutputStream(os);

        int i = bis.read(); // bis.read() retorna -1 quando não há mais bytes - fim do arquivo
        while (i != -1) {
            bos.write((char) i); // escreve o que foi lido no stream de saída
            i = bis.read(); // bis.read() retorna -1 quando não há mais bytes - fim do arquivo
        }
        bis.close(); // fecha o stream de leitura
        bos.close(); // fecha o stream de escrita
    }
}
```

Código 4.4 Leitura e escrita de *streams* de bytes.

4.5.3 Classes `Scanner` e `PrintStream`

A versão 1.5 da linguagem Java trouxe a classe **`java.util.Scanner`**, que implementa operações de entrada de dados pelo teclado (ou arquivo). A classe **`Scanner`** possui vários métodos que possibilitam a entrada de dados de diferentes tipos, dos quais se destacam:

- **`String next()`**, que retorna uma *string* lida do *stream* de entrada;
- **`int nextInt()`**, que retorna um número inteiro, de 32 bits, lido do *stream* de entrada;

  Observação: a classe **`Scanner`** disponibiliza um método **`next<Type>`** para os principais tipos primitivos Java: **`float`**, **`double`**, etc.
- **`boolean hasNext()`**, que retorna **`true`** se ainda existe dado a ser lido do *stream* de entrada.

O Código 4.5 apresenta um programa que lê dois inteiros (linhas 17 e 18) digitados pelo teclado (**`System.in`**) e imprime a soma deles (linha 20). É importante mencionar que a linha 14 poderia ser substituída pela linha 15, que se encontra comentada no código. Nesse caso, os dois inteiros seriam lidos de um arquivo de entrada ("soma.txt") ao invés de serem lidos pelo teclado.

```java
1  import java.io.FileInputStream;
2  import java.io.FileNotFoundException;
3  import java.util.Scanner;
4
5  /**
6   * Exemplo de programa que lê dois inteiros x e y (digitados pelo teclado) e imprime a soma deles.
7   *
8   * @author Delano Medeiros Beder
9   */
10 public class EntradaSoma {
11
12     public static void main(String[] args) {
13
14         Scanner sc = new Scanner(System.in);
15         // Scanner sc = new Scanner(new FileInputStream("soma.txt"));
16
17         int x = sc.nextInt();
18         int y = sc.nextInt();
19
20         System.out.println(x + " + " + y + " = " + (x + y));
21     }
22 }
```

Código 4.5 Classe **`Scanner`** – leitura pelo teclado (ou arquivo).

A versão 1.5 da linguagem Java também trouxe a classe **`java.io.PrintStream`**, discutida na Seção 4.5.2, a qual possui um construtor que recebe o nome de um

arquivo como argumento. Dessa forma, a leitura do teclado com saída para um arquivo ficou muito simples.

O Código 4.6 apresenta um programa que utiliza as classes **Scanner** e **PrintStream** para ler a partir do teclado e escrever o que foi lido em um arquivo de saída. Ou seja, todas as linhas lidas (método **nextLine()** – linha 19) serão escritas (método **println()** – linha 21) em um arquivo de saída. É importante mencionar que a linha 16 poderia ser substituída pela linha 17, que se encontra comentada no código. Nesse caso, o *stream* de saída seria o console (monitor) em substituição a um arquivo de saída.

```java
import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.util.Scanner;

/**
 * Exemplo de código que utiliza Scanner e PrintStream para ler a partir do
 * teclado e escrever em um arquivo (ou no console).
 *
 * @author Delano Medeiros Beder
 */
public class EntradaSaida {

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

        Scanner sc = new Scanner(System.in);
        PrintStream ps = new PrintStream("saida.txt");
        // PrintStream ps = System.out;

        String s = sc.nextLine();
        while (!s.equals("sai")) {
            ps.println(s);
            s = sc.nextLine();
        }
    }
}
```

Código 4.6 Classe **PrintStream** – escrita para arquivo (ou console).

4.5.4 Serialização de objetos

Sendo Java uma linguagem de programação orientada a objetos, seria de se esperar que, além das funcionalidades usuais de entrada e saída, ela oferecesse também alguma funcionalidade para transferência de objetos. O mecanismo de serialização suporta essa funcionalidade. Ou seja, é possível converter um objeto em um *stream* de bytes que depois podem ser convertidos novamente em um objeto.

Através desse mecanismo de serialização, é possível gravar e recuperar objetos de forma transparente. A classe **ObjectOutputStream**, subclasse de **OutputStream**, oferece o método **writeObject()**, que serializa um objeto. Ou seja, converte a representação de um objeto em memória em uma sequência de bytes. E essa sequência

de bytes pode ser enviada em uma conexão de rede (classe `Socket`) ou armazenada em um arquivo de disco. A classe `ObjectInputStream`, subclasse de `InputStream`, oferece o método `readObject()` para a leitura de objetos que foram serializados.

Classes para as quais são previstas serializações e desserializações devem implementar a interface `java.io.Serializable`. Essa interface não especifica nenhum método ou campo – é um exemplo de uma interface *marker*, servindo apenas para registrar a semântica de serialização. É importante lembrar que apenas os atributos são armazenados, os métodos não. O objeto serializável pode conter atributos dos tipos primitivos (`boolean`, `int`, `char`, etc.) como também referência a objetos. Porém, os objetos referenciados também devem implementar a interface `Serializable`.

Além disso, se é desejável que um atributo não seja armazenado, é preciso apenas colocar o modificador `transient` na declaração do atributo. Isso é bastante útil quando o objeto que se deseja armazenar contém uma referência para outro objeto que não é serializável. Os atributos estáticos também não são armazenados.

Serialização é um processo transitivo, ou seja, subclasses serializáveis de classes serializáveis são automaticamente incorporadas à representação serializada do objeto raiz. Para que o processo de desserialização possa operar corretamente, todas as classes envolvidas no processo devem ter um construtor *default* (sem argumentos).

O Código 4.7 apresenta um programa que serializa e desserializa instâncias da classe Conta (Figura 1.2).

O método `gravaConta()` (linhas 15-20) é responsável pela serialização, armazenando seu conteúdo em um arquivo em disco, de objetos da classe `Conta`. A sequência de passos para realizar essa tarefa é: (1) associar a instância de `ObjectOutputStream` ao *stream* de saída (linhas 16 e 17) – no caso desse método, o *stream de saída* é um arquivo em disco (`FileOutputStream`); (2) serializar o objeto, armazenando no *stream* de saída, através do método `writeObject()` (linha 18); e (3) fechar o *stream* de saída (linha 19).

O método `carregaConta()` (linhas 22-28) é responsável pela desserialização de objetos da classe `Conta`. A sequência de passos para realizar essa tarefa é: (1) associar a instância de `ObjectInputStream` ao *stream* de entrada (linhas 23 e 24) – no caso desse método, o *stream* de entrada é um arquivo em disco (`FileInputStream`); (2) ler a representação do objeto, a partir do *stream* de entrada, através do método `readObject()` (linha 25); e (3) fechar o *stream* de entrada (linha 26).

```java
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

/**
 * Essa classe Serializa e desserializa instâncias da classe Conta.
 *
 * @author Delano Medeiros Beder
 */
public class SerializadorConta {

    public static void gravaConta(Conta conta, File file) throws IOException {
        ObjectOutputStream saida = new ObjectOutputStream(
                new FileOutputStream(file));
        saida.writeObject(conta);
        saida.close();
    }

    public static Conta carregaConta(File file) throws IOException, ClassNotFoundException{
        ObjectInputStream entrada = new ObjectInputStream(
                new FileInputStream(file));
        Conta conta = (Conta) entrada.readObject();
        entrada.close();
        return conta;
    }

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

       Conta conta;
       File file = new File("Conta.dat");

       if (!file.exists()) {
           conta = new Conta(0);
       } else {
           conta = carregaConta(file);
           conta.deposito(10);
       }
       System.out.println(conta.getSaldo());
       gravaConta(conta, file);
    }
}
```

Código 4.7 Serialização e desserialização de objetos.

Conforme discutido, o método **main** (linhas 30-43) é o ponto de entrada desse programa. Esse método verifica se o arquivo denominado "Conta.dat" existe (linhas 33-35). Se o arquivo não existe, uma nova instância da classe **Conta** é criada com saldo R$ 0,00 (linha 36).

Caso o arquivo exista, esse arquivo é passado como argumento para o método **CarregaConta()** discutido anteriormente (linha 38). Após o objeto ser desserializado, é realizada uma operação de depósito nessa conta (linha 39).

Nos dois casos (arquivo existe ou não), o saldo da conta é impresso, e, logo após, o objeto é serializado através da invocação do método **salvaConta()** (linhas 41-42).

Fica como exercício para o leitor – execute 5 vezes esse programa e responda as seguintes perguntas: quais valores são impressos? Por quê?

4.6 Leitura e escrita sequencial de caracteres

Conforme dito, as classes abstratas **Reader** e **Writer** definem, respectivamente, o comportamento padrão para a leitura e a escrita de *streams* de caracteres. Ou seja, fontes de dados manipuladas como sequências de caracteres são tratadas pela linguagem Java pela classe **Reader** e suas classes derivadas. Similarmente, destinos de dados manipulados como sequências de caracteres são tratados pela classe **Writer** e suas classes derivadas.

4.6.1 Leitura de *streams* de caracteres

A classe **Reader** oferece as funcionalidades para obter caracteres de alguma fonte de dados – determinar qual delas vai depender de qual classe concreta, derivada de **Reader**, está sendo utilizada. A classe **Reader** apenas discrimina funcionalidades genéricas, como o método **read()**, que retorna um **int**[2]representando o caractere lido, e **ready()**, que retorna um **boolean** indicando, se verdadeiro, que o dispositivo está pronto para disponibilizar o próximo caractere.

Como **Reader** é uma classe abstrata, não é possível criar diretamente objetos dessa classe. É preciso criar objetos de uma de suas subclasses concretas para ter acesso à funcionalidade especificada por **Reader**.

A classe **BufferedReader** incorpora um *buffer* a um objeto **Reader** com o objetivo de melhorar a eficiência das operações de leitura. Adicionalmente, na leitura de texto, a classe **BufferedReader** adiciona um método **readLine()** para ler uma linha completa. A classe **LineNumberReader** estende a classe **BufferedReader** para adicionalmente manter um registro do número de linhas processadas.

As classes **CharArrayReader** e **StringReader** permitem fazer a leitura de caracteres a partir de *arrays* e de *strings* (instâncias da classe **String**), respectivamente. Assim, é possível utilizar a memória principal como uma fonte de caracteres.

A classe **FilterReader** é uma classe abstrata para representar classes que implementam algum tipo de filtragem sobre o dado lido. Sua classe derivada, **PushbackReader**, incorpora a possibilidade de retornar um caráter lido de volta à sua fonte.

[2] Codificação unicode do caractere lido – faixa de 0 a 65535 (‘\u0000’ a ‘\uffff’).

A classe **InputStreamReader** implementa a capacidade de leitura a partir de uma fonte que fornece bytes, traduzindo-os adequadamente para caracteres. Sua classe derivada, **FileReader**, permite associar essa fonte de dados a um arquivo.

A classe **PipedReader** faz a leitura a partir de instâncias da classe **PipedWriter**, estabelecendo um mecanismo de comunicação interprocessos – no caso de Java, entre *threads* de uma mesma máquina virtual.

Para saber mais detalhes sobre a classe **Reader** e suas derivadas, o leitor interessado pode consultar o link: http://docs.oracle.com/javase/7/docs/api/java/io/Reader.html.

O Código 4.8 apresenta um programa que realiza operações de leitura em um *stream* de caracteres. Esse programa lê e imprime o conteúdo de um arquivo. Ou seja, um arquivo de leitura é lido (método **readLine()** – linhas 21 e 24) através da associação das instâncias das classe **FileInputStream** e **InputStreamReader** (linhas 17-18), e seu conteúdo é impresso (linha 23). Ao final da leitura do arquivo, há a impressão da quantidade de linhas lidas (linha 28). É importante salientar que a leitura é bufferizada, uma vez que a classe **LineNumberReader** (linha 19) é subclasse de **BufferedReader**. Dessa forma, é possível ler vários caracteres (várias *strings*) em um único acesso ao disco.

```
import java.io.FileInputStream;                                                              1
import java.io.IOException;                                                                  2
import java.io.InputStreamReader;                                                            3
import java.io.LineNumberReader;                                                             4
                                                                                             5
/**                                                                                          6
 * Exemplo de código que utiliza FileInputStream + InputStreamReader + BufferedReader para   7
 * ler e imprimir o conteúdo de um arquivo. É importante salientar que a leitura é bufferizada. 8
 * Dessa forma, é possível ler vários caracteres (String) em um único acesso ao disco.       9
 *                                                                                           10
 * @author Delano Medeiros Beder                                                             11
 */                                                                                          12
public class TesteReader {                                                                   13
                                                                                             14
      public static void main(String[] args) throws IOException {                            15
                                                                                             16
           InputStream is = new FileInputStream("entrada.txt");                              17
           InputStreamReader isr = new InputStreamReader(is);                                18
           LineNumberReader br = new LineNumberReader(isr);                                  19
                                                                                             20
           String s = br.readLine();                                                         21
           while (s != null) {                                                               22
               System.out.println(s);                                                        23
               s = br.readLine();                                                            24
           }                                                                                 25
                                                                                             26
       System.out.println();                                                                 27
       System.out.println("Foram lidas " + br.getLineNumber() + " linhas.");                 28
    }                                                                                        29
}                                                                                            30
```

Código 4.8 Leitura de um *stream* de caracteres.

4.6.2 Escrita de *streams* de caracteres

As funcionalidades de escrita de texto estão associadas à classe abstrata `Writer` e suas classes derivadas. Entre as funcionalidades genéricas definidas pela classe `Writer` estão o método `write(int)`, em que o argumento representa um caractere, sua variante `write(String)` e o método flush(), que força a saída de dados pendentes para o dispositivo sendo acessado para escrita. Assim como para a classe `Reader`, as funcionalidades da classe `Writer` são implementadas através de suas subclasses concretas.

A classe `BufferedWriter` incorpora um *buffer* a um objeto `Writer`, permitindo uma melhoria de eficiência de escrita ao combinar várias solicitações de escrita de pequenos blocos em uma solicitação de escrita de um bloco maior.

As classes `CharArrayWriter` e `StringWriter` permitem fazer a escrita em *arrays* de caracteres e em instâncias da classe `StringBuffer`, respectivamente.

A classe `FilterWriter` é uma classe abstrata para representar classes que implementam algum tipo de filtragem sobre o dado escrito. A classe `OutputStreamWriter` implementa a capacidade de escrita, tendo como destino uma sequência de bytes, traduzindo-os adequadamente a partir dos caracteres de entrada. Sua classe derivada, `FileWriter`, permite associar esse destino de dados a um arquivo.

Para saber mais detalhes sobre a classe `Writer` e suas derivadas, o leitor interessado pode consultar o link: http://docs.oracle.com/javase/7/docs/api/java/io/Writer.html.

O Código 4.9 apresenta um programa que realiza operações de leitura e escrita em *streams* de caracteres. Esse programa realiza a cópia de um arquivo de entrada. Ou seja, um arquivo de entrada é lido (através do método `readLine()`), e seu conteúdo é escrito (através do método `write()`) em um arquivo de saída. Os arquivos de entrada e saída são representados por instâncias das classes `FileReader` e `FileWriter`, respectivamente. É importante salientar que a leitura e a escrita, nesse programa, são *bufferizadas*. Dessa forma, é possível ler/escrever vários bytes em um único acesso ao dispositivo de entrada/saída.

```java
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

/**
 * Exemplo de código que utiliza Reader (e subclasses) e Writer (e subclasses) para realizar a cópia
 * de um arquivo de entrada. Ou seja, um arquivo de entrada é lido e seu conteúdo é escrito em um
 * arquivo de saída. É importante salientar que a leitura/escrita são bufferizadas.
 *
 * @author Delano Medeiros Beder
 */
public class CopiaCaracteres {

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

        FileReader fr = new FileReader("entrada.txt");
        BufferedReader br = new BufferedReader(fr);
        FileWriter fw = new FileWriter("saida.txt");
        BufferedWriter bw = new BufferedWriter(fw);

        String s = br.readLine();
        while (s != null) {
            bw.write(s);
            bw.write("\n");
            s = br.readLine();
        }
        br.close();
        bw.close();
    }
}
```

Código 4.9 Leitura e escrita de *streams* de caracteres.

4.7 Considerações finais

Esta unidade discutiu como conceitos inerentes às operações de entrada e saída são definidos na linguagem de programação Java. A próxima unidade apresenta o *framework* de coleções presente na linguagem de programação Java.

4.8 Estudos complementares

Para estudos complementares sobre os tópicos abordados nesta unidade, o leitor interessado pode consultar as seguintes referências:

ARNOLD, K.; GOSLING, J.; HOLMES, D. *The Java Programming Language*. 4. ed. Boston: Addison-Wesley, 2005.

CAELUM. *Apostila do curso FJ-11* – Java e Orientação a Objetos. 2014. Disponível em: http://www.caelum.com.br/apostila-java-orientacao-objetos. Acesso em: 12 ago. 2014.

DEITEL, P.; DEITEL, H. *Java*: Como programar. 8. ed. São Paulo: Pearson Brasil, 2010.

UNIDADE 5

Coleções em Java

5.1 Primeiras palavras

Esta unidade tem como objetivo apresentar o *framework* de coleções presente na linguagem de programação Java. O *framework* de coleções Java é orientado a objetos e utiliza os conceitos discutidos anteriormente: classes, classes abstratas, interfaces, polimorfismo, etc.

5.2 Problematizando o tema

Ao final desta unidade, espera-se que o leitor seja capaz de reconhecer e definir precisamente os conceitos inerentes ao *framework* de coleções presente na linguagem Java. Dessa forma, esta unidade pretende discutir as seguintes questões:

- O que representa uma coleção na linguagem Java?
- Quais as semelhanças e diferenças entre as seguintes coleções Java: listas, conjuntos, filas, *deques* e mapas ?
- Como são implementados as operações de inserção, remoção e buscas de elementos na coleções Java?
- O que é necessário para iterar e/ou ordenar as coleções Java ?

5.3 *Framework* de coleções

Como discutido na Unidade 3, a manipulação de *arrays* é uma tarefa bastante árdua devido às características inerentes ao conceito de *arrays* (CAELUM, 2014a):

- Não é possível redimensionar um *array* em Java;
- Não é possível buscar diretamente um determinado elemento cujo índice não é conhecido;
- Não é possível saber quantas posições do *array* já foram populadas sem criar métodos e atributos auxiliares.

Além das dificuldades que os *arrays* apresentam, faltava na linguagem Java um conjunto robusto de classes e interfaces para suprir a necessidade de estruturas de

dados básicas, tais como listas ligadas e tabelas de espalhamento[1]. Com esses objetivos em mente, a linguagem Java incorporou, no pacote **java.util**, um conjunto de classes e interfaces conhecido como *framework* de coleções.

Esse *framework* foi completamente redefinido a partir da versão 1.2 da plataforma Java. As classes até então existentes para a representação de estruturas de dados – **Vector**, **Stack** e **Hashtable** – foram totalmente reprojetadas para se adequar ao novo *framework*. Elas foram mantidas por motivos de compatibilidade, embora a recomendação seja utilizar as novas classes de coleções.

O *framework* de coleções é uma arquitetura unificada para representar e manipular coleções. Uma coleção é simplesmente um objeto que agrupa vários elementos em uma única unidade. Além do *framework* de coleções Java, outro exemplo bastante conhecido é o *C++ Standard Template Library* (JOSUTTIS, 2012). O *framework* de coleções Java contém os seguintes elementos:

- **Interfaces**: estes são os tipos de dados abstratos que representam coleções. Interfaces permitem que coleções sejam manipuladas independentemente dos detalhes de sua implementação.
- **Classes**: estas são as implementações concretas das interfaces presentes no *framework* de coleções. Basicamente, essas classes são estruturas de dados reutilizáveis.
- **Algoritmos**: estes são os métodos que executam computações úteis, tais como pesquisa e ordenação, em objetos que implementam as interfaces presentes no *framework* de coleções. Os algoritmos são considerados polimórficos, isto é, o mesmo método pode ser usado em muitas implementações da interface. Basicamente, esses algoritmos são funcionalidades reutilizáveis. Esses algoritmos serão discutidos na Seção 5.9.

[1] Em inglês: *Hash Tables*.

5.3.1 Principais interfaces do *framework* de coleções

As principais interfaces do *framework* de coleções são apresentadas na Figura 5.1. Essas interfaces permitem que coleções sejam manipuladas independentemente dos detalhes da sua representação. Como pode ser observado, as principais interfaces do *framework* de coleções formam uma hierarquia. Note que a hierarquia consiste de duas árvores distintas – diferentemente das demais interfaces presentes na figura, a interface `Map` não herda da interface `Collection`.

A sintaxe `<T>` diz que as interfaces são genéricas (Seção 3.9). Ao declarar uma instância de coleção, é recomendado especificar o tipo do elemento contido na coleção. A especificação do tipo do elemento permite que o compilador verifique (em tempo de compilação) a corretude do tipo do elemento a ser inserido na coleção, reduzindo, assim, os erros em tempo de execução. Ou seja, se a coleção contiver *Strings*, **T** é `String`; se contiver inteiros, **T** é `Integer`, e assim por diante.

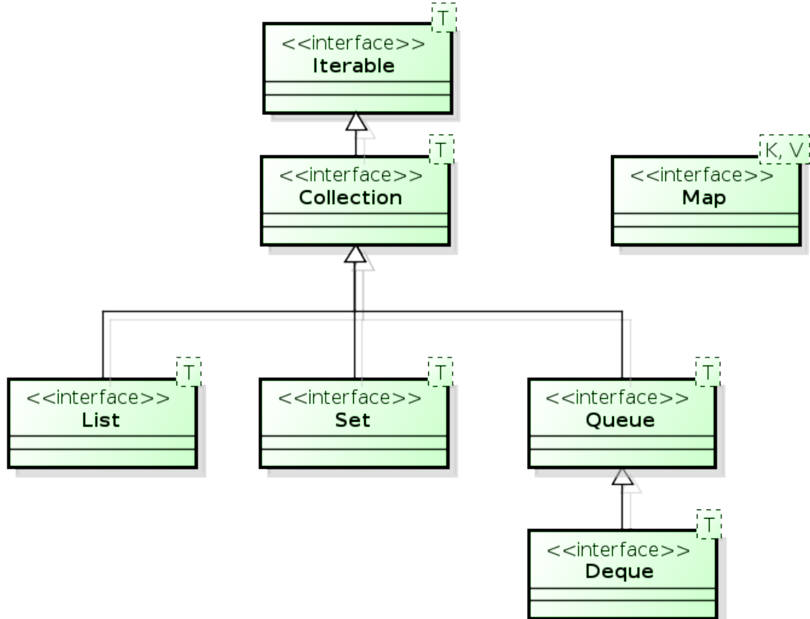


Figura 5.1 Principais interfaces do *framework* de coleções Java.

- `Iterable` – interface que determina que uma coleção pode ter seus elementos alcançados por uma estrutura **enhanced-for** (Seção 3.3.1). A interface `Iterable`, descrita na Seção 5.4.2, possui apenas um método:
  - `Iterator<T> iterator()` – que retorna um `Iterator` para uma coleção do tipo **T**.

- `Collection` – raiz da hierarquia de coleções. A interface `Collection` é o denominador comum que todas as coleções implementam. Alguns tipos de coleção permitem elementos duplicados, e outros não. Alguns tipos de coleção são ordenados, e outros não. A plataforma Java não fornece implementações diretas da interface `Collection`, porém fornece implementações de subinterfaces mais específicas, tais como `List`, `Set` e `Queue`. Entre os principais métodos dessa interface, podem-se citar:

  - **`boolean add(T element)`** – adiciona o elemento especificado à coleção;
  - **`boolean contains(Object element)`** – verifica se a coleção contém o elemento especificado;
  - **`boolean remove(Object element)`** – remove o elemento especificado da coleção;
  - **`int size()`** – retorna a quantidade de elementos presentes na coleção.

- `List` – representa uma coleção ordenada (geralmente chamada de sequência). As listas podem conter elementos duplicados. O usuário de uma lista geralmente tem um controle preciso sobre onde cada elemento é inserido na lista e pode acessar os elementos através de sua posição (índice inteiro). Listas serão discutidas na Seção 5.4.

- `Set` – representa uma coleção que não pode conter elementos duplicados. Essa interface modela e representa a abstração de conjuntos matemáticos. Conjuntos serão discutidos na Seção 5.6.

- `Queue` – representa uma coleção ordenada de objetos, assim como uma lista, porém a semântica de utilização é ligeiramente diferente. Uma fila geralmente é projetada para ter os elementos inseridos no fim da fila e os elementos removidos a partir do início da fila (FIFO – *first-in, first-out*). Exceções a esse comportamento são as filas de prioridades, em que os elementos são inseridos de acordo com um comparador fornecido ou ordem natural dos elementos. Filas serão discutidas na Seção 5.7.

- `Deque` – representa um tipo especial de fila, em que as inserções e remoções de elementos podem ser realizadas em ambas as extremidades da fila. *Deques* podem ser usadas tanto como FIFO (*first-in* , *first-out*) ou LIFO (*last-in, first-out*). *Deque* é a abreviação de *“Double Ended Queue”*. *Deques* serão discutidas na Seção 5.7.

- `Map` – representa conjuntos de pares de objetos, sendo um chamado **chave**, e o outro, **valor**. Mapas permitem valores iguais, porém não permitem chaves repetidas. É importante lembrar que chaves diferentes podem ser ou estar associadas a valores iguais. Na linguagem Java, a interface `Map` não herda da interface `Collection`, mas mesmo assim é possível acessar e manipular chaves e valores de mapas como se estes fossem coleções. Mapas serão discutidos na Seção 5.8.

5.4 Listas

A Figura 5.2 apresenta a interface `List` e suas principais implementações. Uma lista, representada pela interface `List`, é uma coleção que permite elementos duplicados e mantém uma ordenação específica entre os elementos (CAELUM, 2014a). Ou seja, é garantido que, quando se percorre a lista, os elementos serão encontrados em uma ordem predeterminada, definida no momento da inserção dos elementos. Além disso, listas são indexadas – é possível acessar os elementos através de suas posições (índices inteiros) na lista. Entre os principais métodos definidos na interface `List`, podem-se citar:

- **`T get(int index)`** – retorna o elemento armazenado na posição especificada como argumento;
- **`int indexOf(Object o)`** – retorna a posição da primeira ocorrência do objeto especificado como argumento;
- **`List<T> subList(int fromIndex, int toIndex)`** – retorna uma sublista contendo os elementos compreendidos entre as duas posições especificadas, incluindo o elemento da primeira posição, mas não o da segunda.

Como se pode observar pela Figura 5.1, a interface `List` é derivada da interface `Collection`. Portanto, os métodos definidos na interface `Collection`, discutidos na Seção 5.3.1, também podem ser invocados em listas – instâncias das classes concretas que implementam a interface `List`.

A classe abstrata `AbstractList` (Figura 5.2) implementa parcialmente a interface `List`. Ou seja, essa classe fornece uma implementação parcial dos métodos definidos nessa interface. No entanto, essa implementação parcial facilita a definição das implementações customizadas de listas – subclasses concretas de `AbstractList`: `ArrayList`, `LinkedList` e `Vector`.

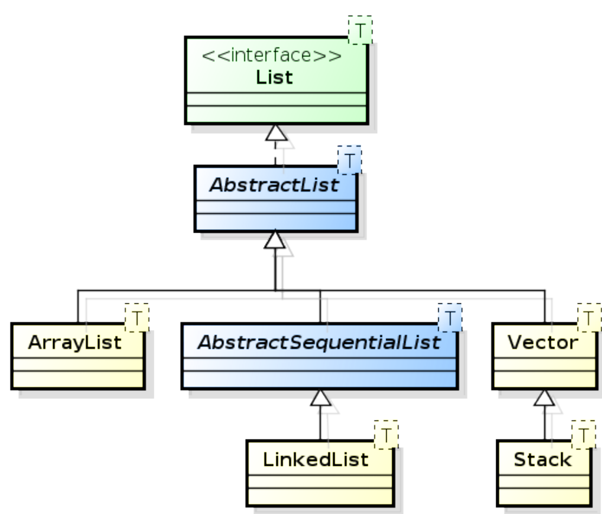


Figura 5.2 Listas – hierarquia de classes e interfaces.

A classe `ArrayList` é a mais utilizada e oferece uma implementação básica da interface `List`. Ela define um atributo *array* com o objetivo de armazenar os elementos. Porém, esse *array* interno é encapsulado, e o programador não tem como acessá-lo.

A classe abstrata `AbstractSequentialList` fornece uma implementação parcial de uma lista dinâmica de acesso sequencial. A classe `LinkedList`, subclasse de `AbstractSequentialList`, oferece uma implementação otimizada para a manipulação de listas dinâmicas (duplamente encadeadas), introduzindo os seguintes métodos:

- `void addFirst(T element)` e `void addLast(T element)` – inserem o elemento na cabeça ou na cauda da lista, respectivamente;
- `T getFirst()` e `T getLast()` – retornam o elemento presente na cabeça ou na cauda da lista, respectivamente;
- `boolean removeFirst()` e `removeLast()` – removem o elemento presente na cabeça ou na cauda da lista, respectivamente.

Comparando as duas classes concretas, pode-se dizer que a classe `ArrayList` é mais rápida na pesquisa do que a classe `LinkedList`, que é mais rápida na inserção e remoção nas extremidades (cabeça e cauda) da lista.

Outra implementação da interface `List` é a classe `Vector`, presente desde a versão 1.0 da plataforma Java e que foi adaptada para o uso com o *framework* de coleções Java, com a inclusão de novos métodos. Ao contrário das classes `ArrayList`

e **LinkedList**, a classe **Vector** é **synchronized**. Se a aplicação não necessita ter o controle sobre as *threads* (a maioria dos casos), é recomendada a utilização da **ArrayList** ao invés do **Vector**.

A classe **Stack** representa uma pilha LIFO (*last-in, first-out*) de objetos. Ou seja, a classe **Stack** é subclasse da classe **Vector** que define quatro métodos os quais permitem que um vetor seja tratado como uma pilha:

- **boolean empty()** – verifica se a pilha está vazia;
- **T peek()** – retorna o elemento no topo da pilha sem removê-lo;
- **T pop()** – remove e retorna o elemento presente no topo da pilha;
- **T push(T item)** – insere o elemento no topo da pilha.

5.4.1 Relacionamento *para muitos* entre classes

Conforme mencionado na Seção 3.5, coleções Java podem ser utilizadas na implementação de relacionamentos *para muitos*. Essa seção apresenta uma implementação mais robusta e mais elegante da associação *para muitos* entre as classes **Pessoa** e **Carro** (Figura 1.10). O Código 5.1 apresenta a implementação da classe **Carro**, que possui 4 atributos: **marca**, **cor**, **ano** e **dono**. O atributo **dono** representa a associação *para um* entre as classes **Carro** e **Pessoa**.

```java
/**
 * Classe Carro
 *
 * @author Delano Medeiros Beder
 */
public class Carro {
  /* Declaração dos atributos da classe */

    private String marca, cor;
    private int ano;
    private Pessoa dono;

  /* Construtor e demais métodos omitidos */

    public String toString() {
        return marca + " " + cor + "(" + ano + ")";
    }
}
```

Código 5.1 Classe **Carro**.

Apenas o método **toString()** é apresentado devido a sua relevância para o exemplo discutido nesta seção. Relembrando, o método **toString()** permite converter uma representação interna de um objeto em uma *string*. Esse método é invocado,

por exemplo, quando uma instância da classe **Carro** é passada como argumento para **System.out.println()** – ver Código 5.2, método **imprimeCarros()**.

