

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Especialização em Informática: Ênfase: Análise de Sistemas

UM ESTUDO COMPARATIVO DAS LINGUAGENS JAVA E C#

por

Rafael Moreno Romão

Monografia de Final de Curso

Prof. Roberto da Silva Bigonha
Orientador

Belo Horizonte
2009

RAFAEL MORENO ROMÃO

UM ESTUDO COMPARATIVO DAS LINGUAGENS JAVA E C#

Monografia apresentada ao Curso de Especialização em Informática do Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do certificado de especialista em informática.

Área de concentração: Análise de Sistemas

Orientador: Prof. Roberto da Silva Bigonha

Belo Horizonte
2009

RESUMO

Este trabalho tem como objetivo realizar uma comparação entre as linguagens de programação Java e C#, explicitando suas semelhanças e diferenças, de modo a permitir ao leitor identificar aquela que se mostra mais adequada a suas necessidades e preferências pessoais. Para tanto, será apresentado um estudo minucioso dos recursos de cada uma das linguagens, comparando-os com os recursos equivalentes da outra, assim como buscando alternativas quando necessário. Para cada recurso apresentado, o leitor será capaz de perceber suas vantagens e desvantagens diante de suas necessidades particulares, sendo que ao final do trabalho será capaz de entender as peculiaridades de tais linguagens e poderá escolher, de modo consciente e sem preconceitos, aquela que se mostra mais conveniente a cada tipo de aplicação.

Palavras-chave: Java. C#. Código Gerenciado.

ABSTRACT

This work has the objective to perform a comparison between the Java and C# programming languages, explaining their similarities and differences, in such a way the reader will be able to identify which of them fits best his personal needs and preferences. For that, it will be presented a rigorous study of the resources of each one of these languages, comparing them to the other's equivalent ones, as well as looking for alternatives when necessary. For each presented resource, the reader will be able to see its advantages and disadvantages in front of his particular needs, being in the end of the work able to understand the peculiarities of such languages and to choose, in an aware and without prejudices way, the one which shows to be the most appropriate for each kind of application.

Keywords: Java. C#. Managed Code.

LISTA DE TABELAS

Tabela 1. Tipos Primitivos da Linguagem Java.....	41
Tabela 2. Tipos Simples da Linguagem C#.....	43

LISTA DE EXEMPLOS

Exemplo 1. Importação de tipos em Java	26
Exemplo 2. Importação estática de tipos em Java	27
Exemplo 3. Qualificadores de espaços de nomes	29
Exemplo 4. Convenções para escrita de código.....	37
Exemplo 5. Convenções para escrita de código em Java.....	37
Exemplo 6. Convenções para escrita de código em C#	38
Exemplo 7. Arranjos.....	45
Exemplo 8. Strings	47
Exemplo 9. Strings	48
Exemplo 10. Strings	48
Exemplo 11. Strings	49
Exemplo 12. Strings	49
Exemplo 13. Strings	50
Exemplo 14. Strings	50
Exemplo 15. Strings	50
Exemplo 16. Enumerações	51
Exemplo 17. Enums na linguagem Java.....	52
Exemplo 18. Enums na linguagem Java.....	53
Exemplo 19. Enums na linguagem Java.....	53
Exemplo 20. Enums na linguagem C#	54
Exemplo 21. Enums na linguagem C#	54
Exemplo 22. Enums na linguagem C#	55
Exemplo 23. Enums na linguagem C#	56
Exemplo 24. Enums na linguagem C#	56
Exemplo 25. Enums na linguagem C#	57
Exemplo 26. Enums na linguagem C#	57
Exemplo 27. Primeiro modelo mental para definição de membros estáticos	59

Exemplo 28. Primeiro modelo mental para definição de membros estáticos	59
Exemplo 29. Primeiro modelo mental para definição de membros estáticos	59
Exemplo 30. Campos	60
Exemplo 31. Campos	61
Exemplo 32. Campos constantes na linguagem Java	61
Exemplo 33. Campos constantes na linguagem Java	61
Exemplo 34. Campos constantes e somente-leitura na linguagem C#	62
Exemplo 35. Campos constantes e somente-leitura na linguagem C#	62
Exemplo 36. Campos constantes e somente-leitura na linguagem C#	63
Exemplo 37. Métodos.....	63
Exemplo 38. Arranjos de parâmetros na linguagem Java.....	64
Exemplo 39. Arranjos de parâmetros na linguagem C#	64
Exemplo 40. Propriedades na linguagem Java	65
Exemplo 41. Propriedades na linguagem C#.....	66
Exemplo 42. Propriedades na linguagem C#.....	66
Exemplo 43. Propriedades indexadas	67
Exemplo 44. Propriedades indexadas	68
Exemplo 45. Propriedades indexadas	68
Exemplo 46. Tipos estáticos.....	70
Exemplo 47. Interfaces.....	70
Exemplo 48. Interfaces.....	71
Exemplo 49. Interfaces.....	72
Exemplo 50. Herança e Polimorfismo.....	73
Exemplo 51. Estendendo um tipo na linguagem Java	73
Exemplo 52. Estendendo um tipo na linguagem C#.....	74
Exemplo 53. Estendendo um tipo na linguagem C#.....	74
Exemplo 54. Métodos finais na linguagem Java	75
Exemplo 55. Métodos finais na linguagem Java	75
Exemplo 56. Métodos selados na linguagem C#.....	76
Exemplo 57. Métodos virtuais na linguagem Java	76

Exemplo 58. Métodos virtuais na linguagem C#.....	77
Exemplo 59. Métodos virtuais na linguagem C#.....	77
Exemplo 60. Métodos abstratos.....	77
Exemplo 61. Métodos abstratos.....	78
Exemplo 62. Métodos sobrescritos na linguagem Java	78
Exemplo 63. Métodos sobrescritos na linguagem Java	79
Exemplo 64. Métodos sobrescritos na linguagem Java	79
Exemplo 65. Métodos sobrescritos na linguagem C#.....	80
Exemplo 66. Métodos sobrescritos na linguagem C#.....	80
Exemplo 67. Métodos sobrescritos na linguagem C#.....	81
Exemplo 68. Métodos sobrescritos na linguagem C#.....	81
Exemplo 69. Métodos reintroduzidos.....	82
Exemplo 70. Métodos reintroduzidos.....	82
Exemplo 71. Invocando um construtor do tipo base	84
Exemplo 72. Invocando um outro construtor do tipo em questão	85
Exemplo 73. Invocando um construtor do tipo base na linguagem C#	86
Exemplo 74. Invocando um outro construtor do tipo em questão	86
Exemplo 75. Bloco estático na linguagem Java	87
Exemplo 76. Construtores estáticos na linguagem C#	87
Exemplo 77. Finalizadores na linguagem Java.....	88
Exemplo 78. Finalizadores na linguagem Java.....	89
Exemplo 79. Finalizadores na linguagem C#	89
Exemplo 80. Finalizadores na linguagem C#	90
Exemplo 81. Modificador public.....	91
Exemplo 82. Modificador padrão.....	92
Exemplo 83. Modificador protected.....	93
Exemplo 84. Modificador private.....	94
Exemplo 85. Modificador public.....	95
Exemplo 86. Modificador internal.....	95
Exemplo 87. Modificador internal.....	96