```java
/**
 * Classe Pessoa
 *
 * @author Delano Medeiros Beder
 */
public class Pessoa {

    /* Declaração dos atributos da classe */

    private String CPF, nome;
    private List<Carro> carros;

    /* Declaração do Construtor */

    public Pessoa(String CPF, String nome) {
        this.CPF = CPF;
        this.nome = nome;
        this.carros = new ArrayList<Carro>();
    }

    /* Demais métodos omitidos */

    public void adicionaCarro(Carro c) {
        carros.add(c);
    }

    public void removeCarro(Carro c) {
        carros.remove(c);
    }

    public void imprimeCarros() {
        for (Carro c: carros) {
            System.out.println(c);
        }
    }

    public static void main(String[] args) {
        Pessoa josé = new Pessoa("0123456789", "José da Silva");
        Carro ferrari = new Carro("Ferrari", "Vermelho", 2012);
        Carro audi = new Carro("Audi", "Branco", 2011);
        Carro porsche = new Carro("Porche", "Amarelo", 2013);

        josé.adicionaCarro(ferrari);
        josé.adicionaCarro(audi);
        josé.adicionaCarro(porsche);

        josé.imprimeCarros();
    }
}
```

Código 5.2 Classe **Pessoa**.

O Código 5.2 apresenta a implementação da classe **Pessoa**, que define três atributos: **CPF**, **nome** e **carros**. O atributo **carros** é uma lista parametrizada pelo tipo **Carro**. Dessa forma, nessa lista podem ser inseridas apenas instâncias da classe **Carro**. Análogo ao Código 5.1, apenas os métodos relevantes foram apresentados.

Conforme se pode observar pelo construtor da classe, a classe `ArrayList` foi utilizada como a implementação da interface `List`. Foram definidos dois métodos, **`adicionaCarro()`** e **`removeCarro()`**, que são responsáveis pela inserção e remoção de instâncias da classe **`Carro`** da lista de carros (atributo **`carros`**).

O método **`imprimeCarros()`** é responsável pela impressão das informações relacionadas a cada carro presente na lista de carros. Note a utilização da estrutura **enhanced-for** (Seção 3.3.1) nesse método.

Finalmente, essa classe possui o método **`main()`**, que é o ponto de entrada desse programa. Conforme se pode observar nesse método, são criadas quatro instâncias de objetos – uma instância da classe **`Pessoa`** (**josé**) e três instâncias da classe **`Carro`** (**ferrari**, **audi** e **porsche**). Logo após, as três instâncias da classe **`Carro`** são adicionadas, pela invocação do método **`adicionaCarro()`**, à lista de carros que o objeto **josé** possui. E, por fim, a lista de carros do objeto **josé** é impressa – método **`imprimeCarros()`**.

A Figura 5.3 apresenta a saída impressa da execução do método **`main()`** da classe **`Pessoa`**. Note que os carros são impressos na ordem em que foram adicionados.

A saída será:
Ferrari Vermelho (2012)
Audi Branco (2011)
Porche Amarelo (2013)

Figura 5.3 Execução da classe `Pessoa`.

5.4.2 Iterando sobre coleções com `java.util.Iterator`

Antes de a versão 1.5 da plataforma Java introduzir a estrutura **enhanced-for**, iterações em coleções eram feitas com a classe **`Iterator`**. Toda coleção fornece acesso a um *iterator*, um objeto que implementa a interface **`Iterator`**, que conhece internamente a coleção e dá acesso a todos os seus elementos. Dessa forma, o método **`imprimeCarros()`** da classe **`Pessoa`** também poderia ser implementado conforme apresentado abaixo.

```
public void imprimeCarros() {
    Iterator<Carro> it = carros.iterator();
    while (it.hasNext()) {
        System.out.println(it.next());
    }
}
```

5.5 Ordenação

Conforme discutido anteriormente, as listas são percorridas de maneira predeterminada de acordo com a inclusão dos elementos na lista. Porém, em muitas situações, é desejável percorrer a lista de maneira ordenada. Por exemplo, o método `imprimeCarros()`, apresentado no Código 5.2, poderia imprimir os carros em ordem alfabética do atributo `marca`.

A classe `Collections` traz um método estático `sort()`, que ordena, em ordem crescente, uma lista (instância de `List`) passada como argumento. Dessa forma, é possível passar uma lista de carros (`List<Carro>`) para esse método. No entanto, para que o método `sort()` funcione apropriadamente, é necessário definir um **critério de ordenação** – uma forma de determinar qual elemento vem antes de qual. Ou seja, é necessário instruir o método `sort()` sobre como comparar as instâncias da classe `Carro` com o objetivo de determinar uma ordem na lista.

Para isso, o método `sort()` necessita que todos os elementos sejam **comparáveis** – ou seja, possuam um método que realiza a comparação entre duas instâncias da classe `Carro`. Logo, é necessário que todos os elementos da lista implementem a interface `java.lang.Comparable`, que define o método `compareTo()`. Esse método deve retornar **zero** se o objeto comparado for **igual** a esse objeto, um número **negativo** se esse objeto for **menor** que o objeto dado, e um número **positivo** se esse objeto for **maior** que o objeto dado. Para ordenar instâncias da classe `Carro` em ordem alfabética do atributo `marca`, basta implementar a interface `Comparable`, conforme apresentado no Código 5.3.

```java
/**
 * Classe Carro
 * @author Delano Medeiros Beder
 */
public class Carro implements Comparable<Carro> {

    /* Declaração dos atributos da classe */
    private String marca, cor;
    private int ano;
    private Pessoa dono;

    /* Construtor e demais métodos omitidos */
    public int compareTo(Carro o) {
        return this.marca.compareTo(o.marca);
    }

    public String toString() {
        return marca + " " + cor + "(" + ano + ")";
    }
}
```

Código 5.3 Classe `Carro` – implementando a interface `Comparable`.

É importante mencionar que a maioria das classes (**`String`**, **`Integer`**, etc.) disponíveis na plataforma Java já foi projetada de forma a implementar a interface **`Comparable`**. Como pode ser observado, o método **`compareTo()`** da classe **`Carro`** baseia-se enormemente no método **`compareTo()`** implementado pela classe **`String`** – o atributo **`marca`** é uma instância da classe **`String`**. Outro ponto importante é que o critério de ordenação é totalmente aberto, definido pelo projetista/programador da classe. Caso seja necessário ordenar por outro atributo (ou até por uma combinação de atributos), basta modificar a implementação do método **`compareTo()`** na classe.

> **Definindo outros critérios de ordenação**: é possível definir outros critérios de ordenação usando um objeto que implementa a interface `Comparator`. Existe um método `sort()` na classe `Collections` que recebe, além de `List`, um `Comparator` definindo um critério de ordenação específico. Logo, é possível ter várias instâncias de objetos que implementam a interface `Comparator`, cada uma definindo um critério diferente de ordenação, para usar quando necessário.

```java
/**
 * Classe Pessoa
 *
 * @author Delano Medeiros Beder
 */
public class Pessoa {
    /* Declaração dos atributos da classe */

    private String CPF, nome;
    private List<Carro> carros;

   /* Construtor e demais métodos omitidos - Ver Código 5.2 */

  public void imprimeCarros() {
      Collections.sort(carros);
      for (Carro c: carros) {
          System.out.println(c);
      }
  }

  public static void main(String[] args) {
      Pessoa josé = new Pessoa("0123456789", "José da Silva");
      Carro ferrari = new Carro("Ferrari", "Vermelho", 2012);
      Carro audi = new Carro("Audi", "Branco", 2011);
      Carro porsche = new Carro("Porche", "Amarelo", 2013);

      josé.adicionaCarro(ferrari);
      josé.adicionaCarro(audi);
      josé.adicionaCarro(porsche);

      josé.imprimeCarros();
  }
}
```

Código 5.4 Método **`imprimeCarros()`** – impressão em ordem alfabética.

Tendo em vista que a classe **`Carro`** agora implementa a interface **`Comparable`**, é possível invocar o método **`sort()`** para ordenar uma lista de instâncias da classe

Carro. Dessa forma, o método **imprimeCarros()**, apresentado no Código 5.4, foi ligeiramente modificado para imprimir os carros em ordem alfabética. Conforme se pode notar, há uma invocação ao método **sort()** antes da impressão dos carros. Fica como exercício para o leitor – em que ordem os carros serão impressos?

5.6 Conjuntos

A Figura 5.4 apresenta a interface **Set** e suas principais extensões e implementações. Um conjunto, representado pela interface **Set**, é uma coleção que funciona de forma análoga aos conjuntos matemáticos. Ou seja, é uma coleção que não permite elementos duplicados.

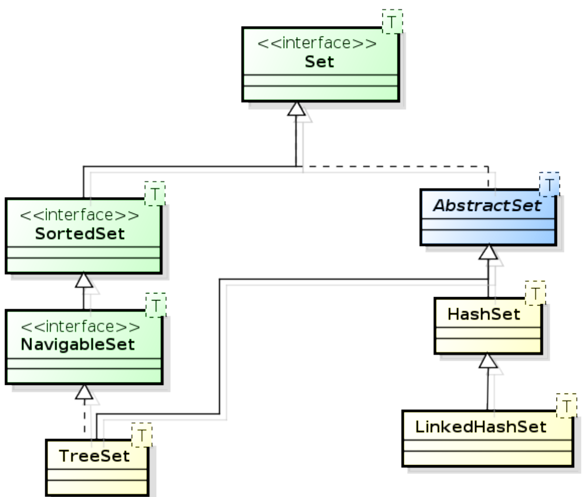


Figura 5.4 Conjuntos – hierarquia de classes e interfaces.

Outra característica fundamental de conjuntos Java é o fato de que a ordem em que os elementos são armazenados pode não ser a ordem na qual eles foram inseridos no conjunto. A interface **Set** não define como deve ser esse comportamento. Tal ordem varia de implementação para implementação.

O uso de um **Set** pode parecer desvantajoso, já que ele não armazena a ordem e não aceita elementos repetidos. Não há métodos que trabalham com índices, como o **get(int)** que as listas possuem. A grande vantagem do **Set** é que existem implementações, como a **HashSet**, que possuem uma performance incomparável com as **Lists** quando usadas para pesquisa.

Como se pode observar pela Figura 5.1, a interface **Set** é derivada da interface **Collection**. Portanto, os métodos definidos na interface **Collection**, discutidos

na Seção 5.3.1, também podem ser invocados em conjuntos – instâncias das classes concretas que implementam a interface `Set`.

A interface `SortedSet` é uma extensão da interface `Set` que agrega o conceito de ordenação ao conjunto. Os elementos do conjunto são ordenados utilizando a ordenação natural (os elementos devem implementar a interface `Comparable`) ou através de um objeto `Comparator` que deve ser passado como argumento, no momento da criação do objeto, ao construtor da classe. A interface `SortedSet` introduz os seguintes métodos:

- **T first()** – retorna o primeiro (menor) elemento do conjunto ordenado;
- **SortedSet<T> headSet(T element)** – retorna um subconjunto dos elementos **menores** que o elemento passado como argumento do método;
- **T last()** – retorna o último (maior) elemento do conjunto ordenado;
- **SortedSet<T> subSet(T from, T to)** – retorna o subconjunto com todos os elementos contidos entre os dois elementos especificados como argumentos do método;
- **SortedSet<T> tailSet(T element)** – retorna um subconjunto dos elementos **maiores** que o elemento passado como argumento do método.

A interface `NavigableSet` é uma extensão da interface `SortedSet` que agrega o conceito de navegação ao conjunto, introduzindo a definição de métodos, tais como:

- **ceiling(T element)** – retorna o menor elemento no conjunto que seja maior ou igual ao elemento passado como argumento, ou `null`, caso não exista o elemento;
- **NavigableSet<T> descendingSet()** – retorna um conjunto em que a ordem é a inversa, se comparada ao conjunto original;
- **floor(T element)** – retorna o maior elemento no conjunto que seja menor ou igual ao elemento passado como argumento, ou `null`, caso não exista o elemento.

A classe abstrata `AbstractSet` (Figura 5.4) implementa parcialmente a interface `Set`. Ou seja, essa classe fornece uma implementação parcial dos métodos definidos

nessa interface. No entanto, essa implementação parcial facilita a definição das implementações customizadas de conjuntos – subclasses concretas de **`AbstractSet`**: **`HashSet`**, **`LinkedHashSet`** e **`TreeSet`**.

A classe concreta **`HashSet`** é uma implementação da interface **`Set`** que utiliza uma tabela de espalhamento[2]para detectar e impedir elementos duplicados. As operações básicas (adicionar, remover, pesquisar, etc.) são realizadas em tempo constante, assumindo que função *hash* dispersa os elementos uniformemente na tabela. Quanto à ordem dos elementos no conjunto, a classe **`HashSet`** não oferece nenhuma garantia.

A classe concreta **`LinkedHashSet`** é subclasse da classe **`HashSet`**. Essa implementação diferencia-se da classe **`HashSet`** por manter uma lista duplamente ligada dos elementos presentes no conjunto. Dessa forma, essa classe garante que a ordem dos elementos do conjunto é a ordem que os elementos foram adicionados. Quanto ao custo das operações básicas, elas também são realizadas em tempo constante. Porém, elas são um pouco mais custosas que as similares implementadas pela classe **`HashSet`**, devido à manipulação da lista duplamente ligada.

Por fim, a classe **`TreeSet`** é uma implementação da interface **`NavigableSet`** que garante que a ordem dos elementos do conjunto é a ordenação natural dos elementos. Ou seja, os elementos devem implementar a interface **`Comparable`**, ou um objeto **`Comparator`** deve ser passado como argumento ao construtor da classe **`TreeSet`**. A complexidade temporal das operações básicas (adicionar, remover, pesquisar, etc.) é *log(n)*, em que *n* representa o número de elementos do conjunto.

5.6.1 Sistema de gerenciamento de obras de arte

Na Seção 3.7 foi discutida a implementação de um sistema de gerenciamento de obras de arte de um museu. Conforme discutido, a implementação do sistema continha algumas deficiências devido à utilização de um *array* de instâncias da classe **`ObraDeArte`**. Esta seção apresenta uma implementação mais robusta para o mesmo sistema a qual se baseia no *framework* de coleções presente na linguagem Java.

Suponha que um novo requisito do sistema é a necessidade de imprimir as obras de artes em ordem alfabética de seu título. Para ordenar instâncias das subclasses de **`ObraDeArte`** em ordem alfabética do atributo **`título`**, basta implementar a interface **`Comparable`**, conforme apresentado no Código 5.5. Vale a pena salientar que a classe **`ObraDeArte`** é abstrata, porém o método **`compareTo()`** é herdado pelas subclasses

[2] Na verdade, uma instância da classe `HashMap` (Seção 5.8).

(classes **Escultura** e **Pintura**). Dessa forma, é possível ordenar obras de artes (pinturas e esculturas) em ordem alfabética de seu título.

```
/**
 * Classe ObraDeArte
 *
 * @author Delano Medeiros Beder
 */
public abstract class ObraDeArte implements Comparable<ObraDeArte>{

    /* Declaração dos atributos da classe */
    private String título, artista, material;
    private int ano;

    /* Construtor e demais métodos omitidos - Ver Código 3.6 */

    public int compareTo(ObraDeArte o) {
        return título.compareTo(o.título);
    }
}
```

Código 5.5 Classe `ObraDeArte` – implementando a interface `Comparable`.

O Código 5.6 apresenta a implementação da classe **Museu** que apenas define o atributo **obras**, um conjunto parametrizado pelo tipo **ObraDeArte**. Dessa forma, nesse conjunto apenas podem ser inseridos instâncias das subclasses de **ObraDeArte**, tendo em vista que a classe **ObraDeArte** é abstrata.

Conforme se pode observar pelo construtor da classe, a classe **TreeSet** foi utilizada como a implementação da interface **Set**. Foram definidos dois métodos, **adicionaObra()** e **removeObra()**, que são responsáveis pela inserção e remoção de instâncias das subclasses de **ObraDeArte**. O método **imprimeColeção()** é responsável pela impressão das informações relacionadas a cada obra de arte presente no conjunto.

Finalmente, essa classe possui o método **main()**, que é o ponto de entrada desse programa. Conforme se pode observar nesse método, são criadas uma instância da classe **Pintura** e uma instância da classe **Escultura** que são adicionadas ao museu. E, por fim, a coleção de obras de arte é impressa (Figura 5.5).

```
==================== ESCULTURA ====================
Título : David                Artista : Michelangelo
Material : Mármore            Ano : 1501
Altura : 4.1
===================== PINTURA =====================
Título : Mona Lisa            Artista : Leonardo da Vinci
Material : Madeira            Ano : 1503
Tipo : Óleo
```

Figura 5.5 Execução da classe `Museu`.

```
/**
 * Classe Museu
 *
 * @author Delano Medeiros Beder
 */
public class Museu {
    /* Declaração dos atributos da classe */
    private Set<ObraDeArte> obras;

    /* Declaração do construtor da classe */
    public Museu() {
        obras = new TreeSet<>();
    }

    /* Declaração dos métodos da classe */
    public void adiciona(ObraDeArte obra) {
        obras.add(obra);
    }

    public void imprimeColeção() {
        for (ObraDeArte obra : obras) {
            obra.imprime();
        }
    }

    public static void main(String[] args) {
        Museu museu = new Museu();
        ObraDeArte o1 = new Pintura("Mona Lisa", "Leonardo da Vinci", "Madeira", 1503, "Óleo");
        museu.adiciona(o1);
        ObraDeArte o2 = new Escultura("David", "Michelangelo", "Mármore", 1501, 4.10);
        museu.adiciona(o2);
        museu.imprimeColeção();
    }
}
```

Código 5.6 Classe `Museu` – impressão em ordem alfabética.

5.7 Filas

A Figura 5.6 apresenta a interface `Queue` e suas principais implementações. Uma fila representa uma coleção ordenada de objetos, assim como uma lista, porém a semântica de utilização é ligeiramente diferente. Uma fila geralmente é projetada para ter os elementos inseridos no fim da fila e os elementos removidos a partir do início da fila (FIFO – *first-in, first-out*). Exceções a esse comportamento são as filas de prioridades, em que os elementos são inseridos de acordo com um comparador fornecido ou ordem natural dos elementos. A interface `Queue` introduz os seguintes métodos:

- **`boolean offer(T element)`** – insere o elemento especificado na fila;
- **`T peek()`** – retorna o elemento na cabeça da fila sem removê-lo, ou **`null`**, se a fila estiver vazia;
- **`T poll()`** – remove e retorna o elemento presente na cabeça da fila, ou **`null`**, se a fila estiver vazia;

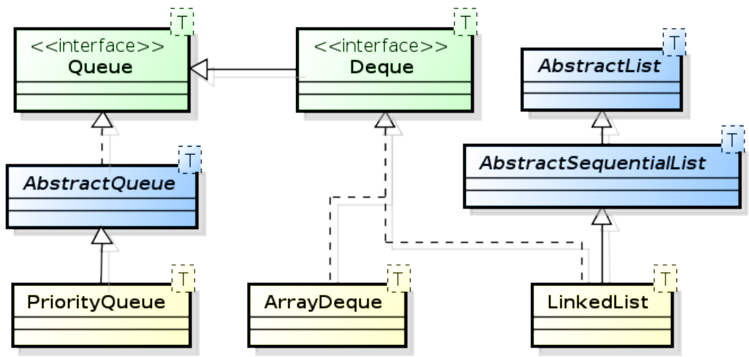


Figura 5.6 Filas – hierarquia de classes e interfaces.

Como se pode observar pela Figura 5.1, a interface **`Queue`** é derivada da interface **`Collection`**. Portanto, os métodos definidos na interface **`Collection`**, discutidos na Seção 5.3.1, também podem ser invocados em filas – instâncias das classes concretas que implementam a interface **`Queue`**.

A classe abstrata **`AbstractQueue`** (Figura 5.6) implementa parcialmente a interface **`Queue`**. Ou seja, essa classe fornece uma implementação parcial dos métodos definidos nessa interface. No entanto, essa implementação parcial facilita a definição das implementações customizadas de filas – subclasses concretas de **`AbstractQueue`**: **`PriorityQueue`**.

A classe **`PriorityQueue`** representa filas de prioridades e implementa a interface **`Queue`** de forma que os elementos são inseridos de acordo com a ordenação natural dos elementos. Ou seja, os elementos devem implementar a interface **`Comparable`**, ou um objeto **`Comparator`** deve ser passado como argumento ao construtor da classe **`PriorityQueue`**. Filas de prioridades serão discutidas na Seção 5.7.1.

A interface **`Deque`** representa um tipo especial de fila, em que as inserções e remoções de elementos podem ser realizadas em ambas as extremidades da fila. *Deques* podem ser usadas tanto como FIFOs (*first-in* , *first-out*) quanto como LIFOs (*last-in, first-out*).

A classe **`ArrayDeque`** oferece uma implementação básica da interface **`Deque`**. Ela define um atributo *array* com o objetivo de armazenar os elementos. Porém, esse *array* interno é encapsulado, e o programador não tem como acessá-lo.

A classe **`LinkedList`** (Seção 5.4) também oferece uma implementação da interface **`Deque`** que é baseada em listas duplamente encadeadas. Ou seja, a classe **`LinkedList`** implementa 3 interfaces do *framework* de coleções: **`List`**, **`Queue`** e **`Deque`**.

5.7.1 Filas de prioridades

Filas de prioridades são filas comuns as quais permitem que elementos sejam adicionados associados a uma prioridade. Cada elemento na fila deve possuir um dado adicional que representa sua prioridade de atendimento. Uma regra explícita define que o elemento de menor (ou maior) prioridade deve ser o primeiro a ser removido da fila, quando uma remoção é requerida. No caso da classe **`PriorityQueue`**, a cabeça da fila é o menor elemento, de acordo com o critério de ordenação especificado.

As filas de prioridade são utilizadas em diversas situações. Por exemplo, em sistemas operacionais, processos possuem diferentes prioridades de execução, e processos de maior prioridade são escalonados preferencialmente no processador (CPU).

O Código 5.7 apresenta um exemplo do uso de filas de prioridades no escalonamento de processos. A classe **`Processo`** representa um processo do sistema operacional. Conforme se pode observar, os processos são comparados de acordo com o atributo **`prioridade`**. O processo de maior prioridade é considerado o "menor", de acordo com o critério implementado no método **`compareTo()`**.

```java
/**
 * Classe Processo
 *
 * @author Delano Medeiros Beder
 */
public class Processo implements Comparable<Processo> {

    /* Declaração do atributo da classe */
    private int prioridade;

    /* Declaração do construtor da classe */
    public Processo(int prioridade) {
        this.prioridade = prioridade;
    }

    /* Demais métodos omitidos */
    public int compareTo(Processo o) {
        return o.prioridade - this.prioridade;
    }

    public static void main(String[] args) {
        Queue<Processo> fila = new PriorityQueue<Processo>();

        fila.add(new Processo(58));
        fila.add(new Processo(74));
        fila.add(new Processo(27)); fila.poll();
        fila.add(new Processo(29)); fila.poll();
        fila.add(new Processo(18)); fila.poll();
        fila.poll();
        fila.poll();
    }
}
```

Código 5.7 Classe **`Processo`** – escalonamento de processos.

A Figura 5.7 apresenta os estados da fila de prioridade de acordo com a execução do método `main()` do Código 5.7.

- Na linha 17, a fila de prioridade é criada vazia;
- Na linha 19, é adicionado um processo de prioridade **58**;
- Na linha 20, é adicionado um processo de prioridade **74**. Visto que esse processo tem prioridade maior que os demais elementos, ele torna-se a cabeça da fila de prioridade;
- Na linha 21, são realizadas duas operações na fila: (1) é adicionado um processo de prioridade **27** e (2) é removido o elemento presente na cabeça da fila (**74**);
- Na linha 22, são realizadas duas operações na fila: (1) é adicionado um processo de prioridade **29** e (2) é removido o elemento presente na cabeça da fila (**58**);
- Na linha 23, são realizadas duas operações na fila: (1) é adicionado um processo de prioridade **18** e (2) é removido o elemento presente na cabeça da fila (**29**);
- Na linha 24, é removido o elemento presente na cabeça da fila (**27**);
- Na linha 24, é removido o elemento presente na cabeça da fila (**18**).

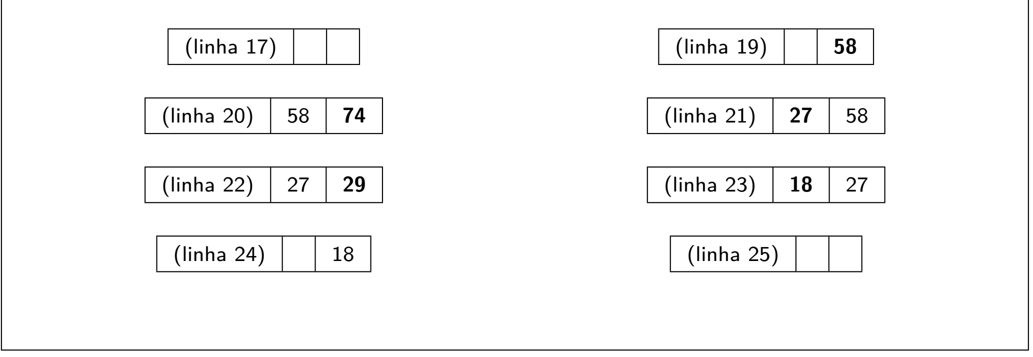


Figura 5.7 Fila de prioridades.

5.8 Mapas

A Figura 5.8 apresenta a interface `Map` e suas principais extensões e implementações. Um mapa, representado pela interface `Map`, é um conjunto de mapeamentos, ou associações, entre um objeto-chave <K> e um objeto-valor associado <V>, em que as chaves são únicas. É equivalente ao conceito de dicionário ou *array* associativo presente em algumas linguagens.

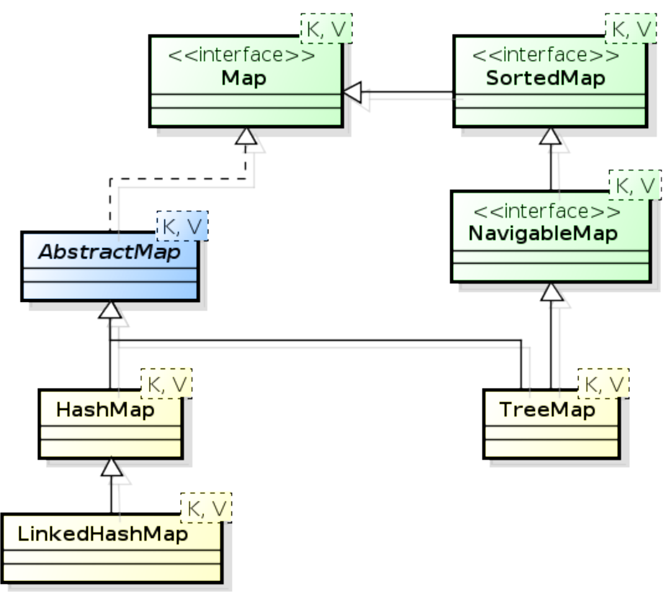


Figura 5.8 Mapas – hierarquia de classes e interfaces.

Um mapa pode ser visualizado como uma tabela com 2 colunas. Cada linha dessa tabela representa uma associação entre um objeto-chave e um objeto-valor. As chaves e os valores devem ser referências a objetos. Valores de tipos primitivos devem ser "empacotados" nas classes *wrappers* correspondentes – **`int`** como **`Integer`**, **`char`** como **`Character`**, etc. A interface `Map` define os seguintes métodos:

- **`boolean containsKey(Object key)`** – retorna **`true`** caso o mapa contenha uma associação para a chave passada como argumento;
- **`boolean containsValue(Object value)`** – retorna **`true`** caso o mapa contenha uma ou mais associações entre chaves e o valor passado como argumento;

- **Set< Entry<K,V> > entrySet()** – retorna um conjunto[3] de mapeamentos chave/valor (instâncias da classe **Entry<K,V>**) contidos no mapa. A classe **Entry<K,V>** define dois métodos:
  - **K getKey()** – retorna a chave do mapeamento chave/valor;
  - **V getValue()** – retorna o valor do mapeamento chave/valor.
- **V get(Object key)** – retorna o valor associado à chave passada como argumento, ou **null**, caso o mapa não contenha nenhuma associação para essa chave;
- **Set<K> keySet()** – retorna um conjunto das chaves contidas no mapa;
- **V put(K key, V value)** – insere no mapa uma associação entre chave/value (argumentos do método);
- **V remove(Object key)** – remove do mapa a associação para a chave passada como argumento, caso exista;
- **int size()** – retorna a quantidade de associações chave/valor contidas no mapa;
- **Collection<V> values()** – retorna uma coleção dos valores contidos no mapa.

A interface **SortedMap** é uma extensão da interface **Map** que agrega o conceito de ordenação ao mapa. As chaves do mapa são ordenadas utilizando a ordenação natural (as chaves devem implementar a interface **Comparable**) ou através de um objeto **Comparator** que deve ser passado como argumento, no momento da criação do objeto, ao construtor da classe. A interface **SortedMap** introduz os seguintes métodos:

- **K firstKey()** – retorna a primeira (menor) chave contida no mapa;
- **SortedMap<T> headMap(K key)** – retorna uma porção do mapa (associações chave/valor) cujas chaves são **menores** que a chave passada como argumento;
- **K lastKey()** – retorna a última (maior) chave contida no mapa;

[3] Instância de alguma classe concreta que implementa a interface Set.

- `SortedMap<T> subMap(K fromKey, K toKey)` – retorna uma porção do mapa (associações chave/valor) cujas chaves são **maiores** ou iguais a `fromKey` e **menores** que `toKey`;
- `SortedMap<T> tailMap(K key)` – retorna uma porção do mapa (associações chave/valor) cujas chaves são **maiores** que a chave passada como argumento.

A interface `NavigableMap` é uma extensão da interface `SortedMap` que agrega o conceito de navegação ao mapa, introduzindo a definição de métodos, tais como:

- `Entry<K,V> ceilingEntry(K key)` – retorna o mapeamento chave/valor associado à menor chave que seja maior ou igual à chave passada como argumento, ou `null`, se a chave não está contida no mapa;
- `K ceilingKey(K key)` – retorna a menor chave que seja maior ou igual à chave passada como argumento, ou `null`, se a chave não estiver contida no mapa;
- `NavigableSet<T> descendingKeySet()` – retorna o conjunto das chaves na ordem inversa, se comparada ao conjunto de chaves contido no mapa;
- `Entry<K,V> firstEntry()` – retorna o mapeamento chave/valor associado à menor chave no mapa, ou `null`, se o mapa estiver vazio;
- `Entry<K,V> floorEntry(K key)` – retorna o mapeamento chave/valor associado à maior chave que seja menor ou igual à chave passada como argumento, ou `null`, se a chave não estiver contida no mapa;
- `K floorKey(K key)` – retorna a maior chave que seja menor ou igual à chave passada como argumento, ou `null`, se a chave não estiver contida no mapa;
- `Entry<K,V> lastEntry()` – retorna o mapeamento chave/valor associado à maior chave no mapa, ou `null`, se o mapa estiver vazio.

A classe abstrata `AbstractMap` (Figura 5.8) implementa parcialmente a interface `Map`. Ou seja, essa classe fornece uma implementação parcial dos métodos definidos nessa interface. No entanto, essa implementação parcial facilita a definição das implementações customizadas de conjuntos – subclasses concretas de `AbstractMap`: `HashMap`, `LinkedHashMap` e `TreeMap`.

A classe concreta `HashMap` é uma implementação da interface `Set` que utiliza uma tabela de espalhamento. As operações básicas (adicionar, remover, pesquisar,

etc.) são realizadas em tempo constante, assumindo que função *hash* dispersa os elementos uniformemente na tabela. Quanto à ordem das associações chave/valor no mapa, a classe `HashMap` não oferece nenhuma garantia.

A classe concreta `LinkedHashMap` é subclasse da classe `HashMap`. Essa implementação diferencia-se da classe `HashMap` por manter uma lista duplamente ligada das associações chave/valor presentes no mapa. Dessa forma, essa classe garante que a ordem das associações chave/valor do mapa é a ordem em que as associações chave/valor foram adicionadas. Quanto ao custo das operações básicas, elas também são realizadas em tempo constante. Porém, elas são um pouco mais custosas que as similares implementadas pela classe `HashMap`, devido à manipulação da lista duplamente ligada.

Por fim, a classe `TreeMap` é um implementação da interface `NavigableMap` que garante que a ordem das associações chave/valor do mapa é a ordenação natural das chaves. Ou seja, as chaves devem implementar a interface `Comparable`, ou um objeto `Comparator` deve ser passado como argumento ao construtor da classe `TreeMap`. A complexidade temporal das operações básicas (adicionar, remover, pesquisar, etc.) é *log(n)*, em que *n* representa o número de associações chave/valor presentes no mapa.

5.8.1 Frequência de letras em um texto

Para exemplificar o emprego de mapas no desenvolvimento de sistemas, considere o problema abaixo:

> Dado um arquivo texto, calcule e imprima a frequência de cada uma das letras ('a' – 'z'), ignorando a diferença entre as letras maiúsculas e minúsculas. As frequências dos caracteres devem ser impressas em ordem alfabética dos caracteres. Os demais caracteres devem ser ignorados.

O Código 5.8 apresenta um programa Java que implementa uma solução para o problema de cálculo de frequência de letras em um arquivo texto.

- Na linha 10, uma instância da classe `Scanner` é criada. Essa instância é utilizada para ler os caracteres presentes no arquivo de entrada;
- Na linha 12, uma instância da classe `TreeMap` é criada. Os objetos-chaves desse mapa são as letras presentes no arquivo de entrada, enquanto os objetos-valores são as frequências de cada uma dessas letras;

```java
/**
 * Classe FrequenciaLetras
 *
 * @author Delano Medeiros Beder
 */
public class FrequenciaLetras {

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

        Scanner sc = new Scanner(new FileInputStream("texto.txt"));

        Map<Character, Integer> mapa = new TreeMap<Character, Integer>();

        while (sc.hasNext()) {
            String s = sc.next();
            for (int i = 0; i < s.length(); i++) {
                char c = s.charAt(i);
                if (Character.isLetter(c)) {
                    c = Character.toUpperCase(c);
                    Integer qtde = mapa.get(c);
                    if (qtde != null) {
                        qtde = qtde + 1;
                    } else {
                        qtde = 1;
                    }
                    mapa.put(c, qtde);
                }
            }
        }

        Iterator<Entry<Character, Integer>> it = mapa.entrySet().iterator();

        while (it.hasNext()) {
            Entry<Character, Integer> entry = it.next();
            System.out.println(entry.getKey() + " " + entry.getValue());
        }
    }
}
```

Código 5.8 Mapas – frequência de letras.

- Nas linhas 14 – 29, todos os caracteres do arquivo são lidos. Na linha 18, verifica-se se o caractere lido é uma letra. Caso não seja letra, o caractere é ignorado;
- Na linha 19, a letra é convertida para maiúscula; Nas linhas 20 – 25, verifica-se se a letra lida se encontra no mapa. Se a letra se encontra no mapa, a frequência é incrementada e posta novamente no mapa. Caso contrário, é colocada uma nova associação chave/valor no mapa: a chave é a letra lida e o valor é 1, indicando que é a primeira ocorrência dessa letra;
- Por fim, nas linhas 31 – 36, são impressas as frequências das letras presentes no arquivo de entrada. É importante salientar que um `Iterator` de instâncias da classe `Entry`, que representa o par chave/valor, é utilizado para realizar a impressão das frequências.

5.9 Algoritmos

A classe **Collections** define alguns métodos que executam computações úteis, tais como pesquisa e ordenação, em objetos que implementam as interfaces presentes no *framework* de coleções. Esses métodos são considerados polimórficos, isto é, o mesmo método pode ser usado em muitas implementações das interfaces.

Na Seção 5.5, foi discutido o método **sort()**, que ordena uma lista em ordem crescente. Nesta seção, são discutidos outros métodos definidos na classe **Collections**. O Código 5.9 ilustra o emprego de alguns desses métodos.

- O método **binarySearch()** busca um elemento em uma lista ordenada utilizando o algoritmo de busca binária;
- Os métodos **min()** e **max()** retornam, respectivamente, o mínimo e o máximo valor contido em uma coleção;
- O método **reverseOrder()** retorna uma instância da classe **Comparator** que, quando utilizado, ordena uma coleção de objetos em ordem inversa à ordem natural definida pela interface **Comparable**;
- O método **rotate()** rotaciona (esquerda para direita) os elementos de uma lista em uma distância especificada.

```
public class Algoritmos {

    public static void main(String[] args) {
        Integer array[] = {9, 1, 2, 8, 7, 3, 4, 6, 5};

        List<Integer> lista = Arrays.asList(array);
        System.out.println(lista); // imprime [9, 1, 2, 8, 7, 3, 4, 6, 5]

        System.out.println(Collections.min(lista)); // imprime 1, pois ele é o menor elemento
        System.out.println(Collections.max(lista)); // imprime 9, pois ele é o maior elemento

        Collections.rotate(lista, 3);
        System.out.println(lista); // imprime [4, 6, 5, 9, 1, 2, 8, 7, 3]

        Collections.sort(lista);
        System.out.println(lista); // imprime [1, 2, 3, 4, 5, 6, 7, 8, 9]

        System.out.println(Collections.binarySearch(lista, 3)); // imprime 2 (posição do elemento 3)

        Collections.sort(lista, Collections.reverseOrder());
        System.out.println(lista); // imprime [9, 8, 7, 6, 5, 4, 3, 2, 1]
    }
}
```

Código 5.9 Classe **Collections** – métodos polimórficos.

5.10 Considerações finais

Esta unidade discutiu o *framework* de coleções presente na linguagem Java. A próxima unidade discute um extenso conjunto de classes e interfaces, presentes na *API* padrão da linguagem Java, para o desenvolvimento de interfaces gráficas com usuários (*GUIs*).

5.11 Estudos complementares

Para estudos complementares sobre os tópicos abordados nesta unidade, o leitor interessado pode consultar as seguintes referências:

ARNOLD, K.; GOSLING, J.; HOLMES, D. *The Java Programming Language*. 4. ed. Boston: Addison-Wesley, 2005.

CAELUM. *Apostila do curso FJ-11* – Java e Orientação a Objetos. 2014. Disponível em: http://www.caelum.com.br/apostila-java-orientacao-objetos. Acesso em: 12 ago. 2014.

CAMARÃO, C.; FIGUEIREDO, L. *Programação de Computadores em Java*. 1. ed. São Paulo: LTC, 2003.

DEITEL, P.; DEITEL, H. *Java*: Como programar. 8. ed. São Paulo: Pearson Brasil, 2010.

ORACLE. *The Java^{TM} Tutorials* – Trail: Collections Java. 2014. Disponível em: http://docs.oracle.com/javase/tutorial/collections/. Acesso em: 12 ago. 2014.

ORACLE. *Java SE Documentation* – Collections Framework Overview. 2014. Disponível em: http:/docs.oracle.com/javase/7/docs/technotes/guides/collections/overview.html. Acesso em: 12 ago. 2014.

UNIDADE 6

Interface gráfica em Java

6.1 Primeiras palavras

A linguagem de programação Java oferece, dentre as funcionalidades incorporadas à sua *API* padrão, um extenso conjunto de classes e interfaces para o desenvolvimento de aplicações gráficas. Esse conjunto facilita a criação de interfaces gráficas com usuários (*GUIs*[1]) e de saídas na forma gráfica, tanto na forma de aplicações *desktop* quanto na forma de *applets* (Seção 8.4).

Aplicações gráficas são criadas através da utilização de componentes gráficos, que estão agrupados em dois grandes pacotes: `java.awt` e `javax.swing`. Esta unidade tem como objetivo apresentar esses dois pacotes que fornecem um conjunto de componentes gráficos incorporados à *API* padrão da linguagem Java. As bibliotecas `AWT` e `Swing` fazem parte do Java Foundation Classes (JFC), uma biblioteca mais vasta que inclui também a *API* Java 2D™, entre outras.

6.2 Problematizando o tema

Ao final desta unidade, espera-se que o leitor seja capaz de reconhecer e definir precisamente os conceitos inerentes ao conjunto de componentes gráficos presente na linguagem Java. Dessa forma, esta unidade pretende discutir as seguintes questões:

- Quais as semelhanças e diferenças entre as bibliotecas `AWT` e `Swing` ?
- O que são gerenciadores de leiaute ? Para que servem ?
- Como são tratadas as ações dos usuários – por exemplo, o clique do *mouse* em um botão – quando realizadas na interface gráfica ?

6.3 AWT – *Abstract Windowing Toolkit*

A biblioteca `AWT`, *Abstract Windowing Toolkit*, é definida através das classes do pacote `java.awt` e seus subpacotes, tais como `java.awt.event`. Essas classes agrupam as funcionalidades gráficas, presentes desde a primeira versão de Java, que operam tendo por base as funcionalidades de alguma biblioteca gráfica do sistema operacional em que a aplicação é executada. Ou seja, a biblioteca `AWT` foi projetada de forma que cada máquina virtual Java implemente seu elemento de interface gráfica.

[1] Em inglês: *Graphical User Interfaces*.

Dessa forma, um componente gráfico (por exemplo, um botão) terá diferentes aparências (*look-and-feel*) se executado em diferentes sistemas operacionais. No entanto, a funcionalidade mantém-se a mesma.

A biblioteca **AWT** contém todas as classes para criação de interfaces do usuário (*GUI*) e para trabalhos gráficos e com imagens. O desenvolvimento de aplicações gráficas está fora do escopo deste livro. No entanto, o leitor interessado pode consultar a *API* Java 2D™, que fornece funcionalidades para a manipulação de gráficos bidimensionais, textos e imagens, através do seguinte link:

- *API* Java 2D™: http://docs.oracle.com/javase/tutorial/2d/overview/index.html.

6.3.1 Interfaces gráficas com usuários

Criar uma interface gráfica com usuários em Java envolve tipicamente a criação de um *container*, um componente que pode receber outros. Em uma aplicação gráfica *desktop*, tipicamente o *container* usado como base é um *frame* (uma janela com bordas e barra de título), enquanto em um *applet* (Seção 8.4) o *container-base* é um *panel* (uma área gráfica inserida em outra).

Após sua criação, os *containers* da aplicação recebem componentes (Seção 6.3.2) de interface com usuários, que permitirão apresentar e receber dados dos usuários. Embora seja possível posicionar esses componentes através de coordenadas absolutas em cada *container*, é recomendável que o projetista utilize sempre um gerenciador de leiaute (Seção 6.3.3) para realizar essa tarefa. Desse modo, garante-se que a aplicação possa ser executada independentemente das características da plataforma em que será executada.

Finalmente, é preciso especificar quais devem ser os efeitos das ações dos usuários, tais como um clique do *mouse* ou uma entrada de texto, quando realizadas sobre cada um desses componentes. Isso se dá através da especificação de classes manipuladoras de eventos (Seção 6.3.4), projetadas para a aplicação. Ao associar instâncias das classes manipuladoras de eventos aos componentes gráficos, o projetista determina o comportamento da aplicação.

6.3.2 Componentes gráficos

Um objeto de interface como um botão ou uma barra de rolagem é chamado de componente. A classe abstrata **Component** é a raiz de todos os componentes **AWT**.

A Figura 6.1 apresenta a hierarquia de componentes gráficos presentes na biblioteca **AWT** (pacote **java.awt**).

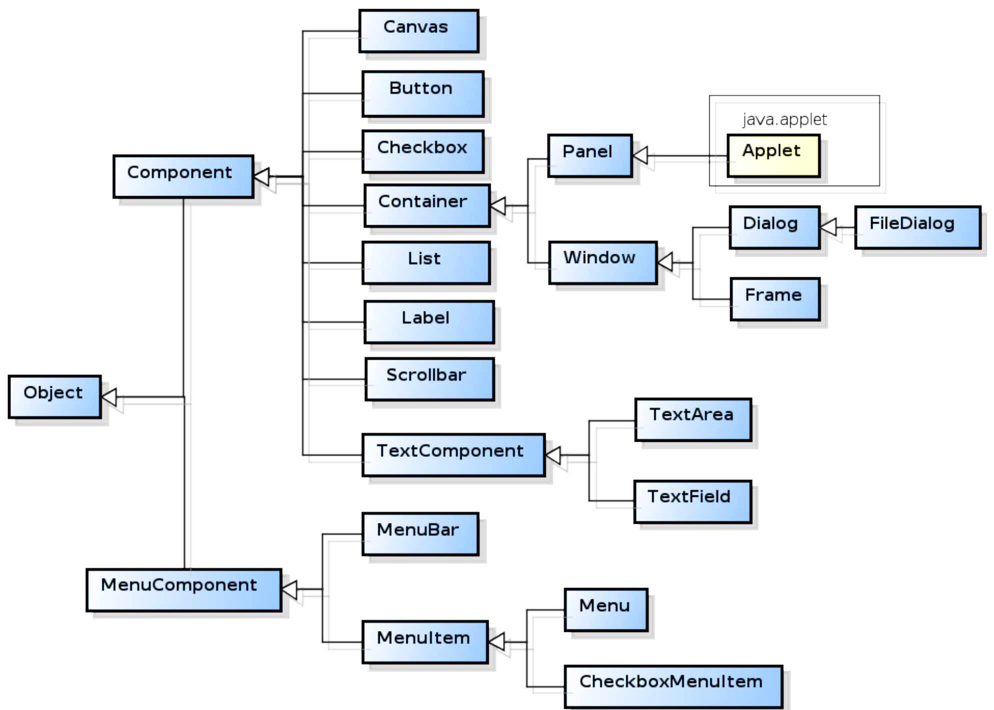


Figura 6.1 Hierarquia de componentes gráficos presentes na biblioteca **AWT**.

Os métodos da classe **Component** definem funcionalidades que dizem respeito à manipulação de qualquer componente gráfico em Java. Por exemplo, os métodos **getSize()** e **setSize()** permitem obter e definir o tamanho do componente, expresso na forma de um objeto da classe **Dimension** – um objeto dessa classe tem campos públicos **width** e **height** que indicam, respectivamente, as dimensões horizontais e verticais do componente em *pixels*. Outro grupo de métodos importantes dessa classe é composto dos métodos que permitem determinar os manipuladores de eventos (Seção 6.3.4) que são relevantes para qualquer tipo de componente gráfico.

A classe abstrata **Container**, subclasse da classe **Component**, representa componentes que podem conter componentes e outros *containers*. Os métodos **add()** e **remove()**, definidos por essa classe, são responsáveis por adicionar e remover componentes, respectivamente.

A classe **Window** é uma classe derivada de **Container** cujos objetos estão associados a janelas. Cada objeto **Window** é uma janela independente em uma aplicação. Raramente uma instância dessa classe é utilizada diretamente, porém a utilização de instâncias de suas subclasses, tais como **Frame**, é muito comum.

A classe **Frame**, subclasse da classe **Container**, representa *frames* que são normalmente vistos como janelas, com opções de minimizar, maximizar e fechar. Aplicações gráficas *desktop* tipicamente utilizam o *frame* como o *container-base*. As Figuras 6.2 e 6.3 apresentam dois exemplos simples de *frames* que contêm componentes gráficos.

A classe **Dialog**, subclasse da classe **Container**, representa caixas de diálogo. Uma caixa de diálogo é uma janela similar a um *frame* com a diferença de que é possível determiná-la como modal que altera o comportamento da janela – não é possível passar para outras janelas no mesmo aplicativo até que se feche a caixa de diálogo. Por fim, a classe **FileDialog** permite selecionar arquivos do sistema através de uma janela específica. É importante salientar que o **FileDialog** retorna apenas uma *string* contendo o nome e o diretório onde o arquivo está localizado.

Em muitas situações, é necessário separar o *frame* em áreas distintas, e em cada uma dessas áreas estaria um conjunto de componentes gráficos (botões, rótulos, etc.). A classe **Panel**, subclasse de **Container**, serve para esse propósito. O **Panel**, como os *frames*, permite que outros componentes gráficos sejam adicionados a ele. Em uma aplicação típica, um ou vários **Panels**, devem estar contidos em um *frame*.

A classe **Button**, subclasse da classe **Component**, representa componentes que podem ser usados para invocar alguma ação quando o usuário clica (pressiona e solta esse componente). Um botão é rotulado com uma *string*, que é sempre centralizada. A Figura 6.2 apresenta um exemplo de um botão **AWT**.

```java
package br.ufscar.si.poo2.awt;

import java.awt.Button;
import java.awt.Frame;

public class ButtonDemo {

    public static void main(String[] args) {
        Frame frame = new Frame("ButtonDemo");
        frame.setResizable(true);
        Button button = new Button("OK");
        frame.add(button);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Figura 6.2 Componentes gráficos **AWT** – classe **Button**.

A classe **List**, subclasse da classe **Component**, representa listas de opções textuais as quais são exibidas e permitem que mais de uma opção seja selecionada. A Figura 6.3 apresenta um exemplo de uma lista **AWT**.

```java
package br.ufscar.si.poo2.awt;

import java.awt.Frame;
import java.awt.List;

public class ListDemo {

    public static void main(String[] args) {
        Frame frame = new Frame("ListDemo");
        frame.setResizable(true);
        List list = new List(4, true);
        list.add("C++");
        list.add("C#");
        list.add("Java");
        list.add("Smalltalk");
        frame.add(list);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Figura 6.3 Componentes gráficos **AWT** – classe `List`.

A classe **Checkbox**, subclasse da classe **Component**, representa campos de seleção do tipo *on/off*. A Figura 6.4 apresenta três campos de seleção **AWT**.

```java
package br.ufscar.si.poo2.awt;

import java.awt.Checkbox;
import java.awt.FlowLayout;
import java.awt.Frame;

public class CheckboxDemo {

    public static void main(String[] args) {
        Frame frame = new Frame("CheckboxDemo");
        frame.setLayout(new FlowLayout());
        Checkbox box1 = new Checkbox("Azul");
        Checkbox box2 = new Checkbox("Verde");
        Checkbox box3 = new Checkbox("Vermelho");
        frame.add(box1);
        frame.add(box2);
        frame.add(box3);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Figura 6.4 Componentes gráficos **AWT** – classe `Checkbox`.

Quando é desejável que o usuário possa escolher apenas uma das opções (objetos **CheckBox**), é necessário associar esses objetos a uma instância da classe **CheckboxGroup**. A aparência mudará para um *radio button*. A Figura 6.5 apresenta três campos de seleção **AWT** associados a um mesmo grupo.

```java
package br.ufscar.si.poo2.awt;

import java.awt.Checkbox;
import java.awt.CheckboxGroup;
import java.awt.FlowLayout;
import java.awt.Frame;

public class CheckboxGroupDemo {

    public static void main(String[] args) {
        Frame frame = new Frame("CheckboxGroupDemo");
        frame.setLayout(new FlowLayout());
        CheckboxGroup group = new CheckboxGroup();
        Checkbox box1 = new Checkbox("Linux", group, true);
        Checkbox box2 = new Checkbox("Mac", group, false);
        Checkbox box3 = new Checkbox("Windows", group, false);
        frame.add(box1);
        frame.add(box2);
        frame.add(box3);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Figura 6.5 Componentes gráficos **AWT** – classe `CheckboxGroup`.

A classe **`TextComponent`**, subclasse de **`Component`**, é a superclasse dos componentes gráficos **`AWT`** que permitem a edição de texto. Um objeto **`TextComponent`** incorpora uma sequência de texto. As principais subclasses de **`TextComponent`** são as classes **`TextField`**, que representa uma única linha de texto, e **`TextArea`**, que é um campo de texto de múltiplas linhas e colunas.

A classe **`Label`**, subclasse da classe **`Component`**, permite desenhar uma *string* (rótulo) em um campo não editável. Adicionalmente, é possível definir a cor (método **`setForeground()`**) e a fonte (método **`setFont()`**) dos textos (**`TextArea`**, **`TextField`** e **`Label`**). A Figura 6.6 apresenta um exemplo de um *frame* composto de dois objetos **`Label`**, um objeto **`TextField`** e um objeto **`TextArea`**. É importante mencionar que a fonte e as cores dos rótulos foram definidas nesse exemplo.

```
public class TextDemo {

    public static void main(String[] args) {
        Frame frame = new Frame("TextoDemo");
        frame.setLayout(new GridLayout(2, 2));
        Label nomeLbl = new Label("Nome:");
        nomeLbl.setForeground(Color.BLUE);
        TextField nomeTxt = new TextField();
        Label endLbl = new Label("Endereço:");
        endLbl.setFont(new Font("TimesRoman", Font.ITALIC, 12));
        endLbl.setForeground(Color.RED);
        TextArea endTxt = new TextArea();
        frame.add(nomeLbl);
        frame.add(nomeTxt);
        frame.add(endLbl);
        frame.add(endTxt);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Figura 6.6 Componentes gráficos **`AWT`** – campos de texto.

A classe abstrata **`MenuComponent`** é a superclasse de todos os componentes relacionados à definição de menus. Nesse sentido, a classe **`MenuComponent`** é análoga à classe **`Component`**, que é a superclasse abstrata dos demais componentes **`AWT`**.

Menus são componentes que podem ser adicionados unicamente a *frames*. A Figura 6.7 apresenta um exemplo de *frame* com uma barra de menu associada. Para criar menus, é necessária a instanciação de três classes (**`MenuBar`**, **`Menu`** e **`MenuItem`**) através dos seguintes passos:

(a) Criação de uma barra de menu, representada pela classe **`MenuBar`**, e a posterior adição dessa barra de menu ao *frame*.

(b) Criação de instâncias da classe **`Menu`** e a posterior adição à barra de menu criada anteriormente. No exemplo apresentado na Figura 6.7, foram criados dois menus: **Arquivo** e **Ajuda**.

```
public class MenuDemo {

    public static void main(String[] args) {
        Frame frame = new Frame("MenuDemo");
        MenuBar bar = new MenuBar();
        frame.setMenuBar(bar);
        Menu menuFile = new Menu("Arquivo");
        MenuItem open = new MenuItem("Abrir", new MenuShortcut(KeyEvent.VK_A));
        MenuItem save = new CheckboxMenuItem("Salvar");
        MenuItem exit = new MenuItem("Sair");
        menuFile.add(open);
        menuFile.add(save);
        menuFile.addSeparator();
        menuFile.add(exit);
        Menu menuHelp = new Menu("Ajuda");
        bar.add(menuFile);
        bar.setHelpMenu(menuHelp);
        frame.pack();
        frame.setVisible(true);
    }
}
```


Figura 6.7 Componentes gráficos `AWT` – menus.

(c) Criação dos itens de menu, representados pela classe **`MenuItem`**, para os menus (instâncias de **`Menu`**) criados anteriormente. No exemplo apresentado na Figura 6.7, foram criados dois itens de menu (**Abrir** e **Salvar**) que foram adicionados ao menu **Arquivo**. Como se pode observar, é possível adicionar teclas de atalho (por exemplo, `CTRL+A`) aos itens de menu e criar itens de menu para seleção *on/off* (instâncias da classe **`CheckboxMenuItem`**). E, por fim, foi utilizado um separador (método **`addSeparator()`**) para dividir os itens.

6.3.3 Gerenciadores de leiautes

Quando um *container* tem mais de um componente, é preciso especificar como esses componentes devem ser dispostos na apresentação. Em Java, um objeto que implementa a interface `LayoutManager` é responsável por esse gerenciamento de disposição. A interface `LayoutManager2` é uma extensão da interface `LayoutManager` que incorpora o conceito de restrições de posicionamento de componentes em um *container*.

Os principais métodos da classe `Container` relacionados ao leiaute de componentes são `setLayout()`, que recebe como argumento um objeto que implementa a interface `LayoutManager` e determina qual a política de gerência de disposição de componentes adotada, e `validate()`, usado para rearranjar os componentes em um *container* se o gerenciador de leiaute sofrer alguma modificação ou novos componentes forem adicionados.

O pacote `java.awt` apresenta vários gerenciadores de leiaute predefinidos. As classes `FlowLayout` e `GridLayout` são implementações da interface `LayoutManager`. As classes `BorderLayout`, `CardLayout` e `GridBagLayout` são implementações da

interface LayoutManager2. *Containers* derivados da classe Window têm como padrão um gerenciador de leiaute do tipo BorderLayout, enquanto aqueles derivados da classe Panel usam como padrão um gerenciador de leiaute do tipo FlowLayout.

FlowLayout – essa classe arranja os componentes sequencialmente na janela, da esquerda para a direita, do topo para baixo, à medida que os componentes são adicionados ao *container*. Como padrão, os componentes são horizontalmente centralizados no container. É possível mudar esse padrão de alinhamento especificando um valor alternativo como parâmetro para um dos construtores da classe ou para o método setAlignment(). Esse parâmetro pode assumir um dos valores constantes definidos na classe, tais como LEFT ou RIGHT.

É possível também modificar a distância em *pixels* entre os componentes arranjados através desse tipo de gerenciador com os métodos setHgap() e setVgap(). Alternativamente, esses valores podem também ser especificados através de construtores da classe FlowLayout. As Figuras 6.4 e 6.5 apresentam dois exemplos de *frames*, compostos de campos de seleção, que utilizam o gerenciador de leiaute FlowLayout.

GridLayout – essa classe é uma implementação da interface LayoutManager que permite distribuir componentes ao longo de linhas e colunas. A distribuição dá-se na ordem de adição dos componentes ao *container*, da esquerda para a direita e de cima para baixo. Essa classe oferece um construtor que permite especificar o número de linhas e colunas desejado para o *grid* – o construtor padrão define um *grid* de uma única coluna.

Adicionalmente, há outro construtor que permite especificar, além da quantidade de linhas e colunas, o espaço em *pixels* entre os componentes nas direções horizontal e vertical. Deve-se observar que, nesse tipo de leiaute, as dimensões dos componentes são ajustadas para ocupar o espaço completo de uma posição no *grid*. A Figura 6.6 apresenta um exemplo de *frame* que utiliza o gerenciador de leiaute GridLayout para definir um *grid* de 2 linhas e 2 colunas na disposição dos componentes.

BorderLayout – essa classe é uma implementação da interface LayoutManager2 adequada para janelas com até cinco componentes. Ela permite arranjar os componentes de um *container* em cinco regiões, cujo posicionamento é representado pelas constantes definidas na classe BorderLayout: NORTH, SOUTH, EAST, WEST e CENTER.

Além do construtor padrão, outro construtor permite especificar a distância horizontal e vertical (em *pixels*) entre os componentes. Esse tipo de leiaute é o padrão para

containers do tipo **Frame**. Para utilizar esse tipo de gerenciador para janelas com mais de cinco componentes, basta definir que o componente inserido em um **BorderLayout** seja um **Panel**, que é um *container* que pode ter seu próprio gerenciador de leiaute. A Figura 6.8 apresenta um exemplo de um *frame* que utiliza o gerenciador de leiaute **BorderLayout** para apresentar os 5 botões nas regiões definidas pelo leiaute: norte, sul, leste, oeste e centro.

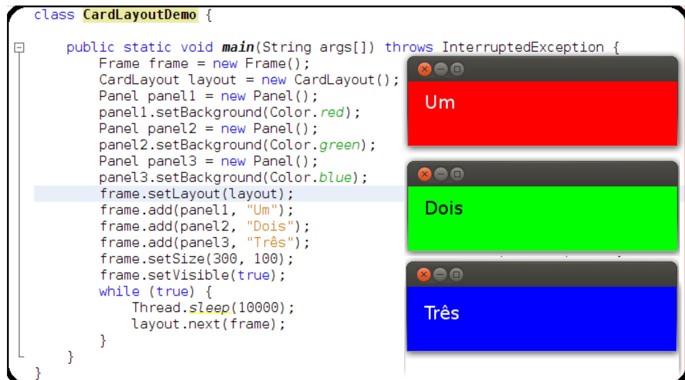
```
public class BorderLayoutDemo {

    public static void main(String[] args) {
        Frame f = new Frame("BorderLayoutDemo");
        BorderLayout layout = new BorderLayout();
        f.setLayout(layout);
        Button b1 = new Button("Norte");
        Button b2 = new Button("Sul");
        Button b3 = new Button("Leste");
        Button b4 = new Button("Oeste");
        Button b5 = new Button("Centro");
        f.add(b1, BorderLayout.NORTH);
        f.add(b2, BorderLayout.SOUTH);
        f.add(b3, BorderLayout.EAST);
        f.add(b4, BorderLayout.WEST);
        f.add(b5, BorderLayout.CENTER);
        f.setSize(200, 100);
        f.pack();
        f.setVisible(true);
    }
}
```

BorderLayoutDemo
Norte
Oeste
Centro
Leste
Sul

Figura 6.8 Gerenciador de leiaute **BorderLayout**.

CardLayout – essa classe é uma implementação da interface **LayoutManager2** que empilha os componentes de um *container* de tal forma que apenas o componente presente no topo é visível. Os métodos desse gerenciador estabelecem funcionalidades que permitem "navegar" entre os componentes empilhados, determinando qual item deve estar visível em um dado momento. Os métodos de navegação **first()**, **next()**, **previous()** e **last()** permitem realizar uma varredura sequencial pelos componentes do *container*.



```
class CardLayoutDemo {

    public static void main(String args[]) throws InterruptedException {
        Frame frame = new Frame();
        CardLayout layout = new CardLayout();
        Panel panel1 = new Panel();
        panel1.setBackground(Color.red);
        Panel panel2 = new Panel();
        panel2.setBackground(Color.green);
        Panel panel3 = new Panel();
        panel3.setBackground(Color.blue);
        frame.setLayout(layout);
        frame.add(panel1, "Um");
        frame.add(panel2, "Dois");
        frame.add(panel3, "Três");
        frame.setSize(300, 100);
        frame.setVisible(true);
        while (true) {
            Thread.sleep(10000);
            layout.next(frame);
        }
    }
}
```


Figura 6.9 Gerenciador de leiaute **CardLayout**.

Tipicamente, os componentes manipulados por um gerenciador do tipo **CardLayout** são *containers*, os quais por sua vez utilizam qualquer outro tipo de gerenciador de leiaute. A Figura 6.9 apresenta um exemplo do uso do gerenciador de

leiaute `CardLayout` para proporcionar a navegação entre três `Panels` – Um, Dois e Três.

`GridBagLayout` – essa classe é o gerenciador de leiaute mais flexível dentre aqueles predefinidos na biblioteca `AWT`. É também, em decorrência dessa flexibilidade, o mais complexo. É uma implementação da interface `LayoutManager2` que permite, como `GridLayout`, arranjar componentes ao longo de uma matriz de linhas e colunas. No entanto, componentes podem ser acrescentados em qualquer ordem e podem também variar em tamanho, possivelmente ocupando mais de uma linha ou coluna.

Uma vez que o desenho da interface tenha sido especificado, a chave para a utilização desse gerenciador é a criação de um objeto de restrição de posicionamento. Esse objeto é instância da classe `GridBagConstraints`. Objetos da classe `GridBagConstraints` determinam como um gerenciador do tipo `GridBagLayout` deve posicionar um dado componente em seu *container*.

Uma vez que esse objeto tenha sido criado e suas restrições especificadas, basta associar essas restrições ao componente usando o método `setConstraints()` e adicioná-lo ao *container* com o método `add()`, sendo o segundo parâmetro as restrições. A especificação das restrições de posicionamento, tamanho e propriedades de um componente nesse tipo de gerenciador é determinada através da atribuição de valores a campos públicos do objeto da classe `GridBagConstraints`.

O posicionamento é especificado pelas variáveis `gridx` e `gridy`, respectivamente para indicar a coluna e a linha onde o componente deve ser posicionado. Para `gridx`, o valor 0 indica a coluna mais à esquerda. Do mesmo modo, para `gridy`, o valor 0 indica a linha mais ao topo. Além de valores absolutos de posicionamento, essa classe define a constante `RELATIVE` para posicionamento relativo, após o último componente incluído, sendo este o valor padrão para esses campos.

O número de células que o componente ocupa no *grid* é indicado pelas variáveis `gridwidth` e `gridheight`, relacionadas respectivamente ao número de colunas e ao número de linhas que serão ocupadas pelo componente. O valor `REMAINDER` para esses campos indica que o componente será o último dessa linha ou coluna, devendo ocupar a largura ou altura restante. O valor padrão desses campos é 1.

Outras variáveis de restrição definidas nessa classe incluem `weightx`, `weighty`, `fill` e `anchor`. Os atributos `weightx` e `weighty` indicam o peso, ou a prioridade, que o componente terá para receber porções de espaço extra horizontalmente ou verticalmente, respectivamente, quando o *container* é redimensionado e espaço adicional

torna-se disponível. O padrão é um componente não receber espaço extra (valor 0).

O atributo **fill** é utilizado quando a área para a apresentação do componente é maior que o tamanho natural do componente. Indica como a apresentação do componente irá ocupar a área disponível, podendo assumir os valores definidos em constantes da classe: **NONE**, não modifica o tamanho do componente (o padrão); **VERTICAL**, ocupa o espaço vertical mas não altera a largura do componente; **HORIZONTAL**, ocupa o espaço disponível na horizontal, mas não altera a altura do componente; e **BOTH**, ocupa os espaços disponíveis nas duas dimensões.

O atributo **anchor** é utilizado quando o tamanho do componente é menor que a área da célula à qual ele foi alocado para indicar a posição do componente na célula. O padrão é **CENTER**, mas outros valores possíveis são **NORTH**, **NORTHEAST**, **EAST**, **SOUTHEAST**, **SOUTH**, **SOUTHWEST**, **WEST** e **NORTHWEST**.

Para maiores detalhes sobre a classe **GridBagLayout**, o leitor interessado pode consultar o link: http://docs.oracle.com/javase/tutorial/uiswing/layout/gridbag.html.

6.3.4 Eventos

No modelo de eventos suportado pela linguagem de programação Java, um componente gráfico pode reconhecer alguma ação do usuário e a partir dela disparar um evento – indicando, por exemplo, que o *mouse* foi pressionado ou que um texto foi modificado – que será capturado por um objeto registrado especificamente para tratar aquele tipo de evento ocorrido naquele componente.