Exemplo 88. Modificador protected.....	96
Exemplo 89. Modificador protected internal.....	97
Exemplo 90. Modificador private.....	98
Exemplo 91. Modificador private.....	98
Exemplo 92. Assemblies amigáveis	99
Exemplo 93. Inicialização dos membros de um tipo	100
Exemplo 94. Inicialização dos membros de um tipo	101
Exemplo 95. Escopo de variáveis locais na linguagem Java	102
Exemplo 96. Escopo de variáveis locais na linguagem C#	103
Exemplo 97. Tipos aninhados	103
Exemplo 98. Classes aninhadas estáticas	104
Exemplo 99. Classes aninhadas não estáticas.....	105
Exemplo 100. Classes aninhadas não estáticas.....	105
Exemplo 101. Classes internas locais.....	106
Exemplo 102. Tipos aninhados na linguagem C#	107
Exemplo 103. Tipos aninhados na linguagem C#	107
Exemplo 104. Tipos aninhados na linguagem C#	108
Exemplo 105. Tipos aninhados na linguagem C#	108
Exemplo 106. Tipos anônimos na linguagem Java.....	110
Exemplo 107. Tipos anônimos na linguagem C#.....	111
Exemplo 108. Tipos anônimos na linguagem C#.....	112
Exemplo 109. Identificação de tipos na linguagem Java.....	113
Exemplo 110. Identificação de tipos na linguagem Java.....	113
Exemplo 111. Identificação de tipos na linguagem C#	114
Exemplo 112. Identificação de tipos na linguagem C#	114
Exemplo 113. Conversão de tipos na linguagem Java.....	115
Exemplo 114. Conversão de tipos na linguagem C#.....	115
Exemplo 115. Conversão de tipos na linguagem C#.....	116
Exemplo 116. Comandos de iteração	116
Exemplo 117. Comandos de iteração	117

Exemplo 118. Comandos de iteração	117
Exemplo 119. Comandos de iteração	118
Exemplo 120. Comandos de iteração	118
Exemplo 121. Comandos de iteração	119
Exemplo 122. Comandos de iteração	120
Exemplo 123. O comando switch.....	121
Exemplo 124. O comando switch.....	121
Exemplo 125. O comando switch.....	121
Exemplo 126. Os comandos checked e unchecked	122
Exemplo 127. Tratamento de eventos na linguagem Java.....	126
Exemplo 128. Tratamento de eventos na linguagem Java.....	126
Exemplo 129. Tratamento de eventos na linguagem C#	127
Exemplo 130. Tratamento de eventos na linguagem C#	127
Exemplo 131. Tratamento de exceções	128
Exemplo 132. Tratamento de exceções	129
Exemplo 133. Tratamento de exceções	130
Exemplo 134. Tipos genéricos	130
Exemplo 135. Tipos genéricos na linguagem Java.....	131
Exemplo 136. Tipos genéricos na linguagem Java.....	132
Exemplo 137. Tipos genéricos na linguagem Java.....	133
Exemplo 138. Tipos genéricos na linguagem Java.....	134
Exemplo 139. Tipos genéricos na linguagem C#	135
Exemplo 140. Tipos genéricos na linguagem C#	136
Exemplo 141. Tipos genéricos na linguagem C#	136
Exemplo 142. Iteradores na linguagem Java	142
Exemplo 143. O comando for estendido	143
Exemplo 144. Iteradores na linguagem C#.....	144
Exemplo 145. O comando foreach	145
Exemplo 146. O comando foreach	146
Exemplo 147. Blocos de iteração	147

Exemplo 148. Paralelismo na linguagem Java	149
Exemplo 149. Paralelismo na linguagem Java	150
Exemplo 150. Paralelismo na linguagem Java	151
Exemplo 151. Paralelismo na linguagem Java	152
Exemplo 152. Paralelismo na linguagem Java	152
Exemplo 153. Paralelismo na linguagem Java	152
Exemplo 154. Paralelismo na linguagem C#.....	153
Exemplo 155. Paralelismo na linguagem C#.....	153
Exemplo 156. Paralelismo na linguagem C#.....	154
Exemplo 157. Paralelismo na linguagem C#.....	154
Exemplo 158. Anotações na linguagem Java	155
Exemplo 159. Anotações na linguagem Java	156
Exemplo 160. Anotações na linguagem Java	156
Exemplo 161. Anotações na linguagem Java	156
Exemplo 162. Anotações na linguagem Java	156
Exemplo 163. Anotações na linguagem Java	157
Exemplo 164. Anotações na linguagem Java	157
Exemplo 165. Anotações na linguagem Java	157
Exemplo 166. Atributos na linguagem C#	158
Exemplo 167. Atributos na linguagem C#	159
Exemplo 168. Atributos na linguagem C#	159
Exemplo 169. Atributos na linguagem C#	160
Exemplo 170. Atributos na linguagem C#	160
Exemplo 171. Programação reflexiva na linguagem Java	161
Exemplo 172. Programação reflexiva na linguagem Java	161
Exemplo 173. Programação reflexiva na linguagem Java	161
Exemplo 174. Programação reflexiva na linguagem Java	162
Exemplo 175. Programação reflexiva na linguagem Java	162
Exemplo 176. Programação reflexiva na linguagem C#	162
Exemplo 177. Programação reflexiva na linguagem C#	163

Exemplo 178. Programação reflexiva na linguagem C#	163
Exemplo 179. Programação reflexiva na linguagem C#	163
Exemplo 180. Programação reflexiva na linguagem C#	163
Exemplo 181. Acesso a código não gerenciado na linguagem Java.....	165
Exemplo 182. Acesso a código não gerenciado na linguagem Java.....	165
Exemplo 183. Acesso a código não gerenciado na linguagem C#	166
Exemplo 184. Diretivas de compilação	167
Exemplo 185. Inicialização de arranjos, coleções e objetos.....	167
Exemplo 186. Passagem de parâmetros por referência	169
Exemplo 187. Passagem de parâmetros por referência	170
Exemplo 188. Passagem de parâmetros por referência	170
Exemplo 189. Passagem de parâmetros por referência	170
Exemplo 190. Passagem de parâmetros por referência	171
Exemplo 191. A cláusula unsafe	171
Exemplo 192. A cláusula using	172
Exemplo 193. Tipos anuláveis	173
Exemplo 194. Tipos anuláveis	173
Exemplo 195. Tipos anuláveis	174
Exemplo 196. Tipos anuláveis	174
Exemplo 197. Tipos anuláveis	174
Exemplo 198. Tipos anuláveis	174
Exemplo 199. Tipos anuláveis	175
Exemplo 200. Tipos parciais	176
Exemplo 201. Tipos parciais	177
Exemplo 202. Sobrecarga de operadores	178
Exemplo 203. Sobrecarga de operadores	178
Exemplo 204. Inferência de tipos	179
Exemplo 205. Inferência de tipos	180
Exemplo 206. Delegates, métodos anônimos e expressões lambda	181
Exemplo 207. Delegates, métodos anônimos e expressões lambda	182