O pacote **java.awt.event** define as diversas classes de eventos que podem ocorrer através das interfaces gráficas. Eventos gráficos são objetos derivados da classe **AWTEvent**, a qual por sua vez é derivada da classe **java.util.EventObject**. Dessa forma, eventos gráficos herdam o método **getSource()**, que permite identificar o objeto que deu origem ao evento.

Eventos gráficos genéricos são especializados de acordo com o tipo de evento sob consideração – por exemplo, pressionar um botão do *mouse* gera um objeto da classe **MouseEvent**. A ação sobre um botão gera um objeto da classe **ActionEvent**, e essa mesma ação sobre o botão de minimizar na barra de título de um *frame* gera um objeto da classe **WindowEvent**.

Outros eventos de interesse definidos em **java.awt.event** incluem: **ItemEvent**, indicando que um item de uma lista de opções foi selecionado; **TextEvent**, quando

o conteúdo de um componente de texto foi modificado; e `KeyEvent`, quando alguma tecla foi pressionada no teclado.

A resposta de uma aplicação a qualquer evento que ocorre em algum componente gráfico é determinada por métodos específicos, registrados para responder a cada evento. O nome e a assinatura de cada um desses métodos são determinados por uma interface Java do tipo *listener*, presente no pacote `java.awt.event`. Assim, para responder a um `ActionEvent`, será utilizado o método definido na interface `ActionListener`; para responder a um `WindowEvent`, os métodos de `WindowListener`. De maneira análoga, existem as interfaces `ItemListener`, `TextListener` e `KeyListener`. Os eventos do *mouse* são manipulados através de métodos especificados em duas interfaces: `MouseListener` (para ações sobre o mouse) e `MouseMotionListener` (para tratar movimentos realizados com o mouse).

Apesar de existir um grande número de eventos e possibilidades de resposta que a aplicação poderia dar a cada um deles, cada aplicação pode especificar apenas aqueles para os quais há interesse que sejam tratados. Para os eventos que o projetista da aplicação tem interesse de oferecer uma reação, ele deve definir classes manipuladoras de eventos (*handlers*) – implementações de cada *listener* correspondente ao tipo de evento de interesse.

Para interfaces *listener* que especificam diversos métodos, classes adaptadoras são definidas. Essas classes adaptadoras são classes abstratas definidas no pacote `java.awt.event` com nome `XXAdapter`, em que `XX` seria o prefixo correspondente ao tipo de evento de interesse. Dessa forma, para que a sua aplicação use uma classe adaptadora para tratar os eventos do tipo `WindowEvent`, é necessário criar uma subclasse de `WindowAdapter`. Nessa classe derivada, os métodos relacionados aos eventos de interesse devem ser reimplementados. Como a classe adaptadora implementa a interface correspondente `WindowListener`, assim o fará a classe derivada. Os métodos não reimplementados herdarão a implementação original, que simplesmente ignora os eventos. Além da classe `WindowAdapter`, outras classes adaptadoras estão presentes, por exemplo `MouseAdapter`, `MouseMotionAdapter` e `KeyAdapter`.

Uma vez que uma classe manipuladora de eventos (*handler*) tenha sido criada para responder aos eventos de um componente, é preciso associar esse componente à instância dessa classe. Para tal, cada tipo de componente gráfico oferece métodos na forma `addXXListener(XXListener l)` e `removeXXListener(XXListener l)`, em que `XX` está associado a algum evento do tipo `XXEvent`. Por exemplo, para o componente `Window` (e por herança para todas os componentes derivados

dele), são definidos os métodos **addWindowListener(WindowListener l)** e **removeWindowListener(WindowListener l)**, uma vez que nesse tipo de componente é possível ocorrer um **WindowEvent**.

A Figura 6.10 apresenta um exemplo de um *frame* simples composto de dois botões e um campo texto. O *frame* foi associado a uma instância de uma classe manipuladora de eventos – classe **WindowHandler**, que é subclasse da classe adaptadora **java.awt.event.WindowAdapter**. Como pode ser observado pelo Código 6.1, o único evento tratado pela classe **WindowAdapter** é o clique no botão de fechar da barra de título do *frame*.

```
public class EventsDemo {

    public static void main(String[] args) {
        Frame f = new Frame("EventsDemo");
        f.setLayout(new FlowLayout());
        f.addWindowListener(new WindowHandler());
        Button b1 = new Button("-");
        Button b2 = new Button("+");
        TextField tValue = new TextField("0");
        tValue.setEditable(false);
        ButtonHandler bh = new ButtonHandler(tValue);
        b1.addActionListener(bh);
        b2.addActionListener(bh);
        f.add(b1);
        f.add(tValue);
        f.add(b2);
        f.setSize(100, 70);
        f.setVisible(true);
    }
}
```

Figura 6.10 Tratamento de eventos **AWT**.

O campo de texto foi configurado de tal forma que não pode ser editado pelo usuário (método **setEditable(false)**). As alterações no conteúdo do campo de texto apenas acontecem em resposta a eventos (cliques) ocorridos nos dois botões. Os dois botões são associados ao mesmo objeto – instância da classe **ButtonHandler** (Código 6.1) que implementa a interface **ActionListener**.

Note que a classe **ButtonHandler** possui uma referência ao campo texto (atributo **field**), de tal forma que:

- a cada clique no botão **b1 (-)** ou no botão **b2 (+)**, o conteúdo do campo texto é decrementado ou incrementado em 1;
- é necessária a conversão de *string*/inteiro/*string* para realizar as operações de decremento/incremento, visto que o campo texto armazena uma *string*.

Uma discussão extensiva dos eventos **AWT** encontra-se fora do escopo deste material. Para um estudo mais aprofundado desse tópico, o leitor interessado pode consultar o seguinte link: http://docs.oracle.com/javase/tutorial/uiswing/events/api.html.

```java
class WindowHandler extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}

class ButtonHandler implements ActionListener {
    private TextField field;

    public ButtonHandler(TextField field) {
        this.field = field;
    }

    public void actionPerformed(ActionEvent e) {
        String command = e.getActionCommand();
        Integer valor = new Integer(field.getText());
        if (command.equals("+"))
            valor = valor + 1;
        else
            valor = valor - 1;
        field.setText(valor.toString());
    }
}
```

Código 6.1 Classes manipuladoras de eventos.

6.4 Swing

Quando a versão 1.0 da linguagem Java foi lançada, apenas continha a biblioteca `AWT` para a programação de interfaces gráficas. Conforme dito, a grande desvantagem da utilização da biblioteca `AWT` é que componentes gráficos têm diferentes aparências (*look-and-feel*) se executados em diferentes sistemas operacionais.

A biblioteca `Swing` é definida através das classes do pacote `javax.swing` e consiste em uma extensão padronizada da biblioteca `AWT` que congrega componentes gráficos que utilizam exclusivamente Java (*lightweight components*), com funcionalidades e aparência independentes do sistema em que a aplicação é executada.

A biblioteca `Swing` é compatível com o `AWT`, mas trabalha de uma maneira totalmente diferente. A biblioteca `Swing` procura renderizar/desenhar por conta própria todos os componentes, ao invés de delegar essa tarefa ao sistema operacional. Por ser uma biblioteca de mais alto nível, ou seja, mais abstração, menor aproximação das *APIs* do sistema operacional, ela tem bem menos performance que outras *APIs* gráficas (por exemplo, `AWT`) e consome mais memória **RAM** em geral. Porém, ela é bem mais completa, e os programas que usam a biblioteca `Swing` têm uma aparência muito parecida, independentemente do sistema operacional utilizado.

Uma discussão extensiva da biblioteca `Swing` encontra-se fora do escopo deste material. Esta seção apenas apresenta uma visão geral das funcionalidades providas pela biblioteca `Swing` com o objetivo de ressaltar as diferenças e

semelhanças com a biblioteca **AWT** discutida anteriormente. Para um estudo mais aprofundado da biblioteca **Swing**, o leitor interessado pode consultar o seguinte link: http://docs.oracle.com/javase/tutorial/uiswing.

6.4.1 Componentes gráficos

De maneira análoga à biblioteca **AWT**, na biblioteca **Swing** um objeto de interface como um botão ou uma barra de rolagem é chamado de componente. A classe **JComponent** é a raiz de todos os componentes **Swing**. A Figura 6.11 apresenta a hierarquia de componentes gráficos presentes na biblioteca **Swing** (pacote **javax.swing**). Conforme se pode observar por essa figura, a classe **JComponent** é derivada da classe **Component**, raiz da hierarquia de componentes **AWT**.

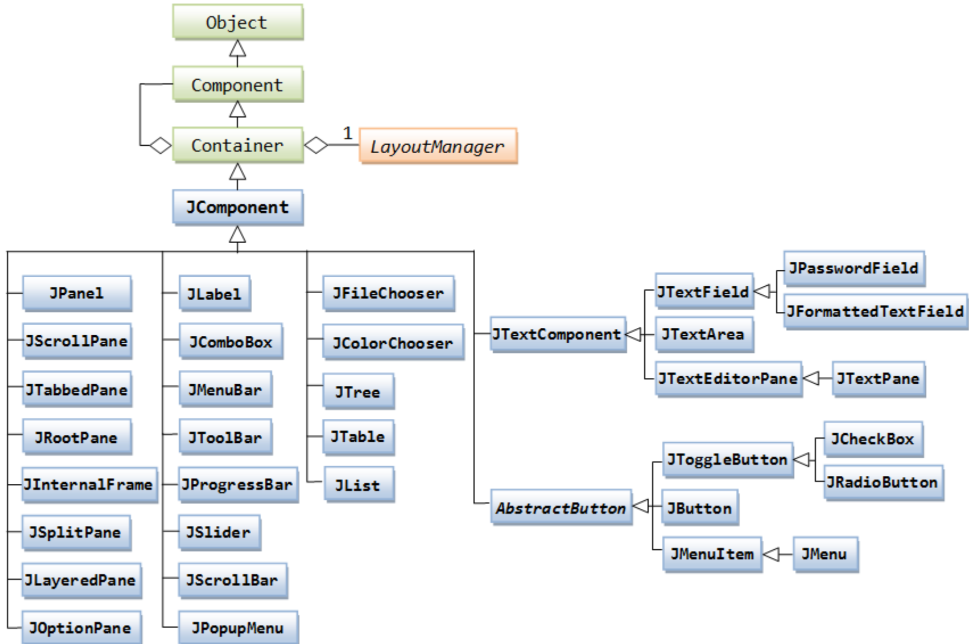


Figura 6.11 Hierarquia de componentes gráficos presentes na biblioteca **Swing**.

Embora a classe **JComponent**, raiz da hierarquia de componentes **Swing**, seja derivada da classe **java.awt.Container**, não se pode acrescentar diretamente um componente gráfico a qualquer componente **Swing**. Para as classes **Swing** que correspondem a *containers* no sentido definido pela biblioteca **AWT**, ou seja, às quais podem ser acrescentados outros componentes, deve-se obter uma referência ao objeto **Container** através do método **getContentPane()**.

Eventos. Os eventos em componentes **Swing** são os mesmos da biblioteca **AWT**, presentes no pacote **java.awt.event**. Ou seja, o modelo de tratamento de eventos da biblioteca **AWT** se aplica também a componentes **Swing**.

Janelas. A biblioteca **Swing** também oferece a sua versão de um *frame* através da classe **JFrame**, que é uma extensão da classe **Frame**. As Figuras 6.12 e 6.13 apresentam dois exemplos simples de *frames* **Swing** que contêm componentes gráficos.

No *frame* do **AWT**, quando o usuário clica no botão fechar, nada acontece até que o evento de fechar a janela seja tratado. No **Swing**, por padrão, a janela é escondida quando o usuário clica no botão de fechar. Observe que a janela será escondida, mas a aplicação Java não é finalizada.

O projetista pode determinar outras opções para quando o usuário clicar no botão de fechar – determinadas pelo método **setDefaultCloseOperation()** do **JFrame**. Os argumentos para esse método podem ser:

- **DO_NOTHING_ON_CLOSE** – não faz nada quando o usuário clica no botão de fechar. Nesse caso, o programa poderia usar uma classe que implementa a interface **WindowListener** que execute uma outra ação em seu método **windowClosing()**.
- **HIDE_ON_CLOSE** (o padrão) – esconde o *frame*. Remove-o da tela.
- **DISPOSE_ON_CLOSE** – faz a mesma coisa que o método **dispose()**. Remove o *frame* da tela e libera os recursos usados.

Gerenciador de leiautes. É possível também determinar um gerenciador de leiaute para um *frame* **Swing**. Esse gerenciador de leiaute deve ser aplicado ao objeto **Container**, obtido através do método **getContentPane()**, assim como os componentes (botões, rótulo, etc.) também devem ser adicionados ao **ContentPane** do **JFrame**. Os gerenciadores de leiaute definidos na biblioteca **AWT** podem ser também aplicados a componentes **Swing**. As Figuras 6.14 e 6.15 apresentam dois exemplos simples de *frames* **Swing** que contêm um gerenciador de leiaute associado.

Botões. Em **Swing**, o componente que define um botão é **JButton**. Além de poder ser definido com rótulos de texto, um botão **Swing** pode conter também ícones. A Figura 6.12 apresenta um exemplo de dois botões **Swing** com ícones.

```java
import java.awt.FlowLayout;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;

public class JButtonDemo {

    public static void main(String[] args) {
        JFrame frame = new JFrame("JButtonDemo");
        frame.setResizable(true);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        JButton b1 = new JButton("Brasil", new ImageIcon("Brasil.gif"));
        JButton b2 = new JButton("França", new ImageIcon("Franca.gif"));
        frame.setLayout(new FlowLayout());
        frame.getContentPane().add(b1);
        frame.getContentPane().add(b2);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Figura 6.12 Componentes gráficos **Swing** – classe **JButton**.

Campos de Seleção. A classe **JComboBox** permite a seleção de uma opção entre as opções exibidas. É semelhante à classe **List**, da biblioteca **AWT**, discutida anteriormente. A Figura 6.13 apresenta um exemplo de um *combo box* **Swing**.

```java
package br.ufscar.si.poo2.swing;

import javax.swing.JComboBox;
import javax.swing.JFrame;

public class JComboBoxDemo {

    public static void main(String[] args) {
        JFrame frame = new JFrame("JComboBoxDemo");
        String[] languages = {"C++", "C#", "Java", "Smalltalk"};
        JComboBox list = new JComboBox(languages);
        frame.getContentPane().add(list);
        frame.setSize(200, 50);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.setResizable(true);
        frame.setVisible(true);
    }
}
```

Figura 6.13 Componentes gráficos **Swing** – classe **JComboBox**.

A classe **JCheckBox** funciona de maneira análoga à classe **Checkbox** do **AWT**. Ou seja, permite a escolha de um ou mais itens a partir de um conjunto de elementos exibidos. A Figura 6.14 apresenta um exemplo de um *check box* **Swing**.

```java
package br.ufscar.si.poo2.swing;

import java.awt.FlowLayout;
import javax.swing.JCheckBox;
import javax.swing.JFrame;

public class JCheckboxDemo {

    public static void main(String[] args) {
        JFrame frame = new JFrame("JCheckboxDemo");
        frame.getContentPane().setLayout(new FlowLayout());
        JCheckBox box1 = new JCheckBox("Azul");
        JCheckBox box2 = new JCheckBox("Verde");
        JCheckBox box3 = new JCheckBox("Vermelho");
        frame.getContentPane().add(box1);
        frame.getContentPane().add(box2);
        frame.getContentPane().add(box3);
        frame.setSize(270, 60);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Figura 6.14 Componentes gráficos **Swing** – classe **JCheckBox**.

Conforme se pode observar, a Figura 6.14 é bastante similar à Figura 6.4. A interface é a mesma, o que diferencia mesmo são os componentes gráficos utilizados (**JFrame**, **JCheckBox**, etc.).

A classe **JRadioButton** implementa *radio buttons*. A utilização da classe **JRadioButton**, associada à classe **ButtonGroup**, permite que apenas uma das opções exibidas seja escolhida, enquanto o **JCheckBox** permite múltipla escolha. A Figura 6.15 apresenta um exemplo de um *radio button* **Swing**.

```
public class JRadionButtonDemo {

    public static void main(String[] args) {
        JFrame frame = new JFrame("JRadioButtonDemo");
        frame.getContentPane().setLayout(new FlowLayout());
        frame.setSize(270, 60);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        JRadioButton b1 = new JRadioButton("Linux");
        JRadioButton b2 = new JRadioButton("Mac");
        JRadioButton b3 = new JRadioButton("Windows");
        ButtonGroup bg = new ButtonGroup();
        bg.add(b1);
        bg.add(b2);
        bg.add(b3);
        frame.getContentPane().add(b1);
        frame.getContentPane().add(b2);
        frame.getContentPane().add(b3);
        frame.setVisible(true);
    }
}
```

JRadioButtonDemo
Linux Mac Windows

Figura 6.15 Componentes gráficos **Swing** – classe **JRadioButton**.

Campos de Texto. O **JTextField** é uma área de edição de texto de uma única linha. Esse componente é equivalente ao componente **TextField** da biblioteca **AWT**. Quando o usuário digitar todo o texto no **JTextField**, ele pode clicar na tecla **return/enter** (que acionará um evento do tipo **ActionEvent**).

O **JTextArea** é uma área de edição de texto de múltiplas linhas. Esse componente é equivalente ao componente **TextArea** da biblioteca AWT. Tendo em vista que no **JTextArea** a tecla **return/enter** insere uma nova linha, é comum utilizar um **JButton** para capturar eventos associados a um **JTextArea**.

Para criação de rótulos, a biblioteca **Swing** disponibiliza a classe **JLabel**. O **JLabel** permite a apresentação de conteúdo puramente textual, assim como o **Label** da biblioteca **AWT**, e também imagens.

A Figura 6.16 apresenta um exemplo de um *frame* simples composto de dois rótulos (objetos **JLabel**), uma área de texto (objeto **JTextArea**) e um campo de texto (objeto **JTextField**). O campo de texto foi configurado de tal forma que não pode ser editado pelo usuário (método **setEditable(false)**). A área de texto foi associada a uma instância de uma classe manipuladora de eventos – classe **KeyHandler** que é subclasse da classe adaptadora **java.awt.event.KeyAdapter**. Como pode ser

```java
public class JTextDemo {

    public static void main(String[] args) {
        JFrame frame = new JFrame("JTextoDemo");
        frame.setLayout(new GridLayout(2, 2));
        JLabel textoLbl = new JLabel("Texto:");
        JTextArea texto = new JTextArea();
        JLabel numLbl = new JLabel("Caracteres:");
        JTextField numCaracteres = new JTextField("0");
        numCaracteres.setEditable(false);
        texto.addKeyListener(new KeyHandler(numCaracteres));
        frame.getContentPane().add(textoLbl);
        frame.getContentPane().add(texto);
        frame.getContentPane().add(numLbl);
        frame.getContentPane().add(numCaracteres);
        frame.setSize(200, 100);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.setVisible(true);
    }

}
```

JTextoDemo
Texto: Programação Orientada a Objetos 2
Caracteres: 33

Figura 6.16 Componentes gráficos `Swing` – campos de texto.

observado pelo Código 6.2, essa classe possui uma referência ao campo texto (atributo `field`) e trata os eventos do teclado (interface `KeyListener`) de tal forma que, a cada tecla digitada, o número de caracteres presente na área de texto (`texto`) é contabilizado, e esse valor é apresentado no campo de texto (`numCaracteres`).

```java
/**
 * Classe KeyHandler – Trata os eventos do teclado (interface KeyListener).
 * A cada tecla digitada, o número de caracteres presente na área de texto é contabilizado
 *
 * @author Delano Medeiros Beder
 */
public class KeyHandler extends KeyAdapter {

    private JTextField field;

    public KeyHandler(JTextField field) {
        this.field = field;
    }

    public void keyReleased(KeyEvent e) {
        JTextArea area = ((JTextArea) e.getComponent());
        field.setText(Integer.toString(area.getText().length()));
    }
}
```

Código 6.2 Classe *handler* de eventos do teclado.

Caixas de Diálogo. Na biblioteca `Swing`, as caixas de diálogo são criadas a partir da classe `JOptionPane` ou da classe `JDialog`. A classe `JOptionPane` é recomendada quando se deseja criar caixas de diálogos mais simples, enquanto a classe `JDialog` para quando se deseja criar uma caixa de diálogo mais sofisticada, com todas as funcionalidades de um *frame*. A classe `JFileChooser` fornece uma janela para navegação pelo sistema de arquivos. Instâncias dessa classe permitem escolher um arquivo para abrir ou especificar o diretório em que um arquivo será salvo. A classe `JFileDialog` equivale à classe `FileDialog` da biblioteca `AWT`.

Menus. O menu em `Swing` funciona de maneira análoga ao menu da biblioteca `AWT`. Os passos são semelhantes ao realizados na Seção 6.3 para construir um menu `AWT`.

(a) Criação de uma barra de menu, representada pela classe **`JMenuBar`**, e a posterior adição dessa barra de menu ao *frame* `Swing`.

(b) Criação de instâncias da classe **`JMenu`** e a posterior adição à barra de menu criada anteriormente. No exemplo apresentado na Figura 6.17, foram criados dois menus: **Arquivo** e **Ajuda**.

(c) Criação dos itens de menu, representados pela classe **`JMenuItem`**, para os menus (instâncias de **`JMenu`**) criados anteriormente. Como se pode observar, é possível adicionar teclas de atalho (por exemplo, **`CTRL+A`**) aos itens de menu e criar itens de menu para seleção *on/off* (instâncias da classe **`JCheckboxMenuItem`**). E, por fim, foi utilizado um separador (método **`addSeparator()`**) para dividir os itens.

```
public class JMenuDemo {

    public static void main(String[] args) {
        JFrame frame = new JFrame("JMenuDemo");
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        JMenuBar bar = new JMenuBar();
        frame.setJMenuBar(bar);
        JMenu menuFile = new JMenu("Arquivo");
        JMenuItem open = new JMenuItem("Abrir");
        open.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_A,
                InputEvent.CTRL_MASK));
        JMenuItem save = new JCheckBoxMenuItem("Salvar");
        JMenuItem exit = new JMenuItem("Sair");
        menuFile.add(open);
        menuFile.add(save);
        menuFile.addSeparator();
        menuFile.add(exit);
        JMenu menuHelp = new JMenu("Ajuda");
        bar.add(menuFile);
        bar.add(menuHelp);
        frame.setSize(200, 100);
        frame.setVisible(true);
    }
}
```

Figura 6.17 Componentes gráficos `Swing` – menus.

6.4.2 Editor de texto

Finalizando a discussão sobre as bibliotecas `Swing` e `AWT`, esta seção apresenta um editor de texto simples implementado utilizando algumas das classes presentes nessas bibliotecas. A implementação Java desse editor de texto é composta de três classes: **`Editor`**, **`FileUtil`** e **`Handler`**.

O Código 6.3 apresenta a classe **`Editor`**, que é um *container* onde os componentes gráficos `Swing` (menu, barra de rolagem, área de texto, etc.) são inseridos.

O método **configure()**, invocado pelo construtor da classe, é responsável por realizar as configurações do editor de texto, assim como inserir os componentes gráficos no *frame*.

Nas linhas 21-24 são criados um painel com barras de rolagem vertical e horizontal (instância da classe **JPaneScroll**) e uma área de texto (instância da classe **JTextArea**) que apresenta o texto editado. Observe que a área de texto é associada a uma instância da classe **Handler**, que é responsável por tratar os eventos de teclado nessa área de texto.

```
1  /**
2   * Classe Editor - Interface gráfica. Container os demais componentes gráficos são inseridos
3   *
4   * @author Delano Medeiros Beder
5   */
6  public class Editor extends JFrame {
7      public static final String TITLE = "Editor de Texto Simples";
8      private JTextArea textArea;
9      private Handler handler;
10
11     public Editor() {
12         handler = new Handler(this);
13         configure();
14     }
15
16     public JTextArea getTextArea() {
17         return textArea;
18     }
19
20     private void configure() {
21         JScrollPane jScrollPane = new JScrollPane();
22         jScrollPane.setViewportView(textArea);
23         textArea = new JTextArea();
24         textArea.addKeyListener(handler);
25         JMenuItem open = new JMenuItem("Abrir");
26         open.addActionListener(handler);
27         JMenuItem save = new JMenuItem("Salvar");
28         save.addActionListener(handler);
29         JMenuItem exit = new JMenuItem("Sair");
30         exit.addActionListener(handler);
31         JMenu menuFile = new JMenu("Arquivo");
32         menuFile.add(open);
33         menuFile.add(save);
34         menuFile.addSeparator();
35         menuFile.add(exit);
36         JMenuBar jMenuBar = new JMenuBar();
37         jMenuBar.add(menuFile);
38         this.setJMenuBar(jMenuBar);
39         JPanel contentPane = ((JPanel) this.getContentPane());
40         contentPane.setLayout(new BorderLayout());
41         contentPane.add(jScrollPane, BorderLayout.CENTER);
42         this.setSize(480, 284);
43         this.setTitle(TITLE);
44         this.setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
45         this.addWindowListener(handler);
46     }
47
48     public static void main(String[] args) {
49         Editor e = new Editor();
50         e.setVisible(true);
51     }
52 }
```

Código 6.3 Classe **Editor**.

Nas linhas 25-38, o menu de opções é criado e adicionado ao editor de texto. Observe que os itens de menu são associados a uma instância da classe **Handler**, que é responsável por tratar os eventos de cliques nos itens de menu. Por fim, nas linhas 39-45, são realizadas as últimas configurações (leiaute, tamanho, etc.). A Figura 6.18 apresenta o leiaute final do editor de texto.

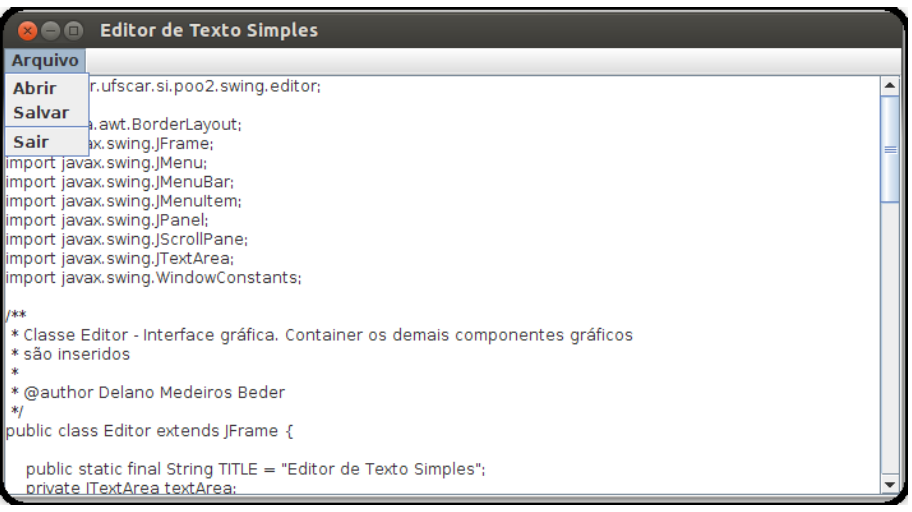


Figura 6.18 Aplicação gráfica **Swing** – editor de texto.

O Código 6.4 apresenta a classe **FileUtil** responsável pelas tarefas de carregar e salvar arquivos. Observe que os métodos **save()** e **load()** utilizam as classes **FileWriter**, **BufferedWriter**, **FileReader** e **BufferedReader**, discutidas na Unidade 4 e que proporcionam operações de entrada e saída *bufferizadas*.

O método **save()** recebe como argumentos um nome de arquivo e uma *string* (no caso, o conteúdo da área de texto presente na interface gráfica) e salva esse conteúdo no arquivo.

O método **load()** recebe como argumento um nome de arquivo e devolve uma *string* com todo o conteúdo presente no arquivo. Note a utilização da classe **StringBuilder** (Seção 3.4.1.1), pois essa é mais eficiente na concatenação de *strings*.

Por questões de simplicidade, o tratamento das exceções relacionadas às operações de entrada e saída é bastante simplista. Em aplicações mais robustas, o tratamento de exceções deveria ser mais sofisticado.

```java
/**
 * Classe FileUtil - Responsável pelas tarefas de carregar e salvar arquivos.
 *
 * @author Delano Medeiros Beder
 */
public class FileUtil {

    public void save(String fileName, String texto) {
        try {
            FileWriter fw = new FileWriter(fileName);
            BufferedWriter bw = new BufferedWriter(fw);
            bw.write(texto);
            bw.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public String load(String fileName) {
        StringBuilder texto = new StringBuilder();
        try {
            FileReader fr = new FileReader(fileName);
            BufferedReader br = new BufferedReader(fr);
            String s = br.readLine();
            while (s != null) {
                texto.append(s).append("\n");
                s = br.readLine();
            }
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return texto.toString();
    }
}
```

Código 6.4 Classe `FileUtil`.

O Código 6.5 apresenta a classe **`Handler`**. Por questões de simplicidade, nessa aplicação-exemplo foi implementada apenas a classe **`Handler`**, que trata todos os eventos possíveis na aplicação. Porém, em sistemas mais robustos, é recomendável a implementação de diferentes classes manipuladoras de eventos (*handlers*) – cada uma especializada em um tipo de evento.

O método **`actionPerformed()`** da interface **`ActionListener`** é responsável por tratar os eventos de cliques nos itens de menu. Esse método tem o seguinte comportamento:

- Se o item de menu **Abrir** for clicado, execute o método **`load()`** que é responsável por carregar o conteúdo de um arquivo. Observe que uma instância da classe **`JFileDialog`** é utilizada para o usuário escolher o nome do arquivo a ser carregado e posteriormente salvo.
- Se o item de menu **Salvar** for clicado, execute o método **`save()`** que é responsável por salvar o conteúdo da área de texto em um arquivo. As tarefas de carregar e salvar arquivos são delegadas para uma instância da classe **`FileUtil`**.

```java
/**
 * Classe Handler - Trata os eventos da interface gráfica
 *
 * @author Delano Medeiros Beder
 */
public class Handler extends WindowAdapter implements ActionListener, KeyListener {
    private Editor editor;
    private JFileChooser jFileChooser;
    private boolean changed;
    private FileUtil util;

    public Handler(Editor editor) {
        this.editor = editor;
        this.jFileChooser = new JFileChooser();
        this.jFileChooser.setMultiSelectionEnabled(false);
        this.changed = false;
        util = new FileUtil();
    }

    public void actionPerformed(ActionEvent e) {
        String command = e.getActionCommand();
        switch (command) {
            case "Abrir":  { load(); break; }
            case "Salvar": { save(); break; }
            case "Sair":   { exit(); break; }
        }
    }

    public void keyPressed(KeyEvent e) { // Não faz nada }

    public void keyReleased(KeyEvent e) {
        if (!changed) {
            editor.setTitle(Editor.TITLE + "*");
            changed = true;
        }
    }

    public void keyTyped(KeyEvent e) { // Não faz nada }

    public void windowClosing(WindowEvent e) { exit(); }

    private void load() {
        int state = jFileChooser.showOpenDialog(editor);
        if (state == JFileChooser.APPROVE_OPTION) {
            String fileName = jFileChooser.getSelectedFile().getAbsolutePath();
            editor.getTextArea().setText(util.load(fileName));
            editor.setTitle(Editor.TITLE);
            changed = false;
        }
    }

    private void save() {
        int state = jFileChooser.showSaveDialog(editor);
        if (state == JFileChooser.APPROVE_OPTION) {
            String fileName = jFileChooser.getSelectedFile().getAbsolutePath();
            util.save(fileName, editor.getTextArea().getText());
            editor.setTitle(Editor.TITLE);
            changed = false;
        }
    }

    private void exit() {
        String msg = "O conteúdo foi modificado. Quer salvar antes de sair?";
        if (changed) {
            int state = JOptionPane.showConfirmDialog(editor, msg);
            if (state == JOptionPane.YES_OPTION) {
                save();
            } else if (state == JOptionPane.CANCEL_OPTION) {
                return;
            }
        }
        System.exit(0);
    }
}
```

Código 6.5 Classe `Handler`.

- Se o item de menu **Sair** for clicado, execute o método `exit()` responsável por sair da aplicação. Observe que, antes de sair, a aplicação verifica se o usuário deseja salvar, caso esteja alterado, o conteúdo presente na área de texto.

O método `KeyReleased()` da interface `KeyListener` é responsável por tratar o evento que ocorre quando uma tecla é liberada, após ter sido pressionada, na área de texto. Esse evento é apenas tratado para atualizar a variável `changed`, que indica se o conteúdo foi alterado ou não. Observe que os demais eventos da interface `KeyListener` são ignorados.