Exemplo 208. Delegates, métodos anônimos e expressões lambda	182
Exemplo 209. Delegates, métodos anônimos e expressões lambda	182
Exemplo 210. Delegates, métodos anônimos e expressões lambda	183
Exemplo 211. Métodos de extensão	184
Exemplo 212. Métodos de extensão	184
Exemplo 213. Métodos de extensão	185
Exemplo 214. Linq	185
Exemplo 215. Linq	186
Exemplo 216. Linq	186
Exemplo 217. LinqToSQL	187

LISTA DE SIGLAS

API	Application Programming Interface
CIL	Common Intermediate Language
CLR	Common Language Runtime
DLL	Dynamic-link Library
ECMA	European Computer Manufacturers Association
GNU	GNU is Not Unix
IEC	International Electrotechnical Commission
ISO	International Organization of Standardization
JCF	Java Collections Framework
JDK	Java Development Kit
JNI	Java Native Interface
JRE	Java Runtime Environment
LINQ	Language Integrated Query
SGBD	Sistema de Gerenciamento de Bancos de Dados
SQL	Structured Query Language

SUMÁRIO

1	INTRODUÇÃO.....	18
2	REFERENCIAL TEÓRICO.....	19
3	METODOLOGIA	20
4	AMBIENTAÇÃO.....	21
4.1	Apresentação das Linguagens.....	21
4.2	Breve História das Linguagens.....	22
4.2.1	A Linguagem Java	22
4.2.2	A Linguagem C#	23
4.3	Estrutura das Linguagens	24
4.4	Importação de Tipos	25
4.4.1	Importação de Tipos em Java.....	25
4.4.2	Importação de Tipos em C#.....	27
4.5	Módulos Executáveis	29
4.6	Linguagem Intermediária	30
4.7	Ambientes de Execução	30
4.8	Requisitos de Sistemas.....	31
4.8.1	Requisitos de Sistema para Aplicações Java	32
4.8.2	Requisitos de Sistema para Aplicações C#	32
4.9	Padronização e Licença	33
4.9.1	Padronização e Licença para a Linguagem Java.....	33
4.9.2	Padronização e Licença para a Linguagem C#.....	33
4.10	Portabilidade de Código e Portabilidade de Binários	33
4.10.1	Portabilidade de Códigos e Binários em Java	34
4.10.2	Portabilidade de Códigos e Binários em C#	34
4.11	Ambientes de Desenvolvimento	35
4.11.1	Ambientes de Desenvolvimento Java	35
4.11.2	Ambientes de Desenvolvimento C#	35
4.12	Convenções para Escrita de Códigos	36

4.12.1	Convenções para Escrita de Código em Java.....	37
4.12.2	Convenções para Escrita de Código em C#.....	38
5	COMPARAÇÃO DOS RECURSOS COMUNS.....	39
5.1	Tipos de Dados Básicos.....	39
5.1.1	Tipos de Dados Básicos da Linguagem Java	40
5.1.2	Tipos de Dados Básicos da Linguagem C#	41
5.2	Arranjos	43
5.3	Strings	46
5.4	Enumerações.....	51
5.4.1	Enums na Linguagem Java:.....	51
5.4.2	Enums na Linguagem C#:.....	53
5.4.3	Operações com Enums:	57
5.5	Definição de Tipos.....	58
5.5.1	Membros de um Tipo.....	58
5.5.2	Tipos Estáticos.....	69
5.6	Interfaces.....	70
5.7	Herança e Polimorfismo	72
5.7.1	Estendendo um Tipo na Linguagem Java:	73
5.7.2	Estendendo um Tipo na Linguagem C#:	74
5.7.3	Herança e Sobrescrita de Métodos.....	74
5.8	Construtores e Finalizadores	82
5.8.1	Construtores	83
5.8.2	Finalizadores	88
5.9	Modificadores de Acesso	90
5.9.1	Modificadores de Acesso da Linguagem Java.....	91
5.9.2	Modificadores de Acesso da Linguagem C#.....	94
5.10	Inicialização dos Membros de um Tipo	99
5.11	Escopo de Variáveis Locais	101
5.11.1	Escopo de Variáveis Locais na Linguagem Java	102
5.11.2	Escopo de Variáveis Locais na Linguagem C#.....	102
5.12	Tipos Aninhados.....	103

5.12.1	Tipos Aninhados na Linguagem Java	104
5.12.2	Tipos Aninhados na Linguagem C#	106
5.13	Tipos Anônimos.....	109
5.13.1	Tipos Anônimos na Linguagem Java:.....	109
5.13.2	Tipos Anônimos na Linguagem C#:.....	110
5.14	Identificação e Conversão de Tipos.....	112
5.14.1	Identificação de Tipos na Linguagem Java	113
5.14.2	Identificação de Tipos na Linguagem C#	113
5.14.3	Conversão de Tipos na Linguagem Java	114
5.14.4	Conversão de Tipos na Linguagem C#	115
5.15	Comandos de Iteração	116
5.16	O Comando Switch	120
5.17	Operações Numéricas	122
5.17.1	Os Operadores Checked e Unchecked	122
5.17.2	O Modificador strictfp.....	123
5.17.3	O Tipo System.Decimal	124
5.18	Tratamento de Eventos.....	124
5.18.1	Tratamento de Eventos na Linguagem Java	125
5.18.2	Tratamento de Eventos na Linguagem C#	127
5.19	Tratamento de Exceções.....	128
5.20	Tipos Genéricos	130
5.20.1	Tipos Genéricos na Linguagem Java	131
5.20.2	Tipos Genéricos na Linguagem C#	134
5.21	Coleções.....	136
5.21.1	Coleções na Linguagem Java	137
5.21.2	Coleções na Linguagem C#	139
5.22	Iteradores	141
5.22.1	Iteradores na Linguagem Java	141
5.22.2	Iteradores na Linguagem C#	143
5.23	Paralelismo	148
5.23.1	Paralelismo na Linguagem Java.....	149

5.23.2	Paralelismo na Linguagem C#.....	153
5.24	Anotações ou Atributos	154
5.24.1	Anotações na Linguagem Java	155
5.24.2	Atributos na Linguagem C#.....	157
5.25	Programação Reflexiva	160
5.25.1	Programação Reflexiva na Linguagem Java.....	161
5.25.2	Programação Reflexiva na Linguagem C#.....	162
5.26	Acesso a Código Não Gerenciado	164
5.26.1	Acesso a Código Não Gerenciado na Linguagem Java.....	164
5.26.2	Acesso a Código Não Gerenciado na Linguagem C#.....	165
6	RECURSOS EXCLUSIVOS DA LINGUAGEM C#	167
6.1	Diretivas de Compilação.....	167
6.2	Inicialização de Arranjos, Coleções e Objetos	167
6.3	Structs.....	168
6.4	Passagem de Parâmetros por Referência	169
6.5	A Cláusula “unsafe”	171
6.6	A Cláusula “using”	171
6.7	Tipos Anuláveis	173
6.8	Tipos Parciais	175
6.9	Sobrecarga de Operadores.....	177
6.10	Geração de Tipos e Código em Tempo de Execução	179
6.10.1	Geração de Tipos em Tempo de Execução	179
6.10.2	Geração de Código em Tempo de Execução.....	179
6.11	Inferência de Tipos	179
6.12	Delegates, Métodos Anônimos e Expressões Lambda.....	180
6.13	Métodos de Extensão	183
6.14	LINQ.....	185
6.14.1	LinqToSQL.....	187
7	CONCLUSÃO	189
	REFERÊNCIAS	190

1 INTRODUÇÃO

Atualmente, os ambientes de execução de código gerenciado têm tido uma aceitação e uma adoção cada vez maior no mercado de *softwares*. Empresas como a Microsoft, através do .NET Framework, e a Sun Microsystems, através da plataforma Java, têm apostado todas as suas fichas em tal paradigma.