Por fim, o método `windowClosing()` da interface `WindowListener` é responsável por tratar o evento de clique no botão de fechar a janela. O comportamento desse método é invocar o método `exit()`, responsável por sair da aplicação.

6.5 Considerações finais

Esta unidade apresentou dois pacotes que fornecem um conjunto de componentes gráficos incorporados à *API* padrão da linguagem Java. As bibliotecas `AWT` e `Swing` fazem parte do Java Foundation Classes (JFC), uma biblioteca mais vasta que inclui também a *API* Java 2D™, entre outras.

A *Standard Widget Toolkit* (`SWT`) é uma biblioteca para uso com a plataforma Java. Foi originalmente desenvolvida pela IBM e hoje é mantida pela Eclipse Foundation em conjunto com o IDE Eclipse. É uma alternativa para as bibliotecas `AWT` e `Swing` discutidas nesta unidade.

A `SWT` (NORTHOVER; WILSON, 2004) é escrita em Java. Para exibir elementos gráficos, a implementação do `SWT` acessa as bibliotecas gráficas nativas do sistema operacional usando o Java Native Interface (JNI) de uma forma similar à `AWT`, que usa *APIs* específicas de um sistema operacional. Programas que usam o `SWT`, assim como o `Swing`, são portáveis, mas a implementação da biblioteca, apesar de parte dela ser escrita em Java, é única para cada sistema. Ou seja, existe uma implementação da biblioteca `SWT` para cada sistema operacional.

A discussão da biblioteca `SWT` encontra-se fora do escopo deste livro. No entanto, o leitor interessado pode consultar a documentação oficial dessa biblioteca através do seguinte link: http://www.eclipse.org/swt/.

Finalizando a discussão desta unidade, é importante observar que a linguagem Java não exige a utilização de um IDE. Todos os comandos necessários ao desenvolvimento poderiam ser feitos em um terminal de comando. No entanto, assim como em outras linguagens de programação, o IDE torna ágil o processo de desenvolvimento ao integrar diferentes funcionalidades (edição, compilação, execução, etc.) e abstrair a sintaxe dos comandos necessários relacionados a essas atividades. Em relação ao desenvolvimento de interfaces gráficas com usuários, essa atividade é bastante facilitada se o desenvolvedor utiliza um IDE que provê suporte pleno. Dessa forma, este material sugere e indica alguns links que podem auxiliar na tarefa de configurar o IDE para que possa ser utilizada no desenvolvimento de interfaces gráficas em Java:

- Netbeans: https://netbeans.org/kb/trails/matisse_pt_BR.html.
- Eclipse: http://www.eclipse.org/windowbuilder/.

A próxima unidade apresenta os conceitos relacionados às operações de acesso e manipulação de banco de dados na linguagem de programação Java.

6.6 Estudos complementares

Para estudos complementares sobre os tópicos abordados nesta unidade, o leitor interessado pode consultar as seguintes referências:

ARNOLD, K.; GOSLING, J.; HOLMES, D. *The Java Programming Language*. 4. ed. Boston: Addison-Wesley, 2005.

DEITEL, P.; DEITEL, H. *Java*: Como programar. 8. ed. São Paulo: Pearson Brasil, 2010.

NETBEANS. *Trilha de Aprendizado das Aplicações de GUI do Java*. 2014. Disponível em: https://netbeans.org/kb/trails/matisse_pt_BR.html. Acesso em: 12 ago. 2014.

ORACLE. *The Java™ Tutorials* – Trail: Creating a GUI With JFC/Swing. 2014. Disponível em: http://docs.oracle.com/javase/tutorial/uiswing. Acesso em: 12 ago. 2014.

UNIDADE 7

Acesso a banco de dados em Java

7.1 Primeiras palavras

Muitos sistemas necessitam manter as informações com as quais trabalham, seja para permitir consultas ou possíveis alterações futuras nas informações. Para que esses dados sejam mantidos de forma persistente, esses sistemas geralmente guardam tais informações em um sistema de banco de dados, que as mantém de forma organizada e prontas para consultas.

Um sistema de banco de dados (ELMASRI; NAVATHE, 2005) é constituído por uma coleção organizada de dados (a base de dados) e pelo software que coordena o acesso a esses dados (o sistema gerenciador de banco de dados, ou **SGBD**). Utilizar um sistema de banco de dados ao invés de simples arquivos (Unidade 4) para armazenar de forma persistente os dados de uma aplicação apresenta diversas vantagens, das quais se destacam:

- O desenvolvedor da aplicação não precisa se preocupar com os detalhes do armazenamento, trabalhando com especificações mais abstratas para a definição e manipulação dos dados;
- Tipicamente, um **SGBD** incorpora mecanismos para controle de acesso concorrente por múltiplos usuários e controle de transações.

Nesse contexto, esta unidade apresenta os conceitos relacionados às operações de acesso a um sistema de banco de dados relacional na linguagem de programação Java.

7.2 Problematizando o tema

Ao final desta unidade, espera-se que o leitor seja capaz de reconhecer e definir precisamente os conceitos relacionados às operações de acesso a um sistema de banco de dados relacional na linguagem de programação Java. Dessa forma, esta unidade pretende discutir as seguintes questões:

- Quais as principais características do modelo de dados relacional ?
- Quais são os passos necessários para conectar-se a um sistema de banco de dados relacional utilizando a *API* `JDBC` ?
- Como são realizadas as consultas e atualizações de informações armazenadas em um sistema de banco de dados relacional utilizando a *API* `JDBC` ?

- Quais as vantagens e desvantagens da utilização do padrão de projeto *DAO* (*Data Access Object*) no acesso a um sistema de banco de dados ?

7.3 Modelo de dados relacional

O modelo de dados relacional, o modelo de dados mais utilizado na atualidade, foi introduzido em 1970 (CODD, 1970) e imediatamente atraiu a atenção em virtude de sua simplicidade e base matemática.

Um banco de dados relacional organiza seus dados em relações (tabelas). Cada relação pode ser vista como uma tabela, em que cada coluna representa os atributos da relação e as linhas representam as tuplas ou elementos da relação. Quando uma relação é pensada como uma tabela de valores, cada linha na tabela representa uma coleção de valores de dados relacionados. Ou seja, no modelo de dados relacional, cada linha na tabela representa um fato que corresponde a uma entidade ou relacionamento no mundo real. O nome da tabela e os nomes das colunas são utilizados para ajudar na interpretação dos valores em cada linha. A tabela (relação) `ObraDeArte`, que será discutida na Seção 7.4, representa as pinturas e esculturas do sistema de gerenciamento de obras de artes (Seção 3.7). A Figura 7.1 apresenta quatro exemplos de tuplas dessa tabela.

ID	TITULO	ARTISTA	MATERIAL	ANO	CATEGORIA	TIPO	ALTURA
1	Mona Lisa	Leonardo da Vinci	Madeira	1503	1	Óleo	<NULL>
2	David	Michelangelo	Mármore	1501	2	<NULL>	4.1
3	O Lavrador de Café	Cândido Portinari	Madeira	1939	1	Óleo	<NULL>
4	O Pensador	Auguste Rodin	Bronze	1909	2	<NULL>	1.4

Figura 7.1 Tuplas de um banco de dados relacional.

Um conceito importante em um banco de dados relacional é o conceito de atributo chave (chave primária), que permite identificar e diferenciar uma tupla de outra. Através do uso de chaves é possível acelerar o acesso a elementos (usando índices) e estabelecer relacionamentos entre as múltiplas tabelas de um sistema de banco de dados relacional.

Essa visão de dados organizados em tabelas oferece um conceito simples e familiar para a estruturação dos dados, sendo um dos motivos do sucesso de sistemas relacionais de dados. Certamente, outros motivos para esse sucesso incluem o forte embasamento matemático por trás dos conceitos utilizados em bancos de dados

relacionais e a uniformização na linguagem de manipulação de sistemas de bancos de dados relacionais através da linguagem `SQL` (Seção 7.3.1).

Sob o ponto de vista matemático, uma relação é o subconjunto do produto cartesiano dos domínios da relação. Sendo um conjunto, é possível realizar operações de conjuntos – tais como união, interseção e diferença – envolvendo duas relações de mesma estrutura.

No entanto, um dos pontos mais fortes do modelo relacional está nos mecanismos de manipulação estabelecidos pela **Álgebra Relacional**. Os três principais operadores da álgebra relacional são **seleção**, **projeção** e **junção** (ELMASRI; NAVATHE, 2005).

A operação de **seleção** tem como argumento uma relação e uma condição (um predicado) envolvendo atributos da relação e/ou valores. O resultado é outra relação contemplando apenas as tuplas para as quais a condição foi verdadeira.

A operação de **projeção** tem como argumento uma relação e uma lista com um subconjunto dos atributos da relação. O resultado é outra relação contendo todas as tuplas da relação, mas apenas com os atributos especificados.

Observe que, se a lista de atributos não englobar a chave da relação, o resultado dessa operação poderia gerar tuplas iguais. Sob o ponto de vista estritamente matemático, os elementos duplicados devem ser eliminados, pois não fazem sentido para um conjunto.

A operação de **junção** recebe como argumentos duas relações e uma condição (um predicado) envolvendo atributos das duas relações. O resultado é uma relação com os atributos das duas relações contendo as tuplas que satisfizeram o predicado especificado. A operação de junção não é uma operação primitiva, pois pode ser expressa em termos da operação de produto cartesiano e da seleção, mas é uma das operações mais poderosas da álgebra relacional.

A forma mais usual de junção é aquela na qual a condição de junção é a igualdade entre valores de dois atributos das relações argumentos. Essa forma é tão usual que recebe o nome de junção natural. Nesse caso, o atributo comum aparece apenas uma vez na relação resultado, já que ele teria, para todas as tuplas, o mesmo valor nas duas colunas.

Uma discussão extensiva do modelo de dados relacional encontra-se fora do escopo deste material. Para um estudo mais aprofundado desse tópico, sugere-se que o leitor consulte a seguinte referência: Elmasri e Navathe (2005).

7.3.1 SQL

A *Structured Query Language* (**SQL**) é uma linguagem padronizada para a definição e manipulação de bancos de dados relacionais. Tipicamente, um **SGBD** oferece um interpretador **SQL** que permite isolar a aplicação dos detalhes de armazenamento dos dados. Se o projetista da aplicação tiver o cuidado de usar apenas as construções padronizadas de **SQL**, ele poderá desenvolver a aplicação sem se preocupar com o **SGBD** que será utilizado depois. Como exemplo de **SGBD**, pode-se citar o **PostgreSQL**[1], que é um software livre, mantido pela **PostgreSQL Global Development Group**, compatível com a linguagem **SQL**.

Os três componentes da linguagem **SQL** são:

1. Uma linguagem de definição de dados (**DDL** – *Data Definition Language*) para definir e revisar a estrutura de bancos de dados relacionais;

2. Uma linguagem de manipulação de dados (**DML** – *Data Manipulation Language*) para ler e escrever os dados; e

3. Uma linguagem de controle de dados (**DCL** – *Data Control Language*) para especificar mecanismos de segurança e integridade dos dados.

A **DDL** supre as facilidades para a criação e manipulação de esquemas relacionais. Uma das necessidades de uma aplicação que irá armazenar seus dados em um sistema de banco de dados relacional é como criar uma identidade para o conjunto de tabelas de sua aplicação. Esse mecanismo não é padronizado em **SQL**, podendo variar de fornecedor a fornecedor de sistema gerenciador de banco de dados (**SGBD**). Algumas possibilidades incluem:

```
CREATE SCHEMA name [AUTHORIZATION user | group]
ou
CREATE DATABASE name
```

Para criar uma tabela, o comando `CREATE TABLE` é utilizado:

```
CREATE TABLE name ( ATTRIBUTE DOMAIN [UNIQUE] [NULL | NOT NULL])
```

[1] http://www.postgresql.org/.

Alguns **SGBDs** adotam, ao invés da forma UNIQUE para indicação de chave da relação, um comando Primary Key após a criação dos atributos que comporão a chave primária da relação.

O campo ATTRIBUTE especifica o nome para a aplicação da coluna da tabela, enquanto DOMAIN especifica o tipo de dado para a coluna. Alguns tipos de dados usuais em **SQL** são INTEGER, REAL, VARCHAR, DATE, TIME e TIMESTAMP. Além desses comandos, ALTER TABLE permite modificar a estrutura de uma tabela existente, e DROP TABLE permite remover uma tabela.

O principal comando da **DML** é o comando SELECT. Por exemplo, para obter todos os dados de uma relação (tabela), utiliza-se a forma básica:

```
SELECT * FROM tabela
```

Através do mesmo comando, é possível especificar projeções e seleções de uma tabela:

```
SELECT colunas FROM tabela WHERE condição
```

A condição pode envolver comparações entre atributos e/ou valores constantes, podendo, para tanto, utilizar comparadores tais como igual (=), maior que (>), menor que (<), maior ou igual que (>=), menor ou igual que (<=), diferente (<>) e comparadores de strings (LIKE '*string*', em que '*string*' pode usar os caracteres '_' e '%' para indicar um caracter qualquer ou uma sequência qualquer de caracteres, respectivamente). As comparações podem ser combinadas através dos conectivos lógicos AND, OR e NOT.

É possível expressar as quatro operações aritméticas (+, -, *, /), tanto para a condição como para a especificação de recuperação dos dados. É possível também especificar a ordem desejada de apresentação dos dados, usando, para tal, a forma:

```
SELECT colunas FROM tabela WHERE condição ORDER BY coluna1
```

Uma alternativa a esse comando é SELECT GROUP BY, que ordena os dados e não apresenta elementos que tenham o mesmo valor para a cláusula de agrupamento.

```
SELECT colunas FROM tabela WHERE condição GROUP BY coluna1
```

Uma importante categoria de funções de SQL inclui as funções de agregação, que permitem computar a média (**AVG**), a quantidade (**COUNT**), o maior ou menor valor (**MAX** ou **MIN**) e o total (**SUM**) das expressões especificadas.

```
SELECT COUNT(Id) FROM Empregado /* Seleciona a quantidade de empregados */
SELECT MIN(Salário) FROM Empregado /* Seleciona o menor salário dos empregados */
SELECT MAX(Salário) FROM Empregado /* Seleciona o maior salário dos empregados */
SELECT SUM(Salário) FROM Empregado /* Calcula a soma dos salários dos empregados */
SELECT AVG(Salário) FROM Empregado /* Calcula a média dos salários dos empregados */
```

Através do comando `SELECT`, é possível especificar consultas envolvendo múltiplas tabelas, tais como:

```
SELECT Carro.* FROM Carro, Pessoa WHERE Carro.PessoaId = Pessoa.Id
```

Nesse caso, a junção das duas tabelas `Carro` e `Pessoa` é realizada tendo os atributos `Carro.PessoaId` e `Pessoa.Id` como atributos de junção.

É possível especificar consultas internas a uma consulta, como em

```
SELECT * FROM Empregado WHERE Salário > (SELECT AVG(Salário) FROM Empregado)
```

Nesse exemplo, apenas são selecionados os empregados que possuem o salário maior que a média salarial da empresa. É possível qualificar os resultados de uma consulta interna através das seguintes funções: **All** (a condição foi verdadeira para todos os elementos resultantes da subconsulta), **Any** (verdade para pelo menos um elemento), **Exists** (verdade se resultado da consulta tem pelo menos um elemento), **In** (verdade se o valor especificado faz parte do resultado), **Not Exists** (verdade se subconsulta teve resultado vazio) e **Not In** (verdade se valor especificado não faz parte do resultado da subconsulta).

A linguagem **SQL** oferece ainda mecanismos para inserir elementos em uma tabela:

```
INSERT INTO tabela (colunas) VALUES (valores)
```

Para modificar valores de colunas de um elemento:

```
UPDATE tabela SET coluna = valor [, coluna = valor]* WHERE condição
```

E para remover elementos de uma tabela:

```
DELETE FROM tabela WHERE condição
```

O Código 7.1 apresenta os comandos **SQL** responsáveis por criar o banco de dados `Museu` e a tabela `ObradeArte` do sistema de gerenciamento de obras de artes, a ser discutido na Seção 7.4. Esse código apresenta também alguns comandos de inserção.

```
CREATE SCHEMA Museu

CREATE TABLE ObraDeArte (
  ID INTEGER not null primary key,
  TITULO VARCHAR(50) not null, ARTISTA VARCHAR(50) not null,
  MATERIAL VARCHAR(30) not null, ANO INTEGER not null,
  CATEGORIA INTEGER not null, TIPO VARCHAR(30),
  ALTURA REAL
)

INSERT INTO ObraDeArte (ID, TITULO, ARTISTA, MATERIAL, ANO, CATEGORIA, TIPO)
VALUES (1, 'Mona Lisa', 'Leonardo da Vinci', 'Madeira', 1503, 1, 'Óleo')
INSERT INTO ObraDeArte (ID, TITULO, ARTISTA, MATERIAL, ANO, CATEGORIA, ALTURA)
VALUES (2, 'David', 'Michelangelo', 'Mármore', 1501, 2, 4.10)
INSERT INTO ObraDeArte (ID, TITULO, ARTISTA, MATERIAL, ANO, CATEGORIA, TIPO)
VALUES (3, 'O Lavrador de Café', 'Cândido Portinari', 'Madeira', 1939, 1, 'Óleo')
INSERT INTO ObraDeArte (ID, TITULO, ARTISTA, MATERIAL, ANO, CATEGORIA, ALTURA)
VALUES (4, 'O Pensador', 'Auguste Rodin', 'Bronze', 1909, 2, 1.40)
```

Código 7.1 Comandos SQL.

A **DCL** é utilizada para especificar mecanismos de segurança e integridade dos dados. Seus dois principais comandos são:

- `GRANT`, que autoriza um ou mais usuários a realizarem, uma operação ou um conjunto de operações em um objeto. O comando `GRANT` ilustrado abaixo autoriza dois usuários (**usuário1** e **usuário2**) a realizarem operações de seleção e atualização em todas as tabelas do banco de dados `Museu`;
- `REVOKE`, que remove uma autorização dada anteriormente pelo comando `GRANT` ou a autorização padrão provida pelo **SGBD**. O comando `REVOKE` ilustrado abaixo remove as autorizações dadas anteriormente aos dois usuários.

```
GRANT SELECT, UPDATE ON Museu TO usuário1, usuário2
REVOKE SELECT, UPDATE ON Museu FROM usuário1, usuário2
```

7.3.2 JDBC

A linguagem **SQL** pode ser utilizada diretamente pelo usuário, quando o **SGBD** oferece um interpretador **SQL** interativo ou através de comandos embutidos em uma aplicação desenvolvida em uma linguagem de programação. No caso da linguagem de programação Java, a forma de interagir com o banco de dados é especificada pela *API* `JDBC` (SPEEGLE, 2001), discutida nesta seção.

O *Java Database Connectivity* (`JDBC`) é uma *API* Java (pacote **`java.sql`**) para execução e manipulação de resultados de consultas **SQL**. Para uma aplicação Java acessar um banco de dados relacional é necessária a execução dos seguintes passos:

1. habilitar o *driver* `JDBC`;
2. estabelecer uma conexão com o banco de dados;
3. executar a consulta **SQL**; e
4. apresentar resultados da consulta.

7.3.2.1 *Drivers* `JDBC`

O *driver* `JDBC` consiste em um componente de software que permite que uma aplicação Java interaja com um banco de dados. Para conectar com banco de dados individuais, o `JDBC` requer *drivers* para cada **SGBD** (**PostgreSQL**, **MySQL**, etc.). O *driver* `JDBC` fornece a conexão ao banco de dados e implementa o protocolo para transferir a consulta e o resultado entre a aplicação-cliente Java e o **SGBD**.

Do ponto de vista da aplicação Java, um *driver* nada mais é do que uma classe cuja funcionalidade precisa ser disponibilizada para a aplicação. A funcionalidade básica que um *driver* deve oferecer é especificada através da interface **`Driver`**. A forma mais usual é carregar o *driver* explicitamente para a **JVM** através do método **`forName()`** da classe **`Class`**, como em

```
Class.forName("org.postgresql.Driver"); // Carrega o driver para o SGBD PostgreSQL
Class.forName("com.mysql.jdbc.Driver"); // Carrega o driver para o SGBD MySQL
Class.forName("org.apache.derby.jdbc.ClientDriver"); // Carrega o driver para o SGBD Apache Derby
```

Alternativamente, a classe **`DriverManager`** estabelece um conjunto básico de serviços para a manipulação de *drivers* `JDBC`. Como parte de sua inicialização, essa

classe tentará obter o valor da propriedade `jdbc.drivers` de um arquivo de definição de propriedades e carregar os *drivers* especificados pelos nomes das classes.

7.3.2.2 Conexão com banco de dados

Uma vez que o *driver* esteja carregado, a aplicação Java pode estabelecer uma conexão com o sistema gerenciador de banco de dados. Para especificar com qual banco de dados se deseja estabelecer a conexão, é utilizada uma *string* na forma de uma `URL` na qual o protocolo é `jdbc:` e o restante da *string* é dependente do *driver*. Por exemplo, a `URL` `jdbc:derby://localhost:1527/Museu`, especifica uma conexão ao banco de dados `Museu` que se encontra sob a gerência do **SGBD Apache Derby** hospedado na máquina local.

Identificado o banco de dados, a sessão, a ser estabelecida para o acesso ao banco de dados, será controlada por uma instância de uma classe que implementa a interface `Connection`. O `DriverManager` oferece o método `getConnection()` para executar essa tarefa. O encerramento de uma sessão é sinalizado pelo método `close()` da conexão.

```
String url = "jdbc:derby://localhost:1527/Museu"; // URL de conexão
String usuário = "root"; // usuário do SGBD
String senha = "root"; // senha do usuário do SGBD
...
Connection c = DriverManager.getConnection(url, usuário, senha);
...
c.close(); // encerra a conexão com o banco
```

7.3.2.3 Execução da consulta

Estabelecida a conexão ao banco de dados, é possível criar uma consulta **SQL** e executá-la a partir da aplicação Java. Para representar uma consulta **SQL**, o `JDBC` utiliza uma instância de uma classe que implementa a interface `Statement`. Um objeto dessa classe pode ser obtido através do método `createStatement()` da classe `Connection`.

Uma vez que uma instância de `Statement` esteja disponível, é possível aplicar a ele o método `executeQuery()`, que recebe como argumento uma *string* representando uma consulta **SQL**. O resultado da execução da consulta é disponibilizado através de um objeto `ResultSet`.

```
...
Connection c = DriverManager.getConnection(url, usuário, senha);
..
Statement s = c.createStatement();
String query = "SELECT * FROM Empregado";
ResultSet r = s.executeQuery(query);
...
s.close();
c.close(); // encerra a conexão com o banco
```

Os métodos da interface **`ResultSet`** permitem a manipulação dos resultados individuais de uma tabela de resultados. Métodos como **`getDouble()`**, **`getInt()`** e **`getString()`**, que recebem como argumento a especificação de uma coluna da tabela, permitem acessar o valor da coluna especificada na tupla corrente para os diversos tipos de dados suportados.

Para varrer a tabela, um cursor é mantido. Inicialmente, ele está posicionado antes do início da tabela, mas pode ser manipulado pelos métodos **`first()`**, **`next()`**, **`previous()`**, **`last()`** e **`absolute(int row)`**. Por exemplo,

```
ResultSet r = s.executeQuery("SELECT * FROM EMPREGADO");
System.out.println("Id Nome");
while (r.next()) { System.out.println(r.getString("Id")+  + r.getString("Nome")); }
r.close();
```

Para lidar com atributos que podem assumir valores nulos, o método **`wasNull()`** é oferecido. Ele retorna verdadeiro quando o valor obtido pelo método **`getXXX()`** for nulo, em que **`XXX`** é um dos tipos **SQL**.

A interface **`ResultSetMetadata`** permite obter informação sobre a tabela com o resultado da consulta. Um objeto desse tipo pode ser obtido através da aplicação do método **`getMetaData()`** ao **`ResultSet`**. Uma vez obtido esse objeto, a informação desejada pode ser obtida através de métodos tais como **`getColumnCount()`**, **`getColumnLabel()`**, **`getColumnTypeName()`** e **`getColumnType()`**. O último método retorna tipos que podem ser identificados a partir de constantes definidas na classe **`java.sql.Types`**.

Além da forma interface **`Statement`**, **`JDBC`** oferece duas formas alternativas que permitem, respectivamente, ter acesso a comandos **SQL** pré-compilados (**`PreparedStatement`**) e a procedimentos armazenados no banco de dados (**`CallableStatement`**).

7.3.2.4 Aplicação ListaObrasDeArte

Esta seção apresenta uma aplicação Java (Código 7.2) que acessa o banco de dados `Museu` hospedado na máquina local. Com propósitos de ilustração, o **Apache Derby**[2]será utilizado nessa aplicação. No entanto, o leitor deve sentir-se à vontade para usar outro **SGBD** se este já estiver instalado e configurado. Os *links* abaixo apresentam dicas na configuração do **Apache Derby** e na posterior integração com os dois principais *IDEs open-source*:

- Netbeans: https://netbeans.org/kb/docs/ide/java-db_pt_BR.html.
- Eclipse: http://db.apache.org/derby/integrate/plugin_help/start_toc.html.

Essa aplicação supõe que o banco de dados `Museu` foi criado e populado conforme descrito no *script* apresentado no Código 7.1. A classe `ListaObrasDeArte` apenas possui o método `main()`, que simplesmente conecta com o banco de dados `Museu` e imprime a lista de obras de arte armazenada nesse banco de dados. A saída da execução desse programa é similar à apresentada na Tabela 7.1.

ID	TITULO	ARTISTA	MATERIAL	ANO	CATEGORIA	TIPO	ALTURA
1	Mona Lisa	Leonardo da Vinci	Madeira	1503	1	Óleo	null
2	David	Michelangelo	Mármore	1501	2	null	4,10
3	O Lavrador de Café	Cândido Portinari	Madeira	1939	1	Óleo	null
4	O Pensador	Auguste Rodin	Bronze	1909	2	null	1,40

Tabela 7.1 Saída da execução da classe `ListaObrasDeArte`.

Conforme se pode observar, o método `main()` executa os quatro passos necessários para acessar um banco de dados relacional utilizando a *API* `JDBC`:

1. habilitar o *driver* `JDBC` (linha 14);
2. estabelecer uma conexão com o banco de dados (linhas 16-21);
3. executar a consulta **SQL** (linhas 23-25); e
4. apresentar resultados da consulta (linhas 27-61).

[2] http://db.apache.org/derby/.

```java
/**
 * Classe ListaObrasDeArte -- Conecta com o banco de dados Museu e imprime a lista de obras de arte
 * armazenada na tabela ObraDeArte presente nesse banco de dados.
 *
 * @author Delano Medeiros Beder
 */
public class ListaObrasDeArte {

    public static void main(String[] args) {
        final String fmt10 = "%-10.10s";
        final String fmt20 = "%-20.20s";

        try {
            Class.forName(driver); // Carrega driver

            // Estabelece conexão com o banco de dados
            String driver = "org.apache.derby.jdbc.ClientDriver";
            String url = "jdbc:derby://localhost:1527/Museu";
            String user = "root";
            String password = "root";
            Connection connection = DriverManager.getConnection(url, user, password);

            // Monta e executa consulta
            Statement s = connection.createStatement();
            ResultSet r = s.executeQuery("SELECT * FROM ObraDeArte");

            // Apresenta estrutura da Tabela
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < 12; i++) {
                sb.append("----------");
            }
            System.out.println(sb);

            ResultSetMetaData m = r.getMetaData();
            int colCount = m.getColumnCount();
            for (int i = 1; i <= colCount; ++i) {
                if (m.getColumnType(i) != Types.VARCHAR) {
                    System.out.printf(fmt10, m.getColumnName(i));
                } else {
                    System.out.printf(fmt20, m.getColumnName(i));
                }
            }
            System.out.println();
            System.out.println(sb);

            while (r.next()) {
                System.out.printf(fmt10, String.valueOf(r.getInt(1))); // Obtém pelo número da coluna
                System.out.printf(fmt20, r.getString("TITULO")); // Obtém pelo nome da coluna
                System.out.printf(fmt20, r.getString(3)); // Obtém pelo número da coluna
                System.out.printf(fmt20, r.getString("MATERIAL")); // Obtém pelo nome da coluna
                System.out.printf(fmt10, String.valueOf(r.getInt(5))); // Obtém pelo número da coluna
                System.out.printf(fmt10, String.valueOf(r.getInt(6))); // Obtém pelo número da coluna
                System.out.printf(fmt20, r.getString("TIPO")); // Obtém pelo nome da coluna
                double altura = r.getDouble(8); // Obtém pelo número da coluna

                if (altura == 0) {
                    System.out.printf(fmt10+"\n", "null");
                } else {
                    System.out.printf("%-10.2f\n", r.getDouble(8));
                }
            }
            r.close();
            connection.close(); // Fecha conexão com banco de dados
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

Código 7.2 Classe `ListaObrasDeArte`.

7.4 Mapeamento objeto-relacional

Apesar do paradigma orientado a objetos estar sendo cada vez mais difundido no processo de desenvolvimento de software, não existem hoje soluções comerciais robustas e amplamente aceitas nesse paradigma para a persistência de dados. Conforme dito, este é um mercado dominado pelos bancos de dados relacionais. Nesse contexto, o mapeamento do modelo orientado a objetos para o relacional é uma necessidade cada vez mais importante no processo de desenvolvimento de software.

O mapeamento objeto-relacional (ORM[3]) é uma técnica de desenvolvimento utilizada para reduzir a impedância da programação orientada aos objetos utilizando bancos de dados relacionais. As tabelas do banco de dados são representadas através de classes, e os registros de cada tabela são representados como instâncias das classes correspondentes.

Com essa técnica, o programador não precisa se preocupar com os comandos na linguagem **SQL**. Ele usará uma interface de programação (conjunto de classes) simples que faz todo o trabalho de persistência.

7.4.1 *Data Access Object*

Uma das abordagens mais utilizadas no desenvolvimento do mapeamento objeto-relacional é o *Data Access Object* (*DAO*), que consiste em um padrão de projeto para a persistência de dados e que permite separar regras de negócio das regras de acesso a banco de dados. Ou seja, em uma aplicação que acessa um banco de dados, todas as funcionalidades relacionadas ao banco de dados, tais como obter conexões ou executar comandos **SQL**, devem ser feitas por classes *DAO* (CRUPI; MALKS; ALUR, 2004).

A principal vantagem de usar *DAOs* é a separação simples e rigorosa entre duas partes importantes de uma aplicação que não devem e não podem conhecer quase nada uma da outra e que podem evoluir frequentemente e independentemente. Modificações na lógica de persistência não alteram a lógica de negócio, desde que a interface entre elas não seja modificada.

[3] Do inglês: Object-Relational Mapping

Além da vantagem mencionada no parágrafo anterior, pode-se citar também as seguintes vantagens da utilização de *DAOs*:

- Pode ser usada em uma vasta porcentagem de aplicações;
- Esconde todos os detalhes relativos a armazenamento de dados do resto da aplicação;
- Atua como um intermediário entre a aplicação e o banco de dados;
- Mitiga ou resolve problemas de comunicação entre a base de dados e a aplicação, evitando estados inconsistentes de dados.

No contexto específico da linguagem de programação Java, um *DAO* pode ser implementado de várias maneiras. Pode variar desde uma simples interface que separa partes de acesso a dados da lógica de negócio de uma aplicação até *frameworks* e produtos comerciais específicos tais como Hibernate[4] e TopLink[5].

Uma discussão extensiva do padrão de projeto *Data Access Object* encontra-se fora do escopo deste material. Para um estudo mais aprofundado desse tópico, sugere-se que o leitor consulte a seguinte referência: Crupi, Malks e Alur (2004).

7.4.2 Sistema de gerenciamento de obras de arte

Esta seção apresenta uma implementação simples, baseada em alguns padrões de projeto, para o mecanismo de persistência do sistema de gerenciamento de obras de artes discutido na Seção 3.7.

Padrões de projeto. Uma das vantagens do paradigma de orientação a objetos é o seu suporte à reusabilidade, que é a prática de incorporar componentes/módulos de software já existentes em sistemas de software para os quais eles não foram originalmente desenvolvidos. Padrões de projeto (GAMMA et al., 2000) procuram documentar conhecimentos e experiências de projetos existentes no intuito de ajudar na busca de soluções apropriadas para problemas de projeto de componentes/módulos de software.

Um padrão de projeto documenta uma alternativa de solução para um problema específico, recorrente em diversas aplicações. Ele tem uma estrutura e formato particular e descreve um problema que ocorre em um domínio em particular e como resolver esse problema.

[4] http://www.hibernate.org/.
[5] http://www.oracle.com/technetwork/middleware/toplink/overview/index.html.

O principal benefício decorrente do uso de padrões de projeto é facilitar a comunicação entre desenvolvedores de software, de uma mesma equipe ou independentes, ao permitir o emprego de estruturas de um nível de abstração maior do que linguagens de programação, porém com o mesmo grau de formalismo destas, contribuindo assim, positivamente, para a reutilização de software.

Uma descrição detalhada dos padrões de projeto existentes está além do escopo deste livro. Para mais detalhes, o leitor interessado deve consultar as seguintes referências: Gamma et al. (2000) e Crupi, Malks e Alur (2004).

7.4.2.1 Classe `ConnectionFactory` – Fábrica de conexões

A classe **`ConnectionFactory`** (Código 7.3) implementa o padrão de projeto *Factory* (GAMMA et al., 2000), que prega o encapsulamento da construção de objetos. Note que o método **`getConnection()`** é uma **fábrica de conexões**. Isto é, esse método é responsável por criar e retornar novas conexões ao banco de dados (instâncias de classes que implementam a interface **`java.sql.Connection`**). O usuário dessa classe basta invocar o método **`getConnection()`** e terá como retorno uma conexão ao banco de dados pronta para o uso, não importando de onde ela veio e eventuais detalhes da criação.

```java
/**
 * Classe ConnectionFactory -- Fábrica de conexões ao banco de dados
 *
 * @author Delano Medeiros Beder
 */
public class ConnectionFactory {
    public Connection getConnection() {
        Connection connection = null;
        try {
            String driver = "org.apache.derby.jdbc.ClientDriver";
            String url = "jdbc:derby://localhost:1527/Museu";
            Class.forName(driver);
            connection = DriverManager.getConnection(url, "root", "root");
        } catch (Exception e) {
            e.printStackTrace();
        }
        return connection;
    }
}
```

Código 7.3 Classe `ConnectionFactory` – Fábrica de conexões.

Ao encapsular dessa maneira, é possível posteriormente reimplementar o método **`getConnection()`** (o comportamento da obtenção de conexões), para, por exemplo, trocar o **SGBD** (de **Apache Derby** para **MySQL**) ou utilizar um mecanismo de *pooling* de conexões, sem ter que alterar as demais classes da aplicação.

7.4.2.2 Classe `ObjectDTO`

A classe `ObjectDTO` (Código 7.4) implementa o padrão de projeto *Data Transfer Object*, ou simplesmente *DTO* (CRUPI; MALKS; ALUR, 2004). Esse padrão de projeto é usado para transferir dados entre subsistemas de um software. *Data Transfer Objects* são frequentemente usados em conjunção com *DAOs* para obter e/ou armazenar dados de um banco de dados.

No caso do mecanismo de persistência do sistema de gerenciamento de obras de artes, a classe `ObjectDTO` torna-se a raiz da hierarquia de objetos que podem ser persistidos no banco de dados. Dessa forma, a classe `ObraDeArte`, que é a raiz da hierarquia de obras de artes, torna-se subclasse da classe `ObjectDTO`.

```
/**
 * Classe ObjectDTO -- Raiz da hierarquia de objetos que podem ser persistidos no banco de dados.
 *
 * @author Delano Medeiros Beder
 */
public abstract class ObjectDTO {

    private int id;

    public int getId() {
        return id;
    }

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

public abstract ObraDeArte extends ObjectDTO {
     /* Declaração de atributos e métodos - Ver Código 3.6 */
}
```

Código 7.4 Classe `ObjectDTO`.

7.4.2.3 Interface `IGenericDAO`

A interface `IGenericDAO` (Código 7.5) define as funcionalidades genéricas necessárias à persistência de dados. Ou seja, define os métodos genéricos que se aplicam a todos os objetos que podem ser persistidos (instâncias das subclasses de `ObjectDTO`).

- O método `insert()` é responsável por inserir uma instância das subclasses de `ObjectDTO` no banco de dados;
- O método `selectByID()` é responsável por selecionar, do banco de dados, um objeto (instância das subclasses de `ObjectDTO`) dado o seu `id`;

```java
/**
 * Interface IGenericDAO -- Define as funcionalidades genéricas da persistência de dados.
 *
 * @author Delano Medeiros Beder
 */
public interface IGenericDAO {

    void insert(ObjectDTO obj) throws DAOException;

    ObjectDTO selectByID(int id) throws DAOException;

    Set<ObjectDTO> selectAll() throws DAOException;

    void delete(int id) throws DAOException;

    void update(ObjectDTO obj) throws DAOException;
}
```

Código 7.5 Interface `IGenericDAO`.

- O método `selectAll()` é responsável por selecionar, do banco de dados, todos os objetos (instâncias das subclasses de `ObjectDTO`). Note que esse método retorna um conjunto (interface `Set`) de instâncias;
- O método `delete()` é responsável por remover, do banco de dados, um objeto (instância das subclasses de `ObjectDTO`) dado o seu `id`;
- E, por fim, o método `update()` é responsável por atualizar, no banco de dados, as informações relacionadas a um objeto (instância das subclasses de `ObjectDTO`).

7.4.2.4 Interface `IObraDeArteDAO`

A interface `IObraDeArteDAO` (Código 7.6) estende a interface `IGenericDAO` e define as funcionalidades específicas relacionadas à persistência de instâncias das subclasses de `ObraDeArte`. Visto que a classe `ObraDeArte` é abstrata, ela não pode ter instâncias; consequentemente, apenas instâncias de suas subclasses que podem ser persistidas em um banco de dados.

- O método `selectAllEsculturas()` é responsável por selecionar, do banco de dados, todas as esculturas (instâncias da classe `Escultura`);
- O método `selectAllPinturas()` é responsável por selecionar, do banco de dados, todas as pinturas (instâncias da classe `Pintura`).

À primeira vista, pode-se argumentar que não há justificativa para separar as funcionalidades relacionadas à persistência de dados em duas interfaces (`IGenericDAO` e

```
/**
 * Interface IObraDeArteDAO — Define as funcionalidades específicas relacionadas à persistência
 * de instâncias das subclasses de ObraDeArte.
 *
 * @author Delano Medeiros Beder
 */
public interface IObraDeArteDAO extends IGenericDAO {

    Set<Escultura> selectAllEsculturas() throws DAOException;

    Set<Pintura> selectAllPinturas() throws DAOException;
}
```

Código 7.6 Interface `IObraDeArteDAO`.

`IObraDeArteDAO`) distintas, pois apenas obras de arte podem ser persistidas nesse sistema. Porém, essa separação proporciona uma maior manutenibilidade[6]ao mecanismo de persistência, pois facilita a sua evolução. Se futuramente for necessário adicionar uma nova entidade ao sistema (por exemplo, persistir uma nova entidade/classe), basta o desenvolvedor adicionar uma nova interface que estenda `IGenericDAO`, sem a necessidade de qualquer alteração nas interfaces e classes existentes.

7.4.2.5 Classe `GenericJDBCDAO`

A classe `GenericJDBCDAO` (Código 7.7) implementa parcialmente a interface `IGenericDAO` utilizando como estratégia para a persistência de dados o emprego da *API* `JDBC` discutida anteriormente nesta unidade.

Note que essa classe define dois métodos abstratos que devem ser implementados pelas subclassses de `GenericJDBCDAO`:

- O método `getTableName()` é responsável por retornar o nome da tabela na qual as operações **SQL** serão executadas;
- O método `createObjectDTO()` é responsável por, dada uma instância da interface `ResultSet` (resultados da execução de uma consulta **SQL**), criar e retornar uma instância de alguma subclasse da classe `ObjectDTO`.

Note que, apesar de serem abstratos, esses dois métodos são invocados nos demais métodos concretos presentes na classe `GenericJDBCDAO`. Ou seja, pode-se considerar cada método concreto como um *Template Method* – padrão de projeto *Template Method* (GAMMA et al., 2000).

[6] Manutenibilidade, em engenharia de software, é o grau de facilidade com que um sistema de software pode ser corrigido ou aperfeiçoado (PRESSMAN, 2011).

```java
/**
 * Classe GenericJDBCDAO — implementa as funcionalidades genéricas da persistência de dados.
 *
 * @author Delano Medeiros Beder
 */
public abstract class GenericJDBCDAO implements IGenericDAO {

    protected abstract String getTableName();
    protected abstract ObjectDTO createObjectDTO(ResultSet rs) throws DAOException;

    public final Set<ObjectDTO> selectAll() throws DAOException {
        Set<ObjectDTO> set = new TreeSet<>();
        Connection connection = new ConnectionFactory().getConnection();
        try {
            String sql = "SELECT * FROM " + this.getTableName();
            Statement stmt = connection.createStatement();
            ResultSet rs = stmt.executeQuery(sql);
            while (rs.next()) {
                set.add(this.createObjectDTO(rs));
            }
            stmt.close(); conn.close(); // fechando a conexão
        } catch (SQLException e) {
            throw new DAOException(e);
        }
        return set;
    }

    public final ObjectDTO selectByID(int id) throws DAOException {
        ObjectDTO dto = null;
        Connection conn = new ConnectionFactory().getConnection();
        try {
            String sql = "SELECT * FROM " + this.getTableName()
                    + " WHERE ID = " + id;
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sql);
            if (rs.next()) {
                dto = this.createObjectDTO(rs);
            }
            stmt.close(); conn.close(); // fechando a conexão
        } catch (SQLException e) {
            throw new DAOException(e);
        }
        return dto;
    }

    protected final int selectLastID() throws DAOException {
        Connection conn = new ConnectionFactory().getConnection();
        int lastID = 0;
        try {
            String sql = "SELECT MAX(ID) FROM " + this.getTableName();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sql);
            if (rs.next())
                lastID = rs.getInt(1);
            stmt.close(); conn.close(); // fechando a conexão
        } catch (SQLException e) {
            throw new DAOException(e);
        }
        return lastID;
    }

    public final void delete(int id) throws DAOException {
        Connection conn = new ConnectionFactory().getConnection();
        try {
            String sql = "DELETE FROM " + this.getTableName() + " WHERE ID = ?";
            PreparedStatement stmt = conn.prepareStatement(sql);
            stmt.setInt(1, id);
            stmt.executeUpdate();
            stmt.close(); conn.close(); // fechando a conexão
        } catch (SQLException e) {
            throw new DAOException(e);
        }
    }
}
```

Código 7.7 Classe `GenericJDBCDAO`.

Um *Template Method* auxilia na definição de um algoritmo com partes deste definidos por métodos abstratos. As subclasses devem se responsabilizar por essas partes abstratas, deste algoritmo, que serão implementadas possivelmente de várias formas, ou seja, cada subclasse irá implementar de acordo com a sua necessidade e oferecer um comportamento concreto construindo todo o algoritmo. No caso do exemplo, discutido nesta seção, cada subclasse, ao implementar os métodos `getTableName()` e `createObjectDTO()`, retornará o nome de uma tabela diferente e construirá o `ObjectDTO` seguindo suas necessidades.

E, por fim, é importante salientar que a implementação dos métodos presentes na classe `GenericJDBCDAO` seguem o mesmo padrão de passos:

1. Obter uma conexão ao banco de dados através da fábrica de conexões discutida na Seção 7.4.2.1;
2. Realizar uma operação **SQL** no banco de dados através da interface `Statement`;
3. Dado o resultado da operação realizada no banco de dados, construir e retornar uma resposta.

A única exceção é o método `delete()`, que se diferencia levemente dos demais métodos por não retornar valores e por utilizar a interface `PreparedStatement`, a qual possibilita a execução de comandos **SQL** pré-compilados.

7.4.2.6 Classe `ObraDeArteJDBCDAO`

A classe `ObraDeArteJDBCDAO` estende a classe `GenericDAO` (Códigos 7.8 e 7.9) e é responsável pela implementação das funcionalidades relacionadas à persistência de instâncias das subclasses de `ObraDeArte`.

É importante salientar que, por motivos de simplicidade, as obras de arte são persistidas em uma única tabela (`ObraDeArte`). A coluna `CATEGORIA`, presente nessa tabela, identifica se a obra de arte é uma pintura (`CATEGORIA = 1`) ou uma escultura (`CATEGORIA = 2`). Essas duas constantes são definidas como os atributos finais `PINTURA` e `ESCULTURA` da classe `ObraDeArteJDBCDAO`. Outra possibilidade seria a persistência de cada subclasse concreta em uma tabela diferente. Ou seja, pinturas seriam persistidas na tabela `Pintura`, e esculturas na tabela `Escultura`.

Conforme se pode observar pela Figura 7.1, a tabela `ObraDeArte` possui duas pinturas (primeira e terceira tupla) e duas esculturas (segunda e quarta tupla).

Pode-se notar que, além dos métodos `getTableName()` e `createObjectDTO()`, a classe `ObraDeArteJDBCDAO` também implementa os métodos definidos na interface `IObraDeArteDAO`:

- O método `getTableName()` retorna o nome da tabela (a *string* "ObraDeArte") utilizada para persistir as instâncias das subclasses de `ObraDeArte`;
- O método `createObjectDTO` retorna uma instância da classe `Pintura` ou da classe `Escultura`. Observe que a coluna `CATEGORIA` é utilizada para determinar a instância de qual classe será criada.
- O método `insert()` é responsável por inserir obras de arte (pinturas ou esculturas) no banco de dados. Uma vez que esse método é genérico (definido na interface `IGenericDAO`), o parâmetro é uma instância da classe `ObjectDTO` – raiz da hierarquia de objetos que podem ser persistidos no banco de dados. Note que esse método utiliza o operador **instanceof** para determinar quais valores serão persistidos no banco de dados. Por fim, esse método utiliza a interface `PreparedStatement`, que possibilita a execução de comandos **SQL** pré-compilados.
- O método `update()` é semelhante ao método `insert()` e é responsável por atualizar as informações relacionadas às obras de arte (pinturas ou esculturas) no banco de dados. Analogamente ao método `insert()`, esse método é genérico (definido na interface `IGenericDAO`), portanto o parâmetro é uma instância da classe `ObjectDTO`.
- O método `selectAllEsculturas()` é responsável por selecionar e retornar, do banco de dados, todas as esculturas (instâncias da classe `Escultura`). Note que a implementação é basicamente invocar o método `selectAll()`, implementado pela classe pai `GenericJDBCDAO`, e construir um novo conjunto com apenas instâncias da classe `Escultura`;
- O método `selectAllPinturas()` é responsável por selecionar, do banco de dados, todas as pinturas (instâncias da classe `Pintura`). A implementação é similar à implementação do método `selectAllEsculturas()`. Ou seja, ele simplesmente invoca o método `selectAll()`, implementado pela classe pai `GenericJDBCDAO`, e constrói um novo conjunto com apenas instâncias da classe `Pintura`.

```java
/**
 * Classe ObraDeArteJDBCDAO -- Implementa as funcionalidades relacionadas à persistência de
 * instâncias das subclasses de ObraDeArte.
 *
 * @author Delano Medeiros Beder
 */
class ObraDeArteJDBCDAO extends GenericJDBCDAO implements IObraDeArteDAO {

    public static final int PINTURA = 1;
    public static final int ESCULTURA = 2;

    protected String getTableName() {
        return "ObraDeArte";
    }

    protected ObraDeArte createObjectDTO(ResultSet rs) throws DAOException {
        ObraDeArte obra;
        try {
            int id = rs.getInt("ID");
            String titulo = rs.getString("TITULO");
            String artista = rs.getString("ARTISTA");
            String material = rs.getString("MATERIAL");
            int ano = rs.getInt("ANO");
            int categoria = rs.getInt("CATEGORIA");
            if (categoria == PINTURA) {
                String tipo = rs.getString("TIPO");
                obra = new Pintura(titulo, artista, material, ano, tipo);
            } else {
                double altura = rs.getDouble("ALTURA");
                obra = new Escultura(titulo, artista, material, ano, altura);
            }
            obra.setId(id);
            return obra;
        } catch (SQLException e) {
            throw new DAOException(e);
        }
    }

    public void insert(ObjectDTO obj) throws DAOException {
        ObraDeArte obra = (ObraDeArte) obj;
        StringBuilder sql = new StringBuilder();
        sql.append("INSERT INTO ");
        sql.append(this.getTableName());
        sql.append(" (ID, TITULO, ARTISTA, MATERIAL, ANO, CATEGORIA, ");
        if (obra instanceof Pintura) {
            sql.append("TIPO) VALUES(?,?,?,?,?,?,?)");
        } else {
            sql.append("ALTURA) VALUES(?,?,?,?,?,?,?)");
        }
        try {
            Connection conn = new ConnectionFactory().getConnection();
            PreparedStatement stmt = conn.prepareStatement(sql.toString());
            stmt.setInt(1, this.selectLastID() + 1);
            stmt.setString(2, obra.getTítulo());
            stmt.setString(3, obra.getArtista());
            stmt.setString(4, obra.getMaterial());
            stmt.setInt(5, obra.getAno());
            if (obra instanceof Pintura) {
                stmt.setInt(6, PINTURA);
                stmt.setString(7, ((Pintura) obra).getTipo());
            } else {
                stmt.setInt(6, ESCULTURA);
                stmt.setDouble(7, ((Escultura) obra).getAltura());
            }
            stmt.executeUpdate();
            stmt.close();
            conn.close();
        } catch (SQLException e) {
            throw new DAOException(e);
        }
    }
    .... Continuação ....
```

Código 7.8 Classe `ObraDeArteJDBCDAO`.

```java
.... Continuação ....

    public void update(ObjectDTO obj) throws DAOException {
        ObraDeArte obra = (ObraDeArte) obj;
        StringBuilder sql = new StringBuilder();
        sql.append("UPDATE ");
        sql.append(this.getTableName());
        sql.append(" SET TITULO = ?, ");
        sql.append("ARTISTA = ?, MATERIAL = ?, ANO = ?, DESC = ?, ");
        if (obra instanceof Pintura) {
            sql.append("TIPO = ? WHERE ID = ?");
        } else {
            sql.append("ALTURA = ? WHERE ID = ?");
        }
        try {
            Connection conn = new ConnectionFactory().getConnection();
            PreparedStatement stmt = conn.prepareStatement(sql.toString());
            stmt.setString(1, obra.getTítulo());
            stmt.setString(2, obra.getArtista());
            stmt.setString(3, obra.getMaterial());
            stmt.setInt(4, obra.getAno());
            if (obra instanceof Pintura) {
                stmt.setInt(5, PINTURA);
                stmt.setString(6, ((Pintura) obra).getTipo());
            } else {
                stmt.setInt(5, ESCULTURA);
                stmt.setDouble(6, ((Escultura) obra).getAltura());
            }
            stmt.executeUpdate();
            stmt.close();
            conn.close();
        } catch (SQLException e) {
            throw new DAOException(e);
        }
    }

    public Set<Escultura> selectAllEsculturas() throws DAOException {
        Set<ObjectDTO> all = this.selectAll();
        Set<Escultura> esculturas = new HashSet<>();
        for (ObjectDTO obj : all) {
            if (obj instanceof Escultura) {
                esculturas.add((Escultura)obj);
            }
        }
        return esculturas;
    }

    public Set<Pintura> selectAllPinturas() throws DAOException {
        Set<ObjectDTO> all = this.selectAll();
        Set<Pintura> pinturas = new TreeSet<>();
        for (ObjectDTO obj : all) {
            if (obj instanceof Pintura) {
                pinturas.add((Pintura)obj);
            }
        }
        return pinturas;
    }
}
```

Código 7.9 Classe `ObraDeArteJDBCDAO` – Continuação.

7.4.2.7 Fábrica de *DAOs*

De maneira análoga à classe **ConnectionFactory**, a classe **DAOFactory** (Código 7.10) também implementa o padrão de projeto *Factory* (GAMMA et al., 2000). Note que o método abstrato **getObraDeArteDAO()** é uma fábrica de **IObraDeArteDAO**, que é responsável pela persistência de instâncias das subclasses de **ObraDeArte**. Esse método abstrato deve ser implementado pelas subclasses de **DAOFactory** de acordo com a estratégia utilizada na persistência dos objetos.

```
/**
 * Classe DAOFactory -- Fábrica de DAOs
 *
 * @author Delano Medeiros Beder
 */
abstract public class DAOFactory {

    private static DAOFactory instance = null;

    public static DAOFactory getInstance() {
        if (instance == null) {
            instance = new JDBCDAOFactory();
        }
        return instance;
    }

    public abstract IObraDeArteDAO getObraDeArteDAO() throws DAOException;
}
```

Código 7.10 Classe **DAOFactory**.

Como exemplo, observe a classe **JDBCDAOFactory**, subclasse de **DAOFactory**, que reimplementa o método **getObraDeArteDAO()** retornando uma instância da classe **ObraDeArteJDBCDAO** a qual, conforme discutido anteriormente, utiliza a *API* **JDBC** como estratégia para a persistência dos objetos.

```
/**
 * Classe JDBCDAOFactory -- Classe concreta que retorna uma fábrica de DAOs que utiliza a
 * API JDBC, como estratégia para a persistência de dados.
 *
 * @author Delano Medeiros Beder
 */
class JDBCDAOFactory extends DAOFactory {

    public IObraDeArteDAO getObraDeArteDAO() throws DAOException {
        return new ObraDeArteJDBCDAO();
    }
}
```

Código 7.11 Classe **JDBCDAOFactory**.

Além disso, para evitar a existência de diferentes fábricas de *DAO*, a classe **DAOFactory** também implementa o padrão de projeto *Singleton* (GAMMA et al., 2000), que garante a existência de apenas uma instância de uma classe, mantendo

um ponto global, representado pelo **getInstance()**, de acesso a essa única instância. No caso do exemplo discutido nesta seção, o objeto *singleton* é uma instância da classe **JDBCDAOFactory**.

Por fim, à primeira vista, pode-se argumentar que não há justificativa para separar as funcionalidades relacionadas à criação de objetos *DAOs* em duas classes (**DAOFactory** e **JDBCDAOFactory**) distintas, pois apenas obras de arte podem ser persistidas nesse sistema. Porém, essa separação proporciona uma maior manutenibilidade ao mecanismo de persistência, pois facilita a sua evolução. Se futuramente for necessário mudar a estratégia utilizada na persistência de dados (por exemplo, de **JDBC** para **JPA**[7]), basta o desenvolvedor adicionar uma nova classe que estende **DAOFactory** sem necessidade de qualquer alteração nas interfaces e classes existentes.

7.4.2.8 Aplicação **ImprimeObrasDeArte**

Esta seção apresenta uma aplicação Java (Código 7.12) similar à aplicação **ListaObrasDeArte** discutida na Seção 7.3.2.4 (Código 7.2).

```
/**
 * Classe ImprimeObrasDeArte -- Recupera e imprime a lista de obras de arte.
 *
 * @author Delano Medeiros Beder
 */
public class ImprimeObrasDeArte {
    public static void main(String[] args) {
        try {
            IObraDeArteDAO dao = DAOFactory.getInstance().getObraDeArteDAO();
            Set<ObjectDTO> set = dao.selectAll();

            for (ObjectDTO obj : set) {
                ObraDeArte obra = ((ObraDeArte) obj);
                System.out.printf("%-10.10s", obra.getId());
                System.out.printf("%-20.20s", obra.getTítulo());
                System.out.printf("%-20.20s", obra.getArtista());
                System.out.printf("%-20.20s", obra.getMaterial());
                System.out.printf("%-10.10s", obra.getAno());
                if (obra instanceof Pintura) {
                    System.out.printf("%-10.10s", "1");
                    System.out.printf("%-20.20s", ((Pintura) obra).getTipo());
                    System.out.printf("%-10.10s" + "\n", "null");
                } else {
                    System.out.printf("%-10.10s", "2");
                    System.out.printf("%-20.20s", "null");
                    System.out.printf("%-10.2f\n", ((Escultura) obra).getAltura());
                }
            }
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

Código 7.12 Classe **ImprimeObrasDeArte**.

[7] http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html.

Essa aplicação supõe que o banco de dados `Museu` foi criado e populado conforme descrito no *script* apresentado no Código 7.1. A classe `ImprimeObrasDeArte` apenas possui o método `main()`, que simplesmente utiliza um objeto *Data Access Object* para recuperar e imprimir a lista de obras de arte armazenadas nesse banco de dados. A saída da execução desse programa é similar à apresentada na Tabela 7.1.

7.5 Considerações finais

Esta unidade apresentou os conceitos relacionados às operações de acesso a um sistema de banco de dados relacional na linguagem de programação Java.

A próxima unidade tem como objetivo introduzir o mundo Java Web ao apresentar os conceitos básicos e tecnologias relacionadas ao desenvolvimento de aplicações Web na linguagem de programação Java.

7.6 Estudos complementares

Para estudos complementares sobre os tópicos abordados nesta unidade, o leitor interessado pode consultar as seguintes referências:

ARNOLD, K.; GOSLING, J.; HOLMES, D. *The Java Programming Language*. 4. ed. Boston: Addison-Wesley, 2005.

CRUPI, J.; MALKS, D.; ALUR, D. *Core J2EE Patterns* – As melhores práticas e estratégias de design. 2. ed. Rio de Janeiro: Campus, 2004.

DEITEL, P.; DEITEL, H. *Java*: Como programar. 8. ed. São Paulo: Pearson Brasil, 2010.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Padrões de Projetos*: Soluções Reutilizáveis de Software Orientado a Objetos. Porto Alegre: Bookman Companhia, 2000.

ORACLE. *The Java™ Tutorials* – Trail: JDBC™Database Access. 2014. Disponível em: http://docs.oracle.com/javase/tutorial/jdbc/. Acesso em: 12 ago. 2014.

SPEEGLE, G. D. *JDBC*: Practical Guide for Java Programmers. 1. ed. Burlington: Morgan Kaufmann, 2001.

UNIDADE 8

Desenvolvimento Web em Java

8.1 Primeiras palavras

Entre os atrativos da linguagem de programação Java está a facilidade que essa linguagem oferece para desenvolver aplicações Web. Nesse contexto, esta unidade tem como objetivo introduzir o mundo Java Web ao apresentar os conceitos básicos e tecnologias relacionadas ao desenvolvimento de aplicações Web na linguagem de programação Java.

8.2 Problematizando o tema

Ao final desta unidade, espera-se que o leitor seja capaz de reconhecer e definir precisamente os conceitos básicos e tecnologias relacionadas ao desenvolvimento de aplicações Web na linguagem de programação Java. Dessa forma, esta unidade pretende discutir as seguintes questões:

- Quais as principais características da linguagem `HTML`?
- Quais as principais características do protocolo `HTTP`?
- O que é um cliente Web ? O que é um servidor Web ? Quais são seus papéis no mundo do desenvolvimento Web ?
- Quais as principais características da tecnologia Java *Applet*?
- Quais as principais características da tecnologia Java *Servlet*?
- Quais as principais características da tecnologia Java *JSP*?
- Quais as semelhanças e diferenças entre essas tecnologias Web presentes na plataforma Java – *Applet*, *Servlet* e *JSP*?

8.3 Desenvolvimento Web

Entre os atrativos da linguagem Java está a facilidade que essa linguagem oferece para desenvolver aplicações para execução em sistemas distribuídos. Já em sua primeira versão, a plataforma Java oferecia facilidades para o desenvolvimento de aplicações cliente-servidor usando os mecanismos da internet, tais como os protocolos TCP/IP e UDP.

Define-se por cliente-servidor uma relação entre processos que atuam em máquinas diferentes. O servidor é aquele que fornece um serviço, e o cliente é aquele que utiliza esse serviço (consumidor) (COULORIS; DOLLIMORE; KINDBERG, 2007).

Um servidor Web é um modelo inteiramente novo, introduzido pela internet, e geralmente conta com uma enorme quantidade de clientes, que se comunicam com servidores, responsáveis por atender a milhares de requisições. A comunicação entre eles é feita utilizando-se o protocolo `HTTP` (*Hypertext Transfer Protocol*), normalmente por meio do envio de requisições de um navegador (*browser*) para um servidor Web, que retorna as respostas por meio de conteúdo Web (tais como páginas `HTML`) de volta ao cliente (PEREIRA, 2006).

Dessa forma, a definição de um servidor Web é a seguinte: trata-se de um aplicativo de software que provê a funcionalidade de aceitar pedidos `HTTP` de clientes, geralmente os navegadores (*browsers*), e servi-los com respostas `HTTP`, incluindo opcionalmente dados, que geralmente são páginas Web, tais como documentos `HTML` com objetos embutidos, como imagens, etc. Como exemplo de servidor Web, pode-se citar o Apache[1], que é um software livre, mantido pela **Apache Software Foundation**, compatível com o protocolo `HTTP`.

8.3.1 Protocolo `HTTP`

O protocolo `HTTP` (*HyperText Transfer Protocol*) é um protocolo de comunicação (na camada de aplicação segundo o modelo OSI) que opera sobre o protocolo `TCP/IP` para estabelecer um mecanismo de serviço do tipo requisição-resposta.

Os principais serviços de `HTTP` incluem:

- `GET` – solicita ao servidor Web o envio de um recurso (página Web, arquivo, imagem, etc.). É o serviço essencial para o protocolo;
- `HEAD` – variante de `GET` que solicita ao servidor Web o envio apenas de informações sobre o recurso;
- `PUT` – permite que o cliente autorizado armazene ou altere o conteúdo de um recurso mantido pelo servidor Web;
- `POST` – permite que o cliente envie mensagens e conteúdo de formulários para servidores Web que irão manipular a informação de maneira adequada;

[1] http://httpd.apache.org/.

- **DELETE** – permite que o cliente autorizado remova um recurso mantido pelo servidor Web.

Um cabeçalho **HTTP** é composto de uma linha contendo a especificação do serviço e recurso associado, seguida por linhas contendo parâmetros. Um exemplo de uma requisição **GET** gerada por um cliente **HTTP** poderia ser:

```
GET http://www.dc.ufscar.br/
Accept: text/html, image/gif, image/jpeg
User-Agent: Mozilla/5.0 (X11; Linux x86\_64) Ubuntu Chromium/28.0.1500.71 Chrome/28.0.1500.71
```

Para a qual a resposta poderia ser:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 19 Oct 2013 12:04:55 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 131
Connection: close
<html>
<head>
  <title>Página de Teste</title>
</head>
<body>
  Olá, esta é uma página de teste.
</body>
</html>
```

A indicação do tipo de conteúdo do recurso (usada nos parâmetros **Accept** e **Content-Type**) segue a especificação no padrão MIME[2] (*Multipurpose Internet Mail Extensions*).

8.3.2 HTML

HTML é a sigla para *Hypertext Markup Language*, ou Linguagem de Marcação de Hypertexto, tradicionalmente utilizada para a criação de páginas Web. Os seus elementos servem para definir a formatação, apresentação, objetos (imagens, sons, etc.) e *links*.

A linguagem consiste em uma série de *tags* integradas em um documento texto. A sintaxe é simples e universal. Para visualizar documentos **HTML**, pode ser usado qualquer navegador Web, tais como o Chrome, Firefox, Internet Explorer ou o Safari.

Uma *tag* é uma etiqueta que executa uma função no documento. Toda página **HTML** deve ser iniciada pela *tag* **<html>** e terminada pela *tag* **</html>**.

[2] http://tools.ietf.org/html/rfc2045.

- **<html>** – define o início de um documento HTML e indica ao navegador que todo conteúdo posterior deve ser tratado como uma série de códigos HTML;
- **<head>** – define o cabeçalho de um documento HTML, que traz informações sobre o documento;
- **<body>** – define o conteúdo principal, o corpo do documento. Esta é a parte do documento HTML que é exibida no navegador.