Com tanto investimento e com a difusão desse tipo de plataforma, as linguagens Java e C# têm ganhado grande destaque no mercado de *softwares* corporativos, sendo que hoje são poucos os nichos de mercado nos quais os desenvolvedores de *software* não necessitam escolher uma dessas linguagens como sua principal linguagem de trabalho.

Considerando-se tal realidade, este trabalho pretende estabelecer um paralelo entre tais linguagens, em suas versões mais recentes, explicitando suas semelhanças, diferenças e falsas semelhanças, de forma minuciosa, de modo a permitir ao leitor identificar aquela que seja mais adequada a suas necessidades e preferências pessoais, além de facilitar o processo de maturação dos usuários de uma delas ao iniciar os estudos e o trabalho com a outra.

2 REFERENCIAL TEÓRICO

As principais referências teóricas para a elaboração deste trabalho foram os documentos de especificação das linguagens Java (GOSLING, 2005) e C# (ECMA, 2006), os quais descrevem todos os aspectos considerados durante a construção da linguagem, e um artigo publicado por Dare Obasanjo (OBASANJO, 2007), o qual faz uma extensa comparação entre essas linguagens, porém com um enfoque diferente do que será usado neste trabalho.

Além disso, foram utilizados os documentos de ajuda das linguagens e ambientes de execução, disponibilizados na Internet, assim como diversos artigos também retirados da Internet, sendo que cada uma das afirmações obtidas de tais fontes foi analisada e testada antes de ser considerado confiável, conforme será descrito no capítulo “Metodologia”.

3 METODOLOGIA

A metodologia utilizada para a elaboração deste trabalho consiste de:

- Levantamento das peculiaridades dos ambientes de desenvolvimento e execução utilizados por cada uma das linguagens;
- Levantamento de todos os recursos disponibilizados por cada uma das linguagens, através de análise de suas especificações;
- Breve comparação dos recursos disponíveis em cada uma delas, a fim de determinar os recursos de uma delas que possuem algum equivalente na outra, assim como aqueles que são exclusivos;
- Seleção dos recursos mais relevantes para serem apresentados no trabalho;
- Estudo aprofundado, através da especificação das linguagens e dos recursos disponíveis na Internet, de cada um dos recursos, para cada uma das linguagens, a fim de elucidar quaisquer dúvidas referentes ao seu conceito e modo de uso;
- Comparação minuciosa de cada um dos recursos equivalentes das duas linguagens e busca por alternativas para os recursos exclusivos;
- Apresentação resumida de cada um dos recursos, de modo independente de linguagem;
- Apresentação de cada recurso, para cada uma das linguagens, comparando-o com o recurso equivalente da outra, sempre que possível;
- Utilização de exemplos de trechos de códigos para a ilustração e verificação de cada um dos aspectos mencionados nas análises e comparações de recursos;

4 AMBIENTAÇÃO

Ao se comparar duas linguagens de programação, há muito mais a se considerar do que apenas as próprias linguagens. O ambiente em que as mesmas serão utilizadas e o ambiente em que os programas criados por elas serão executados não podem deixar de ser considerados.

Este capítulo fará uma breve comparação entre esses ambientes, para o caso das linguagens Java e C#.

4.1 Apresentação das Linguagens

A linguagem de programação Java foi lançada em 1995 pela empresa Sun Microsystems e após se popularizar se tornou uma linguagem livre e de código aberto, ou seja, suas especificações, códigos fontes e direitos de uso não são mais pertencentes à empresa que a criou, mas sim de domínio público.

Assim como a linguagem C#, a linguagem Java se diferencia da maioria das linguagens de programação existentes até a década de 90 por ser executada em um ambiente gerenciado, ou seja, não é executada diretamente sobre o sistema operacional, e por ser compilada em duas fases, uma para código intermediário e outra, no momento de sua execução, para código nativo.

A linguagem C#, por sua vez, surgiu em 2001 inspirando-se principalmente na linguagem Java, que estava em ascendente popularidade naquela época e ao contrário de sua inspiradora, é uma linguagem proprietária, pertencente à Microsoft Corporation, empresa que a desenvolveu. Porém sua especificação, assim como as especificações das bibliotecas do .NET

Framework, ambiente de execução sobre a qual a mesma é executada, é aberta, de modo que aquele que quiser implementar um compilador para a mesma poderá fazê-lo livremente.

Atualmente a linguagem Java conta com diversos compiladores, ambientes de desenvolvimento e ambientes de execução, disponíveis para os principais sistemas operacionais existentes atualmente. Em contra-partida, o .NET Framework e a linguagem C#, só são oficialmente suportados e homologados pela Microsoft para executar sobre algumas versões do sistema operacional Windows, e o seu principal ambiente de desenvolvimento é o Visual Studio, também comercializado pela Microsoft.

As versões das linguagens consideradas neste trabalho serão as versões 1.6 da linguagem Java e 3.0 da linguagem C#.

4.2 Breve História das Linguagens

4.2.1 A Linguagem Java

Em 1991, dentro da Sun Microsystems, foi iniciado um projeto denominado Green Project, liderado por Patrick Naughton, Mike Sheridan e James Gosling, com o objetivo de desenvolver uma linguagem de programação orientada a objetos que antecipasse o que eles acreditavam ser a próxima onda do mundo digital, a convergência dos computadores para equipamentos e eletrodomésticos usados no cotidiano das pessoas.

Em 1992 foi apresentado um projeto denominado StartSeven, um controle remoto com uma interface gráfica *touchscreen*. Para esse novo dispositivo foi desenvolvida uma nova linguagem de programação então denominada Oak.

Em janeiro de 1995 a linguagem Oak foi adaptada para a Internet e rebatizada de Java. Projetada para se mover por meio das redes de dispositivos heterogêneos, redes como a

Internet, suas aplicações poderiam ser executadas dentro de navegadores e tudo seria disponibilizado instantaneamente, o que promoveu a rápida disseminação da tecnologia. Em pouco tempo o número de usuários cresceu bruscamente e grandes fornecedores de tecnologia, como a IBM, anunciaram suporte para a tecnologia Java.

Em 1997, a Sun Microsystems tentou submeter a linguagem à padronização pelos órgãos ISO/IEC e ECMA, mas acabou desistindo.

Em novembro de 2006, a maior parte dos fontes da plataforma Java já havia sido lançada como código aberto, através dos termos da GNU General Public License, processo que foi concluído em maio de 2007, restando apenas uma pequena porção de códigos da qual a Sun não possuía a propriedade.

A versão atual da linguagem Java foi lançada em dezembro de 2006 trazendo diversas modificações em sua estrutura, em grande parte influenciadas pelas novidades trazidas com o C# 3.0.

4.2.2 A Linguagem C#

Em 1999, objetivando combater a crescente expansão da plataforma Java frente às convencionais ferramentas de desenvolvimento Windows, a Microsoft Corporation inicia o desenvolvimento de um ambiente gerenciado de execução e de uma nova linguagem de programação, baseada na linguagem C (ISO, 2007) e tão ou mais prática e produtiva que a linguagem Java.

Um pouco mais tarde, no ano 2000, o projeto .NET é apresentado ao público, assim como sua principal, dentre várias possíveis, linguagens de programação a linguagem C#.

Em 2004 foi lançado o .NET versão 2.0 e junto com ele a segunda versão da linguagem C#, que viria dessa vez a trazer grandes inovações e avanços.

Em 2005 já eram conhecidos todos os novos recursos que estariam disponíveis na versão 3.0, ainda a ser lançada. A especificação da nova versão já havia sido inclusive padronizada pela ECMA.

Em novembro de 2007 é lançado o .NET 3.5 e com ele a versão 3.0 da linguagem C#, trazendo ainda mais inovações e ampliando assim o seu potencial produtivo.

4.3 Estrutura das Linguagens

Tanto a linguagem Java quanto a linguagem C# são baseadas na sintaxe da linguagem C++ (ISO, 1998).

São linguagens compactas e práticas, que visam a segurança e a produtividade, de modo a eliminar diversos dos problemas e preocupações que o desenvolvedor pode ter ao utilizar a linguagem C++. Muitos desses aspectos não se devem especificamente aos recursos da linguagem, mas sim, ao ambiente que gerencia a execução dos aplicativos escritos em tais linguagens: o Java Runtime Environment (JRE), no caso da linguagem Java, e o Common Language Runtime (CLR), no caso da linguagem C#.

No que diz respeito ao suporte ao paradigma da orientação a objetos, podemos dizer que ambas as linguagens são equivalentes, mesmo que atacando certos aspectos de formas distintas.

A linguagem Java, ao ser criada, já previa um amplo suporte às necessidades de uma linguagem preparada para o futuro e para explorar de modo completo tal paradigma, e ao observar a sintaxe da linguagem C#, é fácil notar sua influência, mesmo as linguagens

divergindo-se substancialmente em determinados pontos. A análise de tais pontos é o foco principal deste trabalho.

4.4 Importação de Tipos

Uma característica comum a ambas as linguagens é a presença de um robusto mecanismo de importação de tipos (classes, estruturas e tipos primitivos).

Através desse mecanismo, é possível informar ao compilador que tipos deseja-se utilizar, de modo a não deixar dúvidas quanto a quais são esses tipos, mesmo que existam, em diferentes pontos da aplicação, tipos distintos utilizando o mesmo nome.

Esse mecanismo se caracteriza, em ambas as linguagens, pela atribuição de um nome completo aos tipos, de modo a enquadrá-los em pacotes, que por sua vez também podem ser enquadrados em outros pacotes.

4.4.1 Importação de Tipos em Java

Na linguagem Java, o mecanismo de importação de tipos funciona através da inserção de cada tipo em um pacote, que por sua vez também pode estar inserido em um outro pacote. Desse modo, quando for necessário utilizar um determinado tipo X na definição de um outro tipo Y , basta inserir o endereço do arquivo que contém a definição do tipo X no caminho de busca do JRE, e informar ao compilador, no topo do arquivo que contém a definição do tipo Y , que em tal arquivo haverá referências para o tipo X , utilizando-se a cláusula `import`.

Exemplo:

Arquivo X.java:

```
package pacoteX;           //Definição do pacote em que se encontra o tipo X
public class X {           //Definição do tipo X
}

```

Arquivo Y.java:

```
package pacoteY;           //Definição do pacote em que se encontra o tipo Y
import pacoteX.X;          //Importação do tipo X pertencente
                           //ao pacote pacoteX

public class Y {
    public X x = new X();   //Referência para o tipo X dentro
                           //da definição do tipo Y
}

```

Exemplo 1. Importação de tipos em Java

Algumas considerações devem ser feitas com relação ao endereçamento de tipos em Java:

- A cláusula `import` pode ser usada tanto para notificar o uso de um tipo em particular como o uso de todos os tipos pertencentes a um determinado pacote, através do uso do caracter `*` no lugar do nome do tipo;
- Cada arquivo `.java` deve conter a definição de um único tipo de visibilidade pública;
- Todo arquivo `.java` deve ter o mesmo nome do tipo público contido nele;
- Caso um tipo `X` faça referência a um tipo `Y` e ambos pertençam a um mesmo pacote, a cláusula `import` é desnecessária;
- Caso um pacote `p2` esteja contido em um pacote `p1`, que por sua vez contém um tipo `T`, o endereço desse tipo será `p2.p1.T`;

- A estrutura de pastas onde os arquivos `.java` estiverem distribuídos deve ser coincidente com a estrutura de pacotes em que os tipos estejam distribuídos, ou seja, se o nome completo, ou endereço, de um tipo `T` for `p2.p1.T`, o arquivo `T.java` deve estar contido em uma pasta denominada `p1`, que por sua vez deve estar contida em uma pasta denominada `p2`.
- É possível também utilizar a cláusula `static` em conjunto com a cláusula `import` para importar os membros estáticos de um tipo diretamente.

Exemplo:

```
import static java.awt.Color.*;

public class Test{
    public static void main(String[] args) throws Exception{
        //Membros estáticos do tipo Color acessados diretamente
        System.out.println(RED + " com " + YELLOW + " fica " + ORANGE);
    }
}
```

Exemplo 2. Importação estática de tipos em Java

4.4.2 Importação de Tipos em C#

A importação de tipos na linguagem C# se dá de maneira muito semelhante ao modo como se dá na linguagem Java, porém o termo “pacote” (*package* em inglês) foi substituído pelo termo “espaço de nome” (*namespace* em inglês).

As principais diferenças em relação à linguagem Java são:

- A cláusula usada para informar ao compilador sobre a existência de uma referência para um tipo não pertencente ao mesmo pacote que o tipo em edição é denominada `using`, e sempre que usada faz referência a todo um *namespace*, e não a um tipo em particular, como é possível fazer com a cláusula `import` da linguagem Java;

- Não há nada semelhante ao comando `static import` na linguagem C#;
- Um arquivo `.cs` pode conter a definição de diversos tipos públicos, e não apenas um, como ocorre com os arquivos `.java`. Como consequência disso, os arquivos `.cs` não precisam ter o mesmo nome de nenhum dos tipos contidos nele, mesmo que essa prática possa ser aplicada como convenção;
- A estrutura de pastas onde os arquivos `.cs` estiverem distribuídos não precisa equivaler à estrutura de *namespaces* na qual os arquivos estiverem contidos;
- Diferentemente do que ocorre na plataforma Java, os módulos de execução no .NET Framework não são classes, mas sim, conjuntos de tipos denominados *assemblies*. Um *assembly* pode conter tipos definidos em diferentes *namespaces*, assim como um *namespace* pode conter tipos distribuídos por diferentes *assemblies*. Para ter acesso a um determinado tipo definido em outro *assembly*, além da referência através da cláusula `using`, também é necessário inserir uma referência para o *assembly* que contém o tipo a ser usado. Essa tarefa pode ser simplificada com a ajuda do ambientes de desenvolvimento.