```
<html>
<head>
  <title>Página de Teste</title>
</head>
<body>
  Olá, esta é uma página de teste.
</body>
</html>
```

Observe que as *tags* podem ser escritas tanto em minúsculas quanto em maiúsculas. O fechamento das *tags* sempre é realizado com uma barra "/" precedendo o seu nome.

Dentro do corpo, podemos encontrar outras *tags* que irão moldar a página, tais como:

- **<h1> <h2>** ... **<h6>** – cabeçalhos e títulos no documento em diversos tamanhos. Sendo **<h1>** o maior tamanho, e **<h6>** o menor tamanho;
- **<p>** – insere um novo parágrafo;
- **
** – quebra de linha;
- **<table>** – cria uma tabela (linhas são criadas com <TR>, e novas células com <TD>, já os cabeçalhos das colunas são criados com as etiquetas <TH>);
- **<div>** – determina uma divisão na página a qual pode possuir variadas formatações;
- **<b>**, **<i>**, **<u>** e **<s>** – negrito, itálico, sublinhado e riscado, respectivamente;
- **<img />** imagem;
- **<a>** – *link* para outro local, seja uma página, um e-mail ou outro serviço;

  Ex.: `<a href="http://www.dc.ufscar.br"> Departamento de Computação - UFSCar</a>`
- **<textarea>** – caixa de texto (com mais de uma linha).

Uma discussão extensiva da linguagem HTML encontra-se fora do escopo deste material. Porém, como as aplicações Web são construídas a partir dessa sintaxe, recomenda-se que o desenvolvedor Web tenha uma noção do seu funcionamento. Para um estudo mais aprofundado desse tópico, o leitor interessado pode consultar o seguinte link: http://www.w3schools.com/html/html5_intro.asp.

8.3.3 Páginas dinâmicas

Quando a Web surgiu, seu objetivo era a troca de conteúdos através, principalmente, de páginas HTML estáticas. Eram arquivos escritos no formato HTML e disponibilizados em servidores para serem acessados nos navegadores. Imagens, animações e outros conteúdos também eram disponibilizados.

Mas logo se viu que a Web tinha um enorme potencial de comunicação e interação além da exibição de simples conteúdos. Para atingir esse novo objetivo, porém, páginas estáticas não seriam suficientes. Era necessário tornar disponíveis páginas HTML dinamicamente geradas a partir das requisições dos usuários.

Atualmente, boa parte do que se acessa na Web (portais, *home bankings*, etc.) é baseada em conteúdo dinâmico. O usuário requisita algo ao servidor o qual, por sua vez, processa essa requisição e devolve uma resposta nova para o usuário.

Uma das primeiras ideias para esses "geradores dinâmicos" de páginas HTML foi fazer o servidor Web invocar outro programa externo em cada requisição para gerar o HTML de resposta. Era o famoso CGI, *Common Gateway Interface*, que permitia escrever pequenos programas para apresentar páginas dinâmicas usando, por exemplo, Perl, ASP e até C ou C++.

Na plataforma Java, a primeira e principal tecnologia capaz de gerar páginas dinâmicas são as *Servlets*, que surgiram no ano de 1997. A tecnologia *Servlet* será discutida na Seção 8.5.

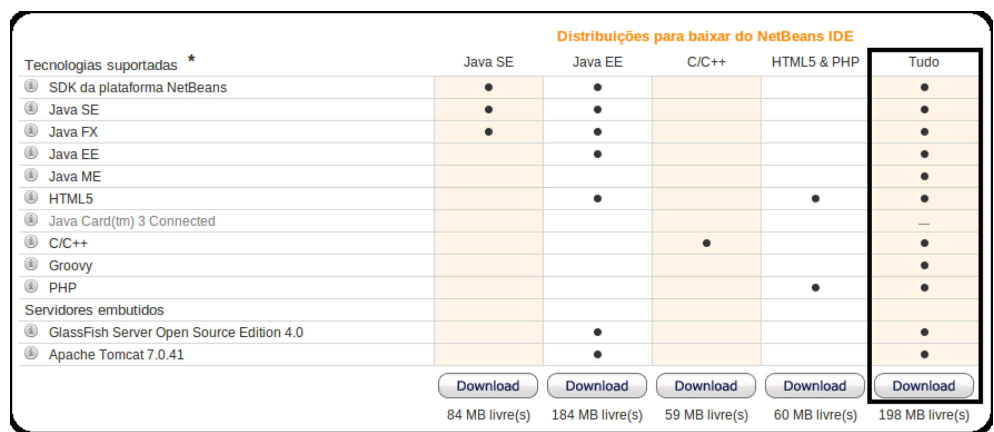
8.3.4 Configuração do ambiente de desenvolvimento

Esta seção apresenta algumas dicas importantes no processo de instalação e configuração do ambiente de desenvolvimento necessário para compilar e executar os exemplos discutidos nesta unidade. A instalação do ambiente é simples e consiste de poucos passos:

Instalação Java. O Java Development Kit (JDK) – versão igual ou superior a 1.5 – será necessário para executar os exemplos apresentados neste material. A última versão do JDK pode ser obtida em http://java.com/pt_BR/download/index.jsp (este material utiliza o JDK versão 1.7.0_65).

Instalação NetBeans. O NetBeans IDE será utilizado para executar os exemplos apresentados neste material, o qual utiliza a versão 7.4 do NetBeans IDE, que pode ser obtida em http://netbeans.org/downloads.

Dicas importantes: (1) faça o download da distribuição **Tudo** (Figura 8.1), pois é a única que vem com suporte ao desenvolvimento Web. (2) Para acessar um tutorial sobre NetBeans + Java no desenvolvimento de aplicações Web, sugere-se a consulta ao endereço https://netbeans.org/kb/trails/java-ee_pt_BR.html.



	Distribuições para baixar do NetBeans IDE				
Tecnologias suportadas *	Java SE	Java EE	C/C++	HTML5 & PHP	Tudo
SDK da plataforma NetBeans	•	•			•
Java SE	•	•			•
Java FX	•	•			•
Java EE		•			•
Java ME					•
HTML5		•		•	•
Java Card(tm) 3 Connected					—
C/C++			•		•
Groovy					•
PHP				•	•
Servidores embutidos					
GlassFish Server Open Source Edition 4.0		•			•
Apache Tomcat 7.0.41		•			•
	Download	Download	Download	Download	Download
	84 MB livre(s)	184 MB livre(s)	59 MB livre(s)	60 MB livre(s)	198 MB livre(s)

Figura 8.1 Distribuições do NetBeans IDE 7.4.

Observação: o NetBeans IDE será utilizado neste material, mas o leitor deve se sentir à vontade para usar outro IDE.

8.4 *Applets*

Applets são programas projetados para ter uma execução independente dentro de alguma outra aplicação, eventualmente interagindo com esta – tipicamente, um navegador Web. Assim, *applets* executam no contexto de outro programa, o qual interage com o *applet* e determina assim sua sequência de execução. Funcionalidades associadas a *applets* Java são agregadas no pacote `java.applet`.

A classe `Applet` é uma extensão da classe `Panel` (Figura 6.1). Assim, o desenvolvimento de um *applet* segue as estratégias de desenvolvimento de qualquer aplicação gráfica. Ou seja, o processo de criação de um *applet* é similar ao processo de criação de uma aplicação `AWT` qualquer – o programa deve ser criado por um editor de texto (ou IDE), e o arquivo de *bytecodes* (com extensão `.class`) deve ser gerado a partir da compilação do arquivo fonte.

Uma vez criado ou disponibilizado o *bytecode*, é preciso incluir uma referência a ele em alguma página Web. Essa página Web, uma vez carregada em algum navegador Web, irá reconhecer o elemento que faz a referência ao *applet*, transferir (*download*) seu *bytecode* para a máquina local e dar início à sua execução.

Como *applets* são programas que podem ser carregados de páginas na internet para ser executado localmente, é necessário assegurar restrições de segurança na execução de *applets*. Ou seja, é fundamental que o usuário esteja protegido contra programas maliciosos que tentam, por exemplo, ler um dado confidencial (número de cartão de crédito ou uma senha) para enviá-lo pela internet. Uma maneira de Java prevenir esses acessos ilegais é fornecendo um gerenciador de segurança (classe `SecurityManager`). Isso apenas é possível pelo fato de o código Java ser interpretado pela Máquina Virtual Java e não diretamente executado pela CPU. Esse modelo é chamado de modelo de segurança *sandbox*. Quando uma das restrições de segurança do Java é violada, o gerenciador de segurança do *applet* gera uma exceção do tipo `SecurityException`.

O nível de controle de segurança é implementado no navegador Web. A maioria dos navegadores previne as seguintes ações:

- Executar outro programa ou métodos nativos;
- Leitura ou escrita de arquivos na máquina local;
- Abrir uma conexão a uma máquina que não seja a hospedeira do *applet*.

8.4.1 Ciclo de vida de um *applet*

Ao contrário das aplicações Java tradicionais, a execução de um *applet* em uma página Web não é iniciada pelo método **main()**. Um *applet* é executado como uma *thread* subordinada ao navegador Web (ou à aplicação **appletviewer**, presente no ambiente de desenvolvimento Java). O navegador será responsável por invocar os métodos da classe **Applet** que controlam a execução de um *applet* (Figura 8.2).

- **init()** – após a execução do construtor da classe, o navegador Web invoca o método **init()**. Esse método é chamado apenas na primeira vez em que o *applet* é iniciado. Geralmente, esse método é usado para: (1) ler parâmetros da página **HTML**, através do método **getParameter()**; (2) adicionar componentes de interface (*GUI*); e (3) definir um novo gerenciador de leiaute.

- **start()** – após a execução do método **init()**, é chamado o método **start()**. Esse método também é chamado toda vez que o *applet* se torna visível, dando reinício às operações que eventualmente tenham sido paralisadas pelo método **stop()**.

- **stop()** – esse método é chamado toda vez que o *applet* deixar de ser visível, por exemplo, quando é minimizado ou quando outra página é chamada. É uma forma de evitar que operações as quais demandam muitos ciclos de CPU continuem a executar desnecessariamente. O método **stop()** também é chamado imediatamente antes do método **destroy()** descrito a seguir.

- **destroy()** – esse método é chamado quando o *applet* terminar a sua execução, liberando todos os recursos alocados, por exemplo, quando o navegador Web finaliza a sua execução.

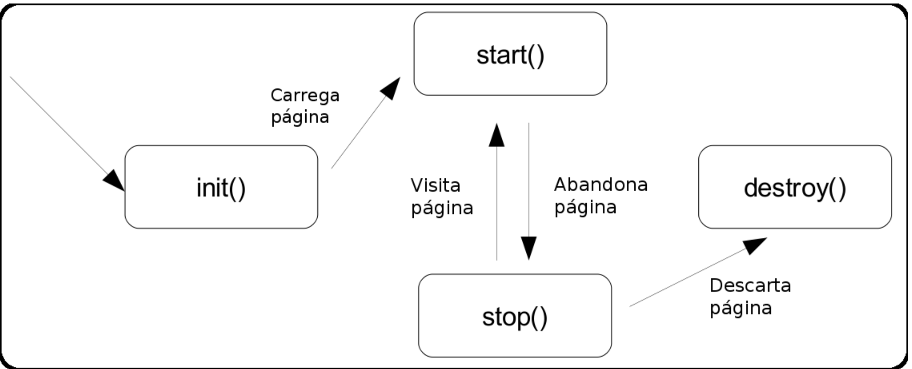


Figura 8.2 Ciclo de vida de um *applet*.

8.4.2 O primeiro exemplo de um *applet*

Nesta seção é apresentado o primeiro exemplo de *applet* que provê uma funcionalidade similar à provida pela aplicação gráfica **AWT**, apresentada na Figura 6.10 e discutida na Seção 6.3.4.

Ou seja, o **AppletDemo** (Código 8.1) utiliza dois botões **AWT** e um campo de texto não editável pelo usuário. Análogas à aplicação discutida na Seção 6.3.4, nesse *applet* as alterações no conteúdo do campo de texto apenas acontecem em resposta a eventos nos dois botões. Os dois botões são associados à classe **AppletDemo**, que implementa a interface **ActionListener** (método **actionPerformed()**) – a cada clique no botão **b1 (-)** ou no botão **b2 (+)**, o conteúdo do campo texto é decrementado ou incrementado em 1.

```java
/**
 * Classe AppletDemo - exemplo de um applet AWT
 *
 * @author Delano Medeiros Beder
 */
public class AppletDemo extends Applet implements ActionListener {
    private Button b1;
    private Button b2;
    private TextField tValue;

    public void init() {
        b1 = new Button("-");
        b2 = new Button("+");
        tValue = new TextField("0");
        tValue.setEditable(false);
        b1.addActionListener(this);
        b2.addActionListener(this);
        this.setLayout(new FlowLayout());
        this.add(b1);
        this.add(tValue);
        this.add(b2);
        String color = this.getParameter("color");
        if (color == null) {
            color = "0xC0C0C0";
        }
        this.setBackground(Color.decode(color));
    }

    public void start() {
        tValue.setText("0");
    }

    public void actionPerformed(ActionEvent e) {
        String command = e.getActionCommand();
        Integer valor = new Integer(tValue.getText());
        if (command.equals("+")) {
            valor = valor + 1;
        } else {
            valor = valor - 1;
        }
        tValue.setText(valor.toString());
    }
}
```

Código 8.1 Classe AppletDemo.

Alguns métodos da classe **Applet** permitem a comunicação do *applet* com o navegador Web no qual ele está inserido. O método **getAppletContext()** permite obter o contexto de execução do *applet*, o que permite, por exemplo, estabelecer a comunicação entre dois *applets* de uma mesma página. O método **getParameter()** permite a obtenção dos parâmetros passados pelo navegador Web para o *applet*. Nesse exemplo, o método **getParameter()** é utilizado para obter a cor do fundo (*background*) do *applet*. A cor é uma *string* que representa um inteiro de 24 bits (padrão **RGB**), sendo cada 8 bits (**R**, **G** ou **B**) representado por dois dígitos hexadecimais (na faixa de 0 a F). A Figura 8.3 apresenta a execução do **AppletDemo** em uma página Web.

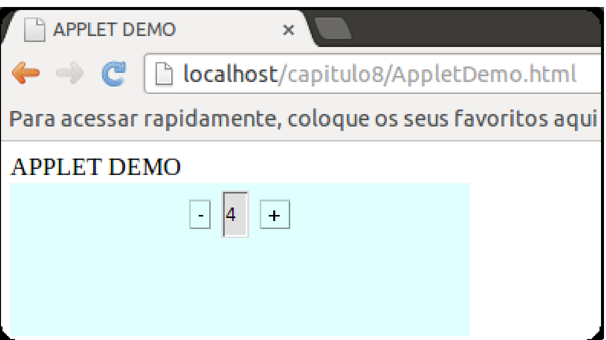


Figura 8.3 Execução do **AppletDemo** no navegador Web.

8.4.3 Carregando um *applet*

Como um *applet* é executado em um navegador Web, ele não é inicializado diretamente por uma linha de comando. Para executar um *applet*, deve-se criar um arquivo **HTML** especificando o *applet* que deve ser carregado. Existe uma *tag* **<applet>** para especificar a chamada de um *applet*, em que:

- **code** – nome do arquivo **.class** que é subclasse da classe **Applet**.
- **archive** – contém a lista de classes (.class) e outros recursos (por exemplo, imagem, som) separados por ",". Ao invés disso, é possível conter o nome de um arquivo compactado (**.jar**) com todos os recursos necessários.
- **width** e **height** – especificam a altura e a largura da área de exibição do *applet*.
- **<param name="nome_atributo" value="valor_atributo">** – especifica parâmetros que serão lidos no *applet* pelo método **getParameter()** no método **init()** do *applet*.

O Código 8.2 apresenta a página **HTML** que executa o **AppletDemo** (Figura 8.3). Note que o parâmetro **color** é passado para o *applet*.

```
<html>
    <head>
        <title>APPLET DEMO</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <div>APPLET DEMO</div>
    <applet code = 'br.ufscar.si.poo2.applet.AppletDemo'
            archive = 'dist/capitulo8.jar'
            width = 300
            height = 100>
        <param name="color" value="0xE0FFFF"/>
    </applet>
</body>
</html>
```

Código 8.2 `AppletDemo.html`.

8.4.4 Classe **JApplet**

Se há a necessidade de utilizar elementos de interface gráfica presentes no pacote **Swing** em um *applet*, este deverá ser uma instância da classe **JApplet** (ou de uma de suas subclasses), caso contrário não funcionará corretamente. A classe **JApplet** é subclasse da classe **Applet**. O ciclo de vida da classe **JApplet** é similar a da classe **Applet**, por isso não será revisado nesta seção. Para mais informações sobre o ciclo de vida da classe **Applet**, consulte a Seção 8.4.1.

Nesta seção é apresentada a classe **JAppletDemo** – um exemplo de *applet* **Swing** que provê a mesma funcionalidade da aplicação **Swing** apresentada na Figura 6.16 e discutida na Seção 6.4.1. A Figura 8.4 apresenta a execução do **JAppletDemo** em uma página Web.

Figura 8.4 Execução do **JAppletDemo** no navegador Web.

A classe **JAppletDemo** (Código 8.3) utiliza dois rótulos (objetos **JLabel**), uma área de texto (objeto **JTextArea**) e um campo de texto (objeto **JTextField**) não editável. De forma análoga à aplicação apresentada na Figura 6.16, a área de texto é associada a uma instância de uma classe que implementa a interface **KeyListener** – no caso desse exemplo, a própria classe **JAppletDemo** implementa a interface **KeyListener**. Como resultado dessa associação, a cada tecla digitada, o número de caracteres presente na área de texto (**texto**) é contabilizado, e esse valor é apresentado no campo de texto (**numCaracteres**).

O método **KeyReleased()** da interface **KeyListener** é responsável por tratar o evento que ocorre quando uma tecla é liberada, após ter sido pressionada, na área de texto. Esse evento é apenas tratado para contabilizar o número de caracteres presentes na área de texto e atualizar o conteúdo do campo de texto com o valor computado. Observe que os demais eventos da interface **KeyListener** (métodos **keyPressed()** e **KeyTyped()**) são ignorados – implementação vazia.

```java
/**
 * Classe JAppletDemo – exemplo de um applet Swing
 *
 * @author Delano Medeiros Beder
 */
public class JAppletDemo extends JApplet implements KeyListener {
    private JLabel textoLbl;
    private JTextArea texto;
    private JLabel numLbl;
    private JTextField numCaracteres;

    public void init() {
        textoLbl = new JLabel("Texto:");
        texto = new JTextArea();
        numLbl = new JLabel("Caracteres:");
        numCaracteres = new JTextField("0");
        numCaracteres.setEditable(false);
        texto.addKeyListener(this);
        this.setLayout(new GridLayout(2, 2));
        this.getContentPane().add(textoLbl);
        this.getContentPane().add(texto);
        this.getContentPane().add(numLbl);
        this.getContentPane().add(numCaracteres);
    }

    public void keyPressed(KeyEvent e) { // Não faz nada }

    public void keyReleased(KeyEvent e) {
        JTextArea area = ((JTextArea) e.getComponent());
        numCaracteres.setText(Integer.toString(area.getText().length()));
    }

    public void keyTyped(KeyEvent e) { // Não faz nada }
}
```

Código 8.3 Classe JAppletDemo.

8.5 *Servlets*

Um *servlet* é um programa Java que segue a especificação *API* **Java Servlet** (do pacote **javax.servlet**) a qual proporciona ao desenvolvedor a possibilidade de adicionar conteúdo dinâmico em um servidor Web usando a plataforma Java. Ou seja, *Servlets* são responsáveis por estender as capacidades de um servidor Web, de forma similar à extensão que os *applets* (Seção 8.4) proporcionam aos navegadores Web (clientes). Para propósitos deste material, serão considerados apenas *servlets* que dão suporte ao protocolo `HTTP`, o mais utilizado na Web.

O nome "*servlet*" vem da ideia de um pequeno servidor cujo objetivo é receber chamadas `HTTP`, processá-las e devolver uma resposta ao cliente (CAELUM, 2014b). *Servlets* dão suporte à geração de conteúdo dinâmico nas páginas Web, acesso a banco de dados, atendimento às múltiplas requisições de clientes e de dados.

Para que *servlets* sejam executados, é necessária a utilização de um *container* de *servlets*, que é um componente de um servidor Web o qual interage com *Servlets* Java. Como exemplo de *container* de *servlets*, pode-se citar o Tomcat[3], que é um software livre desenvolvido pela **Apache Software Foundation**, sendo oficialmente endossado pela Sun™ como a implementação de referência para as tecnologias *Java Servlet* e *Java Server Pages* (*JSP*). Ele tem a capacidade de atuar também como servidor Web, ou pode funcionar integrado a um servidor Web dedicado, como o servidor Apache. Como servidor Web, ele provê um servidor Web `HTTP` puramente implementado em Java.

8.5.1 Classe `HttpServlet`

A classe **`HttpServlet`** é subclasse da classe **`GenericServlet`** (ambas são abstratas). O programador deve estender a classe **`HttpServlet`** e reimplementar pelo menos um de seus métodos para implementar um *servlet* `HTTP` apropriado para um sistema Web. O principal método dessa classe é o **`service()`**, que recebe as requisições `HTTP` e é responsável por despachá-las para o método apropriado do *servlet*. Por exemplo, caso a requisição seja um `POST`, o método **`doPost()`** da classe **`HttpServlet`** será invocado. Dessa forma, o programador deve reimplementar o método **`doPost()`** para atribuir ao *servlet* o comportamento desejado para esse tipo de requisição.

[3] http://tomcat.apache.org.

A classe **HttpServlet** fornece métodos que tratam cada requisição **HTTP**. São eles: **doDelete()**, **doGet()**, **doHead()**, **doOptions()**, **doPost()**, **doPut()** e **doTrace()**.

A classe possui os métodos herdados da classe **GenericServlet**, como os métodos **init()** e **destroy()**, responsáveis pela inicialização e finalização do *servlet*, respectivamente. Esses dois métodos serão discutidos na Seção 8.5.2.

Para mais detalhes sobre a classe **HttpServlet**, o leitor interessado pode consultar o seguinte link: http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServlet.html.

8.5.2 Ciclo de vida de um *servlet*

A Figura 8.5 ilustra o ciclo de vida dos *servlets*, representados pelos métodos **init()**, **service()** e **destroy()** da interface **Servlet**, que a classe **GenericServlet** implementa.

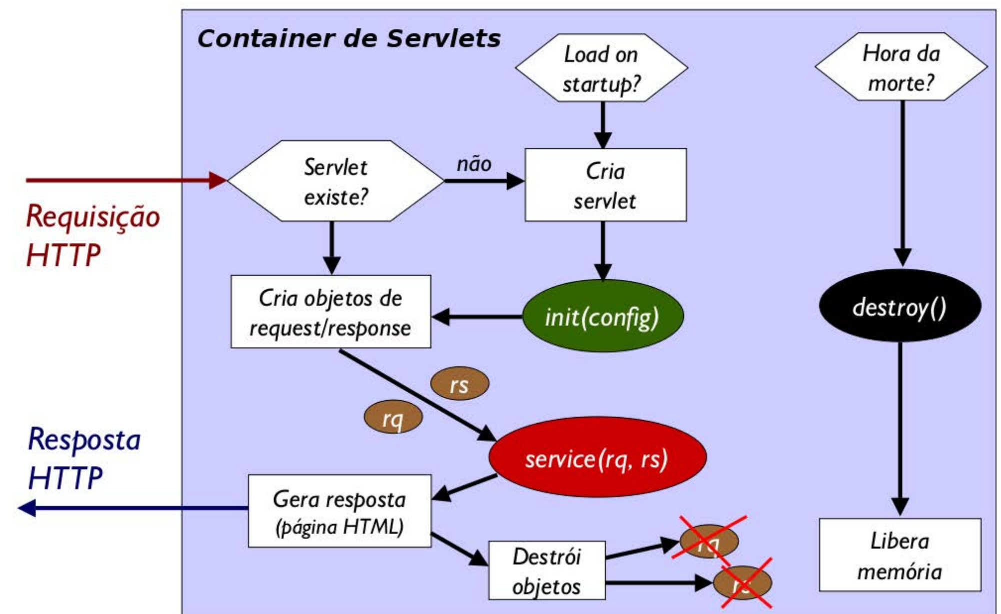


Figura 8.5 Ciclo de vida dos *servlets*.

A classe do *servlet* deve ser carregada pelo *container* de *servlets*. Em seguida, o *container* de *servlets* instancia a classe. Antes de receber requisições do cliente, o objeto instanciado pelo *container* de *servlets* deve ser inicializado por meio do método **init()**. Essa etapa normalmente é utilizada para iniciar conexões de banco de dados e inicializar variáveis para os seus valores padrão.

A partir do momento que um *servlet* é inicializado, o *container* de *servlets* pode utilizá-lo para tratar requisições dos clientes. Essas requisições são instâncias de classes que implementam a interface `ServletRequest` (ou a interface `HttpServletRequest`, no caso de requisições `HTTP`), que são passadas como parâmetros para o método `service()`. Instâncias de classes que implementam a interface `ServletResponse` representam as respostas enviadas para o cliente para cada requisição. No caso de requisições `HTTP`, são instâncias de classes que implementam a interface `HTTPServletResponse`.

Finalmente, o *container* de *servlets* pode determinar que o *servlet* deve ser removido, e para isso invoca o método `destroy()`, que permite a finalização de recursos alocados (conexões de banco de dados, etc.) pelo *servlet*. Após a chamada desse método, qualquer requisição feita ao *container* de *servlets* não pode ser mais tratada por esse *servlet*, pois os seus recursos já foram liberados.

8.5.3 Interface `HttpServletRequest`

A interface `HttpServletRequest` herda da interface `ServletRequest` todos os métodos para lidar com os parâmetros de uma requisição `HTTP`.

- O método `String getParameter(String name)` retorna o valor de um parâmetro da requisição identificado pelo seu nome. Caso o parâmetro possua mais de um valor, será retornado apenas o primeiro deles, sendo preferido o uso do método `getParameterValues()` nesse caso.

- O método `String[] getParameterValues(String name)` retorna um *array* de *strings* que representa os valores de um parâmetro. Observe que o tamanho do *array* apenas será maior que um caso existam vários parâmetros com o mesmo nome.

- O método `Enumeration getParameterNames()` retorna uma enumeração com todos os nomes de parâmetros recebidos na requisição (ou vazia, caso não tenha nenhum parâmetro).

- Finalmente, o método `Map getParameterMap()`, que é muito útil para a manipulação de parâmetros da requisição, pois retorna um objeto `Map` (Seção 5.8) contendo o nome dos parâmetros como as chaves do mapa (`String`), e os seus valores como um *array* de *strings* (`String[]`).

Além de lidar com parâmetros, a interface `HttpServletRequest` também possui métodos para obter os valores de cabeçalhos e *cookies*. Para mais detalhes sobre os métodos da classe `HttpServletRequest`, o leitor interessado pode consultar o seguinte link: http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequest.html.

8.5.4 Atributos

Atributos são objetos associados a uma requisição, sessão ou ao contexto da aplicação, que expressam informações utilizadas pelo *container*, que não podem ser publicadas de outras formas. O objetivo prático no uso de atributos é o compartilhamento de informações entre uma requisição e outra (no caso de atributos de requisição), entre uma mesma (para atributos de sessão) ou mesmo entre sessões distintas (atributos de contexto).

- **Atributos de requisição** – duram apenas o tempo de uma requisição. Ao término de uma requisição, todos os atributos de requisição são destruídos.

- **Atributos de sessão** – quando um usuário acessa o sistema Web, ele estabelece com o servidor uma sessão. Os atributos de sessão existem desde o instante inicial, quando o usuário acessa a aplicação pela primeira vez, até que ela expire por inatividade, seja voluntariamente ou finalizada pela aplicação.

- **Atributos de contexto** – duram desde a inicialização do *container* de *servlets* até que ele seja finalizado.

Atributos podem ser acessados por meio de métodos da interface `ServletRequest` para atributos de requisição, `HttpSession` para atributos de sessão, e `ServletContext` para atributos de contexto. Para obter um objeto `HttpSession` ou `ServletContext`, utiliza-se, respectivamente, o método `getSession()` ou `getServletContext()` da interface `HttpServletRequest`.

Os métodos que manipulam atributos possuem a mesma assinatura nas três classes – `ServleRequest`, `HttpSession` e `ServletContext`:

- `Object getAtribute(String name)` – retorna um objeto representando o atributo com o nome especificado;

- **Enumeration<String> getAttributeNames()** – retorna uma enumeração com cada um dos nomes dos atributos associados à requisição, à sessão ou ao contexto;

- **void setAttribute(String name, Object o)** – recebe como parâmetro o nome do atributo a ser armazenado e o objeto representando o atributo. Esse é o método utilizado efetivamente para associar um objeto com o escopo desejado (requisição, sessão ou contexto);

- **void removeAttribute(String name)** – remove um objeto representando o atributo com o nome especificado.

8.5.5 Interface **HttpServletResponse**

A interface **HttpServletResponse**, que estende a interface **ServletResponse**, é utilizada para construir a resposta para o cliente que enviou a requisição **HTTP**. Antes de escrever a resposta ao cliente, deve-se definir a codificação de caracteres e o tipo de conteúdo que será utilizado para compor o corpo da mensagem – tarefas que podem ser realizadas através dos métodos **setCharacterEncoding()** e **setContentType()** da interface **HttpServletResponse**.

Além desses métodos que servem para formatar o conteúdo da mensagem, existem dois métodos que retornam uma *stream* (Unidade 4) utilizada para enviar os dados de resposta para o cliente. São os métodos **PrintWriter getWriter()** e **ServletOutputStream getOutputStream()**, responsáveis por enviar dados em formato de caracteres (texto) e binário, respectivamente. Tanto a classe **PrintWriter** quanto a classe **ServletOutputStream** possuem o método **flush()**, que deve ser invocado para efetivar e enviar a resposta.

E, por fim, a interface **HttpServletResponse** disponibiliza o método **sendRedirect()** que pode redirecionar o cliente para outra página Web. Outro método utilizado para retornar para o cliente é o método **sendError()**, que envia uma resposta de erro contendo um código de *status* de erro do protocolo **HTTP**. Os códigos de *status* de erros do protocolo **HTTP** podem ser consultados no seguinte link: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

Para saber mais detalhes sobre os métodos da classe **HttpServletResponse**, o leitor interessado pode consultar o seguinte link:

http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletResponse.html.

8.5.6 Exemplo: Conversão de temperaturas

Nesta seção é apresentado um exemplo de uma aplicação Web simples que provê a funcionalidade de conversão de temperaturas. Ou seja, dadas a temperatura inicial e final em graus *Celsius*, a aplicação Web apresenta uma tabela dessas temperaturas convertidas para graus *Fahrenheit* ou graus *Kelvin*. Essa aplicação foi desenvolvida utilizando a tecnologia Servlet Java, e seus principais componentes são discutidos nas próximas subseções.

8.5.6.1 Arquivo `index.html`

Esse arquivo `HTML` (Código 8.4) é a página inicial da aplicação e possui um formulário `HTML` que deve ser preenchido e posteriormente submetido pelo usuário da aplicação. As informações a serem preenchidas são: (1) tipo de conversão (Celsius-Fahrenheit ou Celsius-Kelvin); (2) temperatura inicial em graus Celsius; (3) temperatura final em graus Celsius; e (4) variação da temperatura. Observe que o formulário invoca um *servlet* (`action="conversão"`) cujo apelido é **conversão** (ver discussão sobre a configuração dos *servlets* da aplicação na Seção 8.5.6.2).

```
<html>
    <head>
        <meta content="text/html; charset=utf-8" http-equiv="Content-Type">
        <title>Celsius-Fahrenheit</title>
    </head>
    <body>
        <form name="entry" method="post" action="conversão">
            Conversão:
            <input type ="radio" name="tipo" value="0" checked="checked">Fahrenheit
            <input type ="radio" name="tipo" value="1">Kelvin
            <br/> <br/>
            Inicial (<code>&deg;</code>C):
            <input type ="text" name="inicial">
            <br/> <br/>
            Final   (<code>&deg;</code>C):
            <input type ="text" name="final">
            <br/> <br/>
            Variação:
            <input type ="radio" name="var" value="5" checked="checked">5
            <input type ="radio" name="var" value="10">10
            <br/><br/>
            <input type="reset" name="reset" value="Inicializa Valores">
            <input type="submit" name="submit" value="Solicita a Conversão">
        </form>
    </body>
</html>
```

Código 8.4 Arquivo `index.html`.

8.5.6.2 Arquivo `web.xml`

Esse arquivo encontra-se presente em todas as aplicações Web na plataforma Java. É através da leitura desse arquivo que o *container* de *servlets* fica sabendo quais os níveis de segurança ele deve utilizar, qual o tempo de vida dos *servlets* e até mesmo qual será a página inicial da aplicação.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml
    /ns/j2ee/web-app_2_4.xsd">

    <servlet>
        <servlet-name>conversão</servlet-name>
        <servlet-class>servlet.ServletConversão</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>fahrenheit</servlet-name>
        <servlet-class>servlet.ServletCelsiusFahrenheit</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>kelvin</servlet-name>
        <servlet-class>servlet.ServletCelsiusKelvin</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>conversão</servlet-name>
        <url-pattern>/conversão</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>fahrenheit</servlet-name>
        <url-pattern>/fahrenheit</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>kelvin</servlet-name>
        <url-pattern>/kelvin</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>10</session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>
</web-app>
```

Código 8.5 Arquivo `web.xml`.