4.4.2.1 *Qualificadores de espaços de nomes*

Na linguagem C#, é possível definir qualificadores para os espaços de nomes. Tais qualificadores possibilitam a resolução de conflitos quando mais de um tipo ou espaço de nome semelhantes estão disponíveis.

O exemplo abaixo demonstra o uso do operador `::` que informa ao compilador que o *namespace* à direita deve ser localizado dentro do escopo do qualificador passado à esquerda. Esse qualificador pode ser o comando `global` ou um *alias* para um determinado *namespace*.

Exemplo:

```
using System;
using sys = System;

namespace TestLib {
    class Test {
        public class System {}

        static DateTime Console = DateTime.Now;

        public static void Main(){
            Console.WriteLine("Console é DateTime"); //ERRO
            System.Console.WriteLine("System é tipo interno"); //ERRO

            global::System.Console.WriteLine("OK"); //OK
            sys::Console.WriteLine("OK"); //OK
        }
    }
}
```

Exemplo 3. Qualificadores de espaços de nomes

4.5 Módulos Executáveis

Um módulo executável, ou simplesmente módulo, pode ser definido como a menor unidade de código executável que compõe uma aplicação. Em Java, tal unidade é uma simples classe. Qualquer classe compilada pode ser utilizada por uma aplicação, para isso basta ter acesso ao arquivo `.class` que a contém. No caso da linguagem C#, e todas as demais aceitas pelo CLR, um módulo é composto por um conjunto de classes, denominado *assembly*. Pode-se dizer que assim como um *namespace* C# é análogo a um *package* Java, um *assembly* é análogo a um arquivo `.jar`, com a diferença no fato de que os arquivos `.jar` são opcionais, ao passo que os *assemblies* são obrigatórios para se ter acesso aos tipos definidos. Um *assembly* pode ser representado por um arquivo `.dll` ou um arquivo `.exe`, sendo que apenas o último pode ser executado isoladamente, ao passo que na linguagem Java, qualquer arquivo `.class` pode ser configurado de modo a se tornar um arquivo executável.

4.6 Linguagem Intermediária

Outra característica comum às linguagens Java e C# é o fato de que ambas são compiladas em duas etapas, a primeira pelo ambiente de desenvolvimento e a segunda pelo ambiente de execução. Nessa primeira compilação, são produzidos arquivos escritos em uma linguagem intermediária, a qual é compilada para códigos específicos do sistema operacional no momento da segunda compilação. Essa estratégia permite que diferentes sistemas operacionais sejam capazes de executar os mesmos programas.

No caso específico da linguagem Java, essa linguagem intermediária é conhecida como “Java Bytecode”, já no caso da linguagem C#, ela é conhecida como Common Intermediate Language (CIL). A principal diferença entre elas é o fato de a CIL ter sido projetada para ser independente da linguagem C#, o que permite que outras linguagens sejam utilizadas, dando ao desenvolvedor a possibilidade de escolher, dentre as várias disponíveis, aquela que seja mais adequada a suas preferências ou necessidades.

4.7 Ambientes de Execução

Como foi dito anteriormente, o uso de uma linguagem intermediária permite que o mesmo programa seja executado em diferentes sistemas operacionais. Isso acontece devido ao fato de diferentes versões do ambiente de execução, JRE ou CLR, serem implementadas para cada sistema operacional suportado.

No caso da linguagem Java, por ser uma linguagem não proprietária, há ambientes de execução disponíveis para as mais diversas plataformas e sistemas operacionais: PCs, macintoshes e dispositivos móveis rodando Windows, Linux, Mac OS, Symbian, Android.

No caso da linguagem C#, apesar de a mesma, assim como o CLR, possuir especificações não proprietárias, a sua produtora, a Microsoft Corporation, só fornece implementações dos ambientes de desenvolvimento e execução para as plataformas Windows. Algumas iniciativas, como o projeto Mono¹, suportado pela empresa Novell, estão desenvolvendo tais ambientes de forma aberta e com considerável sucesso, porém a qualidade desses trabalhos ainda deixa muito a desejar, se comparada à qualidade dos ambientes oficialmente suportados pela Microsoft.

Até o momento não há sinais de que tais ambientes serão oficialmente suportados para outras plataformas, porém uma versão resumida do .NET Framework já é suportada para diferentes sistemas operacionais a partir da versão 2.0 do Silverlight², lançada no dia 14 de outubro de 2008. Esse ambiente permite o uso da linguagem C# para o desenvolvimento de aplicações Web de interface rica, semelhante ao, amplamente difundido, ambiente Flash, da empresa Adobe.

4.8 Requisitos de Sistemas

Ambas as linguagens, C# e Java, se destinam à criação de aplicações para ambientes gerenciados, como já apresentado anteriormente. Durante a evolução da linguagem Java, e nos primeiros anos do .NET Framework, muitas críticas foram feitas com relação ao baixo desempenho apresentado por esse tipo de aplicação. Atualmente, não só o desempenho do código gerado pelos compiladores dessas linguagens, como o próprio desempenho dos

¹ Para mais informações acesse: <http://www.mono-project.com>.

² Para mais informações acesse: <http://silverlight.net>.

ambientes de execução melhorou consideravelmente. Além disso, a disponibilidade de computadores rápidos e a custo acessível tem crescido diariamente.

Atualmente os requisitos de sistema para a execução de aplicações desenvolvidas nessas duas linguagens são:

4.8.1 Requisitos de Sistema para Aplicações Java

O ambiente de execução da plataforma Java, JRE, é atualmente suportado para diversos sistemas operacionais para processadores de 32 e 64 bits, sendo que o requisito mínimo de capacidade de processamento não é nem mesmo especificado, uma vez que todos os processadores atuais, no caso dos *desktops* e servidores, já possuem tal capacidade. Já quanto à disponibilidade de memória RAM, para a maioria dos sistemas operacionais, 64MB já são suficientes, sendo que apenas para as versões de servidor do sistema operacional Windows esse valor passa para 128MB.

4.8.2 Requisitos de Sistema para Aplicações C#

Já no caso do .NET Framework, versão 3.5, principal ambiente de execução ao qual se destinam as aplicações desenvolvidas com a linguagem C#, versão 3.0, os requisitos de sistema são bastante mais restritivos.

Há suporte para plataformas de 32 e 64 bits, porém apenas para os sistemas operacionais Windows XP SP2 ou superior, Windows Vista, Windows Server 2003 e Windows Server 2008. Com relação à capacidade mínima de memória RAM exigida, o valor indicado é de 96MB, enquanto a capacidade mínima de processamento indicada é de 500Mhz.

4.9 Padronização e Licença

Como já dito na introdução deste trabalho, ambas as linguagens, Java e C#, possuem especificações padronizadas e de licença livre, sendo que apenas ambientes de execução, bibliotecas e ambientes de desenvolvimento estão sujeitos a direitos de propriedade.

4.9.1 Padronização e Licença para a Linguagem Java

A Sun Microsystems, empresa que coordena o desenvolvimento da linguagem Java, já tentou por duas vezes padronizar a linguagem Java junto aos órgãos ISO e ECMA, mas não obteve sucesso, sendo então padronizada pelo Java Community Process, um processo de formalização criado especificamente para esse fim.

4.9.2 Padronização e Licença para a Linguagem C#

No caso da linguagem C#, a padronização foi realizada junto ao ECMA, através da especificação ECMA-334, juntamente à especificação ECMA-335, que regulamenta a CLI, linguagem intermediária executada pelo .NET Framework. Essa padronização permitiu avanços de iniciativas de código aberto, como o já citado projeto Mono, que vem conseguindo grande sucesso e um alto grau de compatibilidade de códigos e binários entre as suas aplicações e as aplicações desenvolvidas para o .NET Framework.

4.10 Portabilidade de Código e Portabilidade de Binários

Uma das grandes vantagens da linguagem Java sobre a linguagem C#, citada em quase toda comparação, é a possibilidade de se transportar programas desenvolvidos em um determinado sistemas operacional para sistemas operacionais diferentes. Esse transporte de programas de um ambiente para outro é conhecido como portabilidade de *software*.

Mas o que viria a ser de fato essa portabilidade? Uma portabilidade de códigos? A qual permitiria que o mesmo código fonte fosse compilado de modo a produzir diferentes executáveis, cada um destinado a uma implementação específica do ambiente de execução? Ou uma portabilidade de binários? A qual permitiria que o mesmo *software* fosse executado em diferentes implementações do ambiente de execução?

O fato é que, devido às particularidades de cada ambiente de execução, dificilmente um mesmo módulo executável, ou até mesmo um mesmo código fonte, possa ser completamente transportado, ou seja, transportado sem que sejam necessárias ao menos pequenas alterações. Apesar disso, ao menos teoricamente, tal portabilidade é possível em ambas as plataformas: Java e .NET Framework, tanto para códigos fontes quanto para binários.

4.10.1 Portabilidade de Códigos e Binários em Java

Essa portabilidade é mais praticada no caso da plataforma Java, tanto devido à grande variedade de ambientes de execução disponíveis, quanto ao esforço feito pelos desenvolvedores para manter seus códigos, assim como seus binários, compatíveis com quaisquer variações do ambiente de execução. Ainda assim, na maioria das grandes aplicações, são necessárias compilações específicas para cada sistema operacional.

4.10.2 Portabilidade de Códigos e Binários em C#

No caso da linguagem C#, e das demais linguagens do .NET Framework, essa portabilidade é garantida em todos os sistemas operacionais oficialmente suportados, o que não é uma grande vantagem, uma vez que todos são variações do sistema operacional Windows. Ainda assim, pode-se notar um alto nível de compatibilidade de códigos e principalmente de binários ao se considerar também os códigos e binários produzidos usando a implementação do *framework* desenvolvida pelo projeto Mono, sendo possível compilar e executar códigos originalmente

escritos para as versões *linux* do ambiente de execução Mono utilizando as ferramentas de desenvolvimento e o ambiente de execução oficial da Microsoft, o .NET Framework, assim como também é possível o inverso, desde que, em ambos os casos, sejam utilizadas bibliotecas desenvolvidas de modo a serem independentes do sistema operacional, o que não se aplica às bibliotecas oficiais do .NET Framework.

4.11 Ambientes de Desenvolvimento

Uma grande diferença pode ser notada ao se comparar a disponibilidade de ambientes de desenvolvimento para as duas linguagens.

4.11.1 Ambientes de Desenvolvimento Java

Há atualmente diversos ambientes de desenvolvimento para a linguagem Java, dentre eles ambientes gratuitos, como é o caso dos dois mais conhecidos, o Eclipse e o NetBeans, e outros proprietários, como é o caso do JBuilder ou do XCode.

Cada um desses ambientes possui suas vantagens e suas desvantagens, sendo que cabe a cada equipe de desenvolvimento escolher aquele que seja mais adequado às suas necessidades. A maioria deles está disponível em todos os sistemas operacionais suportados, em especial os gratuitos.

4.11.2 Ambientes de Desenvolvimento C#

Como era de se esperar, todos os ambientes de execução disponibilizados pela Microsoft são aplicações Windows. Todas elas variações do Visual Studio, plataforma de desenvolvimento de *software* oficial da empresa, a qual conta com: desde versões gratuitas, caso das versões Express, como com versões empresariais, destinadas a grandes equipes que trabalham em

conjunto no mesmo projeto, disponibilizando a elas ferramentas completas de controle de requisitos, diagramação, desenvolvimento, testes, controle de versão e distribuição.

Há também ambientes de desenvolvimento desenvolvidos por terceiros, que podem ser usados tanto para aplicações .NET como para aplicações Mono, como é o caso do ambiente Delphi, consagrado por sua implementação Win32 que utiliza a linguagem pascal, e da implementação desenvolvida pelo projeto Mono, baseada na plataforma eclipse, e conhecido como MonoDevelop, o qual se encontra, atualmente, mais limitado até mesmo que as versões gratuitas do Visual Studio.

4.12 Convenções para Escrita de Códigos

É comum em todos os ambientes de produção de *software* o uso de convenções para padronizar o modo como os programadores escrevem os códigos fonte dos programas, o que permite que o mesmo se mantenha uniforme e fácil de ser compreendido por todos os membros da equipe de desenvolvimento.

Ambas as linguagens, Java e C#, identificam as declarações fazendo distinção entre letras maiúsculas e letras minúsculas. Essa característica reforça a necessidade de se adotar convenções de escrita de código. Tais convenções são adotadas por ambas as comunidades de desenvolvedores, no que se refere à nomenclatura de pacotes, tipos, métodos e campos.

Para tais convenções, três padrões de escrita são utilizados: Camel Case, Pascal Case e Upper Case. Os dois primeiros fornecem nomes constituídos pela junção de palavras de modo que a partir da segunda, as letras iniciais são escritas em maiúsculo e as demais em minúsculo, sendo que no caso do padrão Camel Case, a primeira letra da primeira palavra deve ser escrita em minúsculo, ao contrário do Pascal Case, que estabelece que a primeira letra da primeira

palavra deve ser escrita em maiúsculo. O padrão Upper Case estabelece que todas as letras devem ser escritas em maiúsculo.

Exemplo:

```
package pacoteDeTeste; //Nome do pacote escrito em Camel Case

public class ClasseDeTeste { //Nome do tipo escrito em Pascal Case
    public void salvaDados() { //Nome do método escrito em Camel Case
        //...
    }
    public static final int CONSTANTE = 0; //Nome de constante em Upper Case
}
```

Exemplo 4. Convenções para escrita de código

4.12.1 Convenções para Escrita de Código em Java

Para a comunidade Java, a adoção das convenções de escrita de código é algo quase que obrigatório, sendo que alguns compiladores chegam ao ponto de recusar a compilação de códigos que violem tais convenções.

Para nomenclatura de classes, ficou estabelecido o uso de Pascal Case, ao passo que para pacotes, métodos, campos, variáveis locais e parâmetros de métodos, ficou estabelecido o uso do padrão Camel Case.

O padrão Upper Case deve ser usado para nomenclatura de constantes.