As *tags* **`<servlet>`** e **`</servlet>`** indicam que entre elas se encontra a configuração de um *servlet*. A *tag* **`<servlet-name>`** contém o apelido dado ao *servlet*. A *tag* <servlet-class> indica a classe que implementa o *servlet*. Note que na aplicação de conversão de temperaturas foram configurados três *servlets*, os quais serão discutidos adiante nesta seção.

As *tags* **`<servlet-mapping>`** e **`</servlet-mapping>`** indicam que entre elas se encontra um mapeamento entre um *servlet* e uma `URL`. A *tag* **`<url-pattern>`** indica qual `URL` deve ser informada para ativar um *servlet*. A *tag* **`<servlet-name>`** indica o nome do *servlet* que deve ser invocado. Aqui se deve informar o apelido de

algum *servlet* mapeado no passo anterior. Note que na aplicação de conversão de temperaturas os três *servlets* configurados anteriormente foram mapeados para três **URLs**.

É possível também determinar o tempo de vida das sessões de sua aplicação, muito útil quando se tem uma aplicação que transporta na sessão dados importantes. No caso da aplicação de conversão de temperaturas, o tempo das sessões está configurado em 10 minutos. Ou seja, se a aplicação ficar parada por 10 minutos, as sessões são eliminadas na memória.

Outra configuração interessante é a dos arquivos iniciais. No caso da aplicação de conversão de temperaturas, o arquivo inicial é o arquivo **index.html**, discutido anteriormente.

Para saber mais detalhes sobre o arquivo **web.xml**, o leitor interessado pode consultar o seguinte link:

http://docs.oracle.com/cd/E13222_01/wls/docs92/webapp/configureservlet.html.

8.5.6.3 Classe **ServletConversão**

A classe **ServletConversão** é o *servlet* invocado após a submissão do formulário **HTML** presente no arquivo **index.html**, pois, conforme configurado no arquivo **web.xml**, esse *servlet* está mapeado para a **URL** **conversão**, que deve ser invocada após a submissão do formulário (**action="conversão"**).

Conforme se pode observar pelo Código 8.6, esse *servlet* implementa os métodos **doGet()** e **doPost()**, cuja função é simplesmente delegar a execução para o método **processRequest()**. Dessa forma, esse *servlet* possui o mesmo comportamento para requisições **POST** e **GET**.

Vale a pena salientar os seguintes aspectos da implementação do método **processRequest()**:

- Os valores dos parâmetros **inicial**, **final**, **var** e **tipo**, que foram passados ao *servlet*, são obtidos nas linhas 19, 21, 23 e 25. Os valores dos três primeiros parâmetros são armazenados como atributos de sessão nas linhas 20, 22 e 24;

- Se algum dos valores é inválido (por exemplo, parâmetro vazio), a exceção **NumberFormatException** é levantada, e o tratamento dessa exceção consiste em redirecionar, através da invocação do método **sendRedirect()**, o cliente

para uma página de erro (arquivo `erro.html`), conforme apresentado na Figura 8.6.

Figura 8.6 Invocação do *servlet* com dados inválidos.

- Conforme se pode observar, esse *servlet* é simplesmente um roteador de requisições. Ou seja, o valor do parâmetro `tipo` é levado em consideração na escolha de para qual *servlet* a requisição deve ser redirecionada – `ServletCelsiusFahrenheit`, se valor igual a **0**, e `ServletCelsiusKelvin`, caso contrário;

```
1  /**
2   * Classe ServletConversão — Servlet invocado pelo index.html. Roteador para (C -> F) ou (C -> K)
3   *
4   * @author Delano Medeiros Beder
5   */
6  public class ServletConversão extends HttpServlet {
7      protected void doGet(HttpServletRequest request, HttpServletResponse response)
8                  throws ServletException, IOException {
9          processRequest(request, response);
10     }
11     protected void doPost(HttpServletRequest request, HttpServletResponse response)
12                 throws ServletException, IOException {
13         processRequest(request, response);
14     }
15     protected void processRequest(HttpServletRequest request, HttpServletResponse response)
16                 throws ServletException, IOException {
17         try {
18             HttpSession session = request.getSession();
19             String iniValue = request.getParameter("inicial");
20             session.setAttribute("inicial", Integer.parseInt(iniValue));
21             String finValue = request.getParameter("final");
22             session.setAttribute("final", Integer.parseInt(finValue));
23             String varValue = request.getParameter("var");
24             session.setAttribute("var", Integer.parseInt(varValue));
25             String tipo = Integer.parseInt(request.getParameter("tipo"));
26             switch (tipo) {
27                 case 0: { response.sendRedirect("fahrenheit"); break; }
28                 case 1: { response.sendRedirect("kelvin"); break; }
29             }
30         } catch (NumberFormatException exc) {
31             response.sendRedirect("erro.html");
32         }
33     }
34 }
```

Código 8.6 Classe `ServletConversão`.

8.5.6.4 Classe **`ServletCelsiusFahrenheit`**

A classe **`ServletCelsiusFahrenheit`** (Código 8.7) é subclasse da classe **`ServletConversão`** e, portanto, herda os métodos **`doGet()`** e **`doPost()`**, que delegam ao método **`processRequest()`** a responsabilidade de tratar requisições **`GET`** e **`POST`**.

```java
/**
 * Classe ServletCelsiusFahrenheit -- Constrói uma tabela  de conversão de temperaturas (C -> F)
 *
 * @author Delano Medeiros Beder
 */
public class ServletCelsiusFahrenheit extends ServletConversão {
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException {
        int iniValue = (Integer)request.getSession().getAttribute("inicial");
        int finValue = (Integer)request.getSession().getAttribute("final");
        int varValue = (Integer)request.getSession().getAttribute("var");
        double fahr;
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html> <head> <title>Celsius-Fahrenheit</title> </head>");
        out.println("<body> <table border = \"1\">");
        out.println("<tr> <th> Celsius (<code>&deg;</code>C) </th>");
        out.println("<th> Fahrenheit (<code>&deg;</code>F) </th> </tr>");
        for (int celsius = iniValue; celsius <= finValue; celsius += varValue) {
            fahr = 9.0 / 5.0 * celsius + 32;
            out.println("<tr> <th>" + celsius + "</th>");
            out.println("<th>" + fahr + "</th> </tr>");
        }
        out.println("</table> </body> </html>");
        out.close();
    }
}
```

Código 8.7 Classe **`ServletCelsiusFahrenheit`**.

No caso da classe **`ServletCelsiusFahrenheit`**, o método **`processRequest()`** simplesmente constrói dinamicamente e retorna uma página **`HTML`** que contém uma tabela de conversão de temperaturas (°C para °F). Note que esse método recupera os valores dos atributos de sessão **`inicial`**, **`final`** e **`var`**. Um exemplo de página **`HTML`** gerada dinamicamente é apresentada na Figura 8.7.

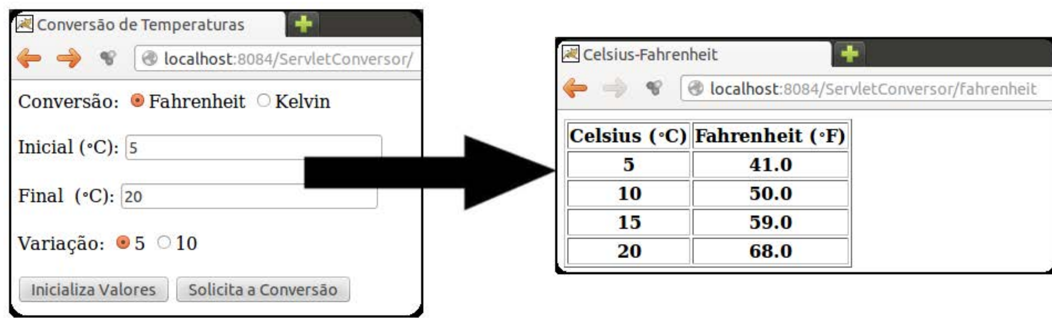


Figura 8.7 Conversão de temperaturas (°C para °F).

8.5.6.5 Classe `ServletCelsiusKelvin`

Analogamente, a classe **`ServletCelsiusKelvin`** (Código 8.8) também é subclasse de **`ServletConversão`** e, portanto, herda os métodos **`doGet()`** e **`doPost()`**, que delegam ao método **`processRequest()`** a responsabilidade de tratar requisições **`GET`** e **`POST`**.

```java
/**
 * Classe ServletCelsiusKelvin -- Constrói uma tabela  de conversão de temperaturas (C -> K)
 *
 * @author Delano Medeiros Beder
 */
public class ServletCelsiusKelvin extends HttpServlet {
    private void processRequest(HttpServletRequest request, HttpServletResponse response) throws
         ServletException, IOException {
        int iniValue = (Integer)request.getSession().getAttribute("inicial");
        int finValue = (Integer)request.getSession().getAttribute("final");
        int varValue = (Integer)request.getSession().getAttribute("var");
        double kelvin;
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html> <head> <title>Celsius-Kelvin</title> </head>");
        out.println("<body> <table border = \"1\">");
        out.println("<tr> <th> Celsius (<code>&deg;</code>C) </th>");
        out.println("<th> Kelvin (<code>&deg;</code>K) </th> </tr>");
        for (int celsius = iniValue; celsius <= finValue; celsius += varValue) {
            kelvin = celsius + 273.15;
            out.println("<tr> <th>" + celsius + "</th>");
            out.println("<th>" + kelvin + "</th> </tr>");
        }
        out.println("</table> </body> </html>");
        out.close();
    }
}
```

Código 8.8 Classe `ServletCelsiusKelvin`.

No caso da classe **`ServletCelsiusKelvin`**, o método **`processRequest()`** simplesmente constrói dinamicamente e retorna uma página **`HTML`** que contém uma tabela de conversão de temperaturas ($°\mathrm{C}$ para $°\mathrm{K}$). Note que esse método recupera os valores dos atributos de sessão **`inicial`**, **`final`** e **`var`**. Um exemplo de página **`HTML`** gerada dinamicamente é apresentada na Figura 8.8.

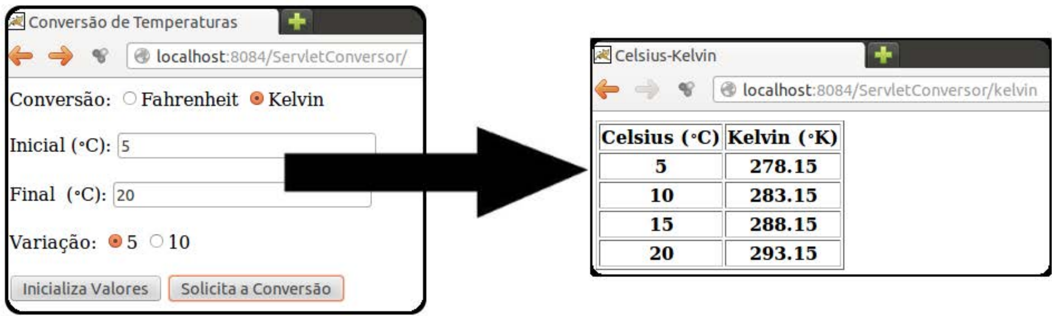


Figura 8.8 Conversão de temperaturas ($°\mathrm{C}$ para $°\mathrm{K}$).

8.6 *Java Server Pages*

A tecnologia de *Java Server Pages (JSP)* é uma extensão da tecnologia *Servlet* Java – as páginas *JSP* são internamente convertidas, pelo *container JSP*, em *servlets*. O principal objetivo das páginas *JSP* é gerar conteúdo dinâmico a partir do servidor Web, algo que não pode ser realizado utilizando apenas páginas `HTML`.

Uma página *JSP* utiliza a mesma sintaxe das páginas `HTML`, incluindo trechos estáticos, formatação, tabelas, etc. As *tags* são declaradas da mesma forma, ou seja, delimitadas pelos símbolos **<** e **>**. A diferença é que existem *tags* para os elementos *JSP* que são interpretadas e processadas pelo *container JSP*. O exemplo abaixo conta com *tags* `HTML` tais como `<html>` e `<body>`, além do texto estático **Olá**. Porém, as páginas *JSP* permitem a inclusão de código Java por meio de *scripts*, que são exemplificados mediante o *scriplet* que declara a variável **s**, e a expressão delimitada por **<%=** e **%>**.

```
<html>
<body>
<% String s = "Mundo"; %>
Olá <%= s %>
</body>
</html>
```

Comentários. Comentários *JSP* são completamente ignorados pelo tradutor da página e não aparecerão para o cliente que faz a requisição da página.

```
<%-- Comentário JSP --%>
<!-- Comentário HTML -->
```

Qualquer código válido da linguagem Java pode ser embutido em um *scriptlet*. O código abaixo declara uma variável `double`, invoca o método `sqrt()` da classe `Math` e utiliza o objeto implícito `out` para imprimir o valor na saída do *JSP*. Objetos implícitos serão discutidos na Seção 8.6.2.

```
<% double d = Math.sqrt(23);
out.println(d); %>
```

Outra maneira de imprimir é por meio de expressões. Estas são convertidas para *strings* e inseridas na posição apropriada. Note que, ao contrário de *scriptlets*, as expressões **não** devem terminar com ponto e vírgula (;).

```
<%= d %>
<%= new Java.util.Date() %>
```

8.6.1 Ciclo de vida de um *JSP*

A Figura 8.9 ilustra o ciclo de vida das páginas *JSP*. Uma página *JSP* tem um ciclo de vida bem parecido com o de um *servlet*. Como as páginas *JSP* são internamente convertidas em um *servlet*, a semelhança é justificável. A sequência abaixo descreve a sucessão de eventos que ocorre com páginas *JSP*:

- **Tradução** – o objetivo dessa fase é transformar uma página *JSP* em uma classe *servlet*;
- **Compilação** – ocorre quando a classe *servlet* é compilada em *bytecodes* Java de maneira análoga à compilação normal de outras classes;
- **Carregamento e instanciação da classe** – durante a execução de uma página *JSP*, o *container JSP* deve carregar os *bytecodes* e instanciar o *servlet* compilado;
- **Execução** – nessa fase, a página já pode atender às requisições `HTTP` e engloba os métodos `jspInit()`, `_jspService()` e `jspDestroy()`, que fazem parte das interfaces `JspPage` e `HttpJspPage`, as quais derivam da interface `Servlet`.

A implementação da página *JSP* é conduzida da mesma forma que um *servlet*. O método `init()` do *servlet* compilado deve invocar o método `jspInit()`, que é invocado quando a página é inicializada (e apenas uma vez). Dessa forma, o autor da página `JSP` pode definir o seu próprio método `jspInit()`.

Já o método `_jspService()` é definido baseado no conteúdo da página `JSP`. O método `service()` faz a chamada a esse método passando como parâmetros dois objetos, *request* e *response*, derivados das interfaces `ServletRequest` e `ServletResponse`. No método `_jspService()`, são definidos os objetos implícitos (Seção 8.6.2), e todos os *scriptlets* e expressões definidos na página `JSP` são incorporados ao corpo desse método.

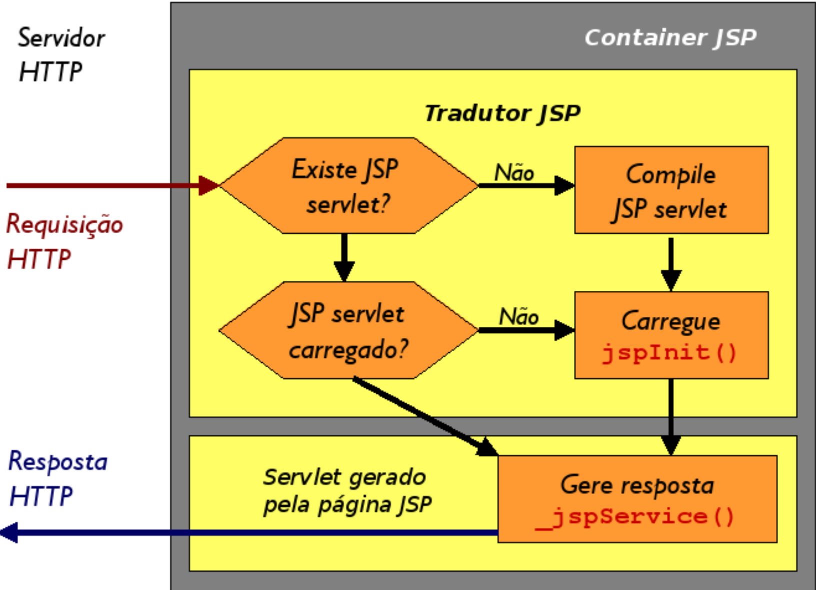


Figura 8.9 Ciclo de vida de páginas *JSP*.

O ciclo de vida da página JSP termina quando o *container* JSP resolve remover o JSP de funcionamento, por algum motivo (tempo de acesso, coleta de lixo, etc.), e o método `jspDestroy()` é invocado por meio do método `destroy()` do *servlet* compilado.

8.6.2 Objetos implícitos

São nove os objetos implícitos que podem ser acessados por *scriptlets* e expressões. Todos os objetos implícitos têm escopo da página JSP, com exceção dos próprios objetos relacionados a outros escopos (`application`, `session` e `request`).

- Os objetos implícitos `application`, `session`, `request` e `page` são relacionados aos escopos do contexto, sessão, requisição e da página JSP, respectivamente. São normalmente utilizados para ter acesso aos atributos (Seção 8.5.4) ligados a esses contextos.
- O objeto implícito `config` é uma instância da classe `ServletConfig` e pode acessar as informações relacionadas ao *servlet* que está atendendo a requisição.
- Os objetos `response` e `out` representam a resposta e a saída para o cliente. O objeto `response` é uma instância de alguma classe que implementa a interface `HttpServletResponse`. O objeto `out`, instância da classe `JspWriter`, é utilizado para escrever a saída diretamente.

- O objeto **exception** representa a exceção causada pela execução da requisição solicitada ao *servlet*. Esse objeto é uma instância da classe **Throwable** (Seção 3.10), da qual derivam todas as exceções e erros.

- O objeto **pageContext**, instância da classe **PageContext**, representa o contexto da página JSP. Todos os objetos implícitos podem ser obtidos pela invocação de um método **getXXX()** correspondente nesse objeto. Por exemplo, a referência ao objeto implícito **session** pode ser obtida através da invocação de **pageContext.getSession()**.

8.6.3 Exemplo: conversão de temperaturas

Esta seção apresenta uma diferente implementação da aplicação Web, discutida na Seção 8.5.6, que provê a funcionalidade de conversão de temperaturas. Diferentemente da aplicação Web discutida na Seção 8.5.6, a aplicação Web discutida aqui foi desenvolvida utilizando páginas JSP, e seus principais componentes são discutidos nas próximas subseções.

8.6.3.1 Arquivo **index.html**

Análogo ao arquivo **index.html** discutido na Seção 8.5.6.1, esse arquivo HTML (Código 8.9) é a página inicial da aplicação e possui um formulário HTML que deve ser preenchido e posteriormente submetido pelo usuário da aplicação. As informações a serem preenchidas são as mesmas: (1) tipo de conversão (Celsius-Fahrenheit ou Celsius-Kelvin); (2) temperatura inicial em graus Celsius; (3) temperatura final em graus Celsius; e (4) variação da temperatura.

Ou seja, o contéudo dos dois arquivos **index.html** são praticamente os mesmos. A única diferença é que o formulário HTML do arquivo apresentado nesta seção invoca uma página JSP (**conversão.jsp**) ao invés de um *servlet*. A página **conversão.jsp** será discutida na Seção 8.6.3.3.

```
<html>
    <head>
        <meta content="text/html; charset=utf-8" http-equiv="Content-Type">
        <title>Celsius-Fahrenheit</title>
    </head>
    <body>
        <form name="entry" method="post" action="conversão.jsp">
            Conversão:
            <input type ="radio" name="tipo" value="0" checked="checked">Fahrenheit
            <input type ="radio" name="tipo" value="1">Kelvin
            <br/> <br/>
            Inicial (<code>&deg;</code>C):
            <input type ="text" name="inicial">
            <br/> <br/>
            Final   (<code>&deg;</code>C):
            <input type ="text" name="final">
            <br/> <br/>
            Variação:
            <input type ="radio" name="var" value="5" checked="checked">5
            <input type ="radio" name="var" value="10">10
            <br/><br/>
            <input type="reset" name="reset" value="Inicializa Valores">
            <input type="submit" name="submit" value="Solicita a Conversão">
        </form>
    </body>
</html>
```

Código 8.9 Arquivo `index.html`.

8.6.3.2 Arquivo `web.xml`

Conforme discutido, esse arquivo encontra-se presente em todas as aplicações Web na plataforma Java. Tendo em vista que essa aplicação apenas usa páginas `JSP`, o arquivo **`web.xml`** é bastante simples. Apesar de páginas `JSP` serem convertidas em *servlets*, elas não precisam ser configuradas. Dessa forma, esse arquivo apenas configura o tempo de vida das sessões e o arquivo inicial da aplicação (**`index.html`**).

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml
    /ns/j2ee/web-app_2_4.xsd">
    <session-config>
        <session-timeout>
            10
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>
            index.html
        </welcome-file>
    </welcome-file-list>
</web-app>
```

Código 8.10 Arquivo `web.xml`.

8.6.3.3 Página JSP – conversão.jsp

A página **conversão.jsp** é a página JSP invocada após a submissão do formulário HTML presente no arquivo **index.html**, discutido anteriormente.

Conforme se pode observar, a implementação dessa página JSP é bastante similar à implementação da classe **ServletConversão**, discutida na Seção 8.5.6.3. É importante salientar os seguintes aspectos da implementação:

- Os objetos implícitos **request**, **response** e **session** são acessados;
- Os valores dos parâmetros são obtidos e armazenados como atributos de sessão;
- A requisição será redirecionada para a página **fahrenheit.jsp**, caso o valor do parâmetro **tipo** for igual a **0**, e para **kelvin.jsp**, caso contrário;
- Se algum desses valores é inválido, há um redirecionamento para uma página de erro.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<%-- JSP chamado pelo index.html => roteador para (C -> F) ou (C -> K) --%>

<html>
    <head>
    </head>
    <body>
        <%
            try {
                String iniValue = request.getParameter("inicial");
                session.setAttribute("inicial", Integer.parseInt(iniValue));
                String finValue = request.getParameter("final");
                session.setAttribute("final", Integer.parseInt(finValue));
                String varValue = request.getParameter("var");
                session.setAttribute("var", Integer.parseInt(varValue));

                int tipo = Integer.parseInt(request.getParameter("tipo"));

                switch (tipo) {
                    case 0: {
                        response.sendRedirect("fahrenheit.jsp");
                        break;
                    }
                    case 1: {
                        response.sendRedirect("kelvin.jsp");
                        break;
                    }
                }
            } catch (NumberFormatException exc) {
                response.sendRedirect("erro.html");
            }
        %>
    </body>
</html>
```

Código 8.11 Página JSP – conversão.jsp.

8.6.3.4 Página **JSP** – **fahrenheit.jsp**

A página **fahrenheit.jsp** é responsável por construir dinamicamente uma página **HTML** que contém uma tabela de conversão de temperaturas (°C para °F). Conforme se pode observar, a implementação dessa página **JSP** é bastante similar à implementação da classe **ServletCelsiusFahrenheit**, discutida na Seção 8.5.6.4.

Note que essa página JSP recupera os valores dos atributos de sessão **inicial**, **final** e **var** através de invocação do método **getAttribute()** no objeto implícito **session**. Um exemplo de página **HTML** gerada dinamicamente por essa página JSP é similar à apresentada na Figura 8.7.

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%-- JSP que constrói uma tabela  de conversão de temperaturas (C -> F) --%>

<html>
    <head><title>Celsius-Fahrenheit</title></head>
    <body>

        <%-- Recuperando as variáveis de sessão inicial, final e var --%>
        <%-- Declarando a variável local fahr --%>

        <%
            int iniValue = (Integer) session.getAttribute("inicial");
            int finValue = (Integer) session.getAttribute("final");
            int varValue = (Integer) session.getAttribute("var");;
            double fahr;
        %>

        <table border = "1">
            <tr>
                <th> Celsius (<code>&deg;</code>C) </th>
                <th> Fahrenheit (<code>&deg;</code>F) </th>
            </tr>

            <%-- Definindo o comando for --%>

            <%
                for (int celsius = iniValue; celsius <= finValue; celsius += varValue) {
                    fahr = 9 / 5.0 * celsius + 32;
            %>

            <tr>
                <th> <%= celsius %> </th>
                <th> <%= fahr %> </th>
            </tr>

            <%-- Fechando o comando for --%>

            <%
                }
            %>
        </table>
    </body>
</html>
```

Código 8.12 Página JSP – **fahrenheit.jsp**.

8.6.3.5 Página JSP – kelvin.jsp

A página **kelvin.jsp** é responsável por construir dinamicamente uma página **HTML** que contém uma tabela de conversão de temperaturas (°C para °K). Conforme se pode observar, a implementação dessa página **JSP** é bastante similar à implementação da classe **ServletCelsiusKelvin**, discutida na Seção 8.5.6.5.

Note que essa página JSP recupera os valores dos atributos de sessão **inicial**, **final** e **var** através de invocação do método **getAttribute()** no objeto implícito **session**. Um exemplo de página **HTML** gerada dinamicamente por essa página JSP é similar à apresentada na Figura 8.8.

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%-- JSP que constrói uma tabela  de conversão de temperaturas (C -> K) --%>

<html>
    <head><title>Celsius-Fahrenheit</title></head>
    <body>

        <%-- Recuperando as variáveis de sessão inicial, final e var --%>
        <%-- Declarando a variável local kelvin --%>

        <%
            int iniValue = (Integer) session.getAttribute("inicial");
            int finValue = (Integer) session.getAttribute("final");
            int varValue = (Integer) session.getAttribute("var");;
            double kelvin;
        %>

        <table border = "1">
            <tr>
                <th> Celsius (<code>&deg;</code>C) </th>
                <th> Kelvin (<code>&deg;</code>K) </th>
            </tr>

            <%-- Definindo o comando for --%>

            <%
                for (int celsius = iniValue; celsius <= finValue; celsius += varValue) {
                    kelvin = celsius + 273.15;
            %>

            <tr>
                <th> <%= celsius %> </th>
                <th> <%= kelvin %> </th>
            </tr>

            <%-- Fechando o comando for --%>

            <%
                }
            %>
        </table>
    </body>
</html>
```

Código 8.13 Página JSP – kelvin.jsp.

8.7 Considerações finais

Esta unidade apresentou uma breve introdução do desenvolvimento Web na plataforma Java ao apresentar os conceitos básicos e as tecnologias Java *Applet*, *Servlet* e *JSP*.

Uma discussão aprofundada do desenvolvimento Web na plataforma Java encontra-se fora do escopo deste livro. Porém, é importante salientar que as tecnologias mais recentes, tais como o *framework* `Spring` (WALLS, 2011), utilizam como base as tecnologias discutidas nesta unidade. Dessa forma, acredita-se que, como uma primeira introdução, o conteúdo aqui apresentado é suficiente.

8.8 Estudos complementares

Para estudos complementares sobre os tópicos abordados nesta unidade, o leitor interessado pode consultar as seguintes referências:

ARNOLD, K.; GOSLING, J.; HOLMES, D. *The Java Programming Language*. 4. ed. Boston: Addison-Wesley, 2005.

CAELUM. *Apostila do curso FJ-21* – Java para Desenvolvimento Web. 2014. Disponível em: http://www.caelum.com.br/apostila-java-web. Acesso em: 12 ago. 2014.

DEITEL, P.; DEITEL, H. *Java*: Como programar. 8. ed. São Paulo: Pearson Brasil, 2010.

NETBEANS. *Trilha de Aprendizado do Java EE e Java Web*. 2014. Disponível em: https://netbeans.org/kb/trails/java-ee_pt_BR.html. Acesso em: 12 ago. 2014.

ORACLE DOCS. *The Java™ Tutorials* – Lesson: Java Applets. 2014. Disponível em: http://docs.oracle.com/javase/tutorial/deployment/applet/index.html. Acesso em: 12 ago. 2014.

PEREIRA, R. *Guia de Java na Web*. 1. ed. Rio de Janeiro: Ciência Moderna Ltda, 2006.

Referências

BARNES, J. *Programming in Ada 2005*. Boston: Addison-Wesley, 2006.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *UML*: Guia do Usuário. 2. ed. Rio de Janeiro: Campus, 2000.

CAELUM. *Apostila do curso FJ-11* – Java e Orientação a Objetos. 2014a. Disponível em: http://www.caelum.com.br/apostila-java-orientacao-objetos. Acesso em: 12 ago. 2014.

CAELUM. *Apostila do curso FJ-21* – Java para Desenvolvimento Web. 2014b. Disponível em: http://www.caelum.com.br/apostila-java-web. Acesso em: 12 ago. 2014.

CODD, E. F. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, New York, v. 13, n. 6, p. 377-387, 1970.

COULORIS, G.; DOLLIMORE, J.; KINDBERG, T. *Sistemas Distribuídos*: Conceitos e Projeto. 4. ed. Porto Alegre: Bookman Companhia, 2007.

CRUPI, J.; MALKS, D.; ALUR, D. *Core J2EE Patterns* – As Melhores Práticas e Estratégias de Design. 2. ed. Rio de Janeiro: Campus, 2004.

DAHL, O.-J.; MYHRHAUG, B.; NYGAARD, K. Some Features of the SIMULA 67 Language. In: CONFERENCE ON APPLICATIONS OF SIMULATIONS, 2., 1968, New York. *Proceedings ...*. New York: Winter Simulation Conference, 1968. p. 29-31.

ELMASRI, R.; NAVATHE, S. B. *Sistemas de Banco de Dados*. 4. ed. São Paulo: Pearson Addison Wesley, 2005.

FELDMAN, S. A Conversation with Alan Kay. *Queue*, ACM, New York, v. 2, n. 9, p. 20-30, dez. 2004.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Padrões de Projetos*: Soluções Reutilizáveis de Software Orientado a Objetos. Porto Alegre: Bookman Companhia, 2000.

JONES, R.; LINS, R. D. *Garbage Collection*: Algorithms for Automatic Dynamic Memory Management. 1. ed. Oxford: Willey & Sons Ltda, 1999.

JOSUTTIS, N. M. *The C++ Standard Library* – A Tutorial and Reference. 2. ed. Boston: Addison-Wesley, 2012.

KAY, A. C. The Early History of Smalltalk. In: BERGIN, J. T. J.; GIBSON, J. R. G. (Ed.). *History of Programming Languages – II*. New York: ACM, 1996. p. 511-598.

LINDHOLM, T.; YELLIN, F. *The Java™ Virtual Machine Specification*. 2. ed. Boston: Addison-Wesley, 1999.

NORTHOVER, S.; WILSON, M. *SWT*: The Standard Widget Toolkit. 1. ed. Boston: Addison-Wesley, 2004.

PAGE-JONES, M. *Fundamentos do Desenho Orientado a Objeto com UML*. São Paulo: MAKRON *Books* Ltda., 2001.

PEREIRA, R. *Guia de Java na Web*. 1. ed. Rio de Janeiro: Ciência Moderna Ltda, 2006.

PRESSMAN, R. *Engenharia de Software*: Uma Abordagem Profissional. 7. ed. São Paulo: Bookman, 2011.

SCOTT, M. L. *Programming Language Pragmatics*. 3. ed. Burlington: Morgan Kaufmann, 2009.

SEBESTA, R. W. *Conceitos de Linguagens de Programação*. 5. ed. Porto Alegre: Bookman Companhia, 2003.

SPEEGLE, G. D. *JDBC*: Practical Guide for Java Programmers. 1. ed. Burlington: Morgan Kaufmann, 2001.

WALLS, C. *Spring in Action*. 3. ed. New York: Manning Publications Co., 2011.

WAZLAWICK, R. S. *Análise e Projeto de Sistemas de Informação Orientados a Objetos*. 2. ed. Rio de Janeiro: Elsevier, 2011.

SOBRE O AUTOR

Delano Medeiros Beder

Prof. Delano Medeiros Beder, doutor pela Universidade Estadual de Campinas – Unicamp (2001), é professor adjunto do Departamento de Computação da Universidade Federal de São Carlos – UFSCar. Tem experiência na área de Ciência da Computação, com ênfase em Engenharia de Software, atuando principalmente nos seguintes temas: técnicas de estruturação de software (padrões, arquiteturas, componentes de software, etc.), engenharia Web (metodologias ágeis, desenvolvimento orientado a testes, etc.), tolerância a falhas (tratamento de exceções, mecanismos de recuperação de erros e ações atômicas coordenadas) e desenvolvimento de jogos educacionais, tendo publicado suas pesquisas em relevantes conferências e periódicos da Ciência da Computação.