Exemplo:

```
package pacoteDeTeste;

public class ClasseDeTeste {
    private int campo1 = 0;
    public void salvaDados(int param1, String param2) {
        int variavel1 = 0;
        //...
    }
    public static final int CONSTANTE = 0;
}
```

Exemplo 5. Convenções para escrita de código em Java

4.12.2 Convenções para Escrita de Código em C#

No caso da linguagem C#, não há restrições impostas pelo ambiente de desenvolvimento, porém há um consenso da comunidade de desenvolvedores em utilizar os mesmos padrões observados na implementação das bibliotecas do .NET Framework, ou seja, Camel Case para nomenclatura de campos privados e protegidos, variáveis locais e parâmetros de métodos e Pascal Case para nomenclatura de *namespaces*, tipos, métodos, propriedades e campos constantes ou públicos.

Exemplo:

```
namespace NamespaceDeTeste {  
    public class ClasseDeTeste {  
        private int campo1 = 0;  
        public void SalvaDados(int param1, string param2) {  
            int variavel1 = 0;  
            //...  
        }  
        public const int Constante = 0;  
    }  
}
```

Exemplo 6. Convenções para escrita de código em C#

5 COMPARAÇÃO DOS RECURSOS COMUNS

Até o lançamento da versão 2.0 da linguagem C#, eram poucos os recursos disponíveis em tal linguagem que não estavam também disponíveis na linguagem Java, assim como eram poucos os recursos disponíveis na linguagem Java que não podiam ser encontrados na linguagem C#. Porém, com o lançamento das versões 2.0, e posteriormente 3.5, do .NET Framework, assim como das versões 2.0, e posteriormente 3.0, da linguagem C#, o conjunto de recursos disponibilizados por tal linguagem que não se encontravam disponíveis na linguagem Java cresceu consideravelmente.

Com o lançamento da versão 1.6 da linguagem Java, o conjunto de recursos disponibilizados por ela se tornou mais próximo do conjunto de recursos disponibilizados pela versão 2.0 da linguagem C#. Para as novas versões da linguagem Java, a partir do lançamento do Java Development Kit (JDK) versão 7, espera-se uma maior equiparação entre os recursos das duas linguagens.

Este capítulo apresentará uma comparação entre os recursos comuns às linguagens Java, versão 1.6, e C#, versão 3.0, sendo que no capítulo seguinte serão apresentados os recursos exclusivos à linguagem C#.

5.1 Tipos de Dados Básicos

Em toda linguagem de programação, está disponível um conjunto básico de tipos de dados, os quais podem ser usados para a composição de novos tipos ou diretamente na implementação dos programas. Na maioria dessas linguagens, tais tipos são divididos em tipos primitivos, que representam dados simples, e objetos, que representam dados compostos. No caso específico da linguagem C#, e das demais linguagens de programação do .NET Framework, a

divisão dos tipos em tipos primitivos ou objetos, não existe, sendo que todo tipo, mesmo aqueles que representam dados simples, podem ser considerados objetos, sendo a sua categorização feita através do tipo de alocação em memória: por valor, alocados na memória *stack* (ECMA, 2006, p. 439), ou por referência, alocados na memória *heap* (ECMA, 2006, p. 440).

5.1.1 Tipos de Dados Básicos da Linguagem Java

Assim como ocorre com a maioria das linguagens de programação, e ao contrário do que ocorre com a plataforma .NET, os tipos básicos da linguagem Java são divididos em tipos primitivos e objetos, e assim como os tipos simples da plataforma .NET, os tipos primitivos da linguagem Java são alocados diretamente na memória *stack* do computador, ao contrário dos objetos, que são alocados na memória *heap*.

A linguagem Java disponibiliza um mecanismo conhecido como *boxing* e *unboxing* para tornar mais prático o uso dos tipos primitivos. Esse mecanismo consiste na conversão das instâncias dos tipos primitivos em instâncias de objetos, instâncias de classes conhecidas como *wrapper classes* (GOSLING, 2005, p. 86), de modo a permitir que sejam utilizadas como tal. As *wrapper classes* também são úteis na realização de algumas operações específicas a cada tipo primitivo, como conversões de, e para, outros tipos de dados e acesso a valores constantes de uso recorrente.

Os tipos primitivos da linguagem Java são:

Nome	Valores Possíveis	Tamanho	Tipo de Dado	Wrapper Class
byte	-128 a 127	1 byte	Números Inteiros	java.lang.Byte
short	-32768 a 32767	2 bytes	Números Inteiros	java.lang.Short
int	-2147483648 a 2147483647	4 bytes	Números Inteiros	java.lang.Integer
long	-9223372036854775808 a 9223372036854775807	8 bytes	Números Inteiros	java.lang.Long
char	0 a 65535	2 bytes	Números Inteiros	java.lang.Character
float	1.40×10^{45} a 3.4028235×10^{38}	4 bytes	Números Decimais	java.lang.Float
double	4.9×10^{-324} a $1.7976931348623157 \times 10^{308}$	8 bytes	Números Decimais	java.lang.Double
boolean	true ou false	1 byte	Verdadeiro ou Falso	Java.lang.Boolean

Tabela 1. Tipos Primitivos da Linguagem Java

5.1.2 Tipos de Dados Básicos da Linguagem C#

Se classificarmos os tipos básicos da linguagem C# conforme o tipo de alocação em memória, teremos os tipos simples, alocados por valor, e as classes, alocadas por referência.

Tipos simples também são conhecidos como *structs*, ou *Value Types*, ao passo que as classes também são conhecidas como *Reference Types*.

Os tipos simples da linguagem C#, assim como os tipos primitivos da linguagem Java, são usados para armazenar dados que possam ser representados por um bloco seqüencial de *bytes*, armazenados na memória *stack*.

O ambiente de execução do .NET Framework foi desenvolvido de modo a tratar todos os tipos definidos como *struct*, ou seja, aqueles que estendem o tipo `System.ValueType`, como sendo tipos de alocação por valor, assim como a fazer automaticamente as operações de *boxing* e *unboxing* quando tais tipos forem utilizados onde se espera utilizar tipos de alocação por referência.

Uma vez que o *boxing* e o *unboxing* são realizados automaticamente, e a maioria dos recursos disponíveis a uma classe também estão disponíveis nos *structs*, o uso de *wrapper classes* se torna desnecessário na linguagem C#.

A linguagem C# ainda fornece, aos tipos simples, apelidos (*aliases* em inglês) para facilitar a utilização dos mesmos, de modo que tipos como o `System.Int32` ou o `System.Double` possam ser referenciados apenas como `int` ou `double`, respectivamente.

Os tipos simples da linguagem C# são:

Nome	Valores Possíveis	Tamanho	Tipo de Dado	Tipo Real
sbyte	-128 a 128	1 byte	Números Inteiros	System.SByte
byte	0 a 255	1 byte	Números Inteiros	System.Byte
char	0 a 65535	2 bytes	Números Inteiros	System.Char
short	-32768 a 32767	2 bytes	Números Inteiros	System.Int16
ushort	0 a 65535	2 bytes	Números Inteiros	System.UInt16
int	-2147483648 a 2147483647	4 bytes	Números Inteiros	System.Int32
uint	0 a 4294967296	4 bytes	Números Inteiros	System.UInt32
long	- 9223372036854775808 a 9223372036854775807	8 bytes	Números Inteiros	System.Int64
ulong	0 a 18446744073709551616	8 bytes	Números Inteiros	System.UInt64
float	1.5×10^{-45} a 3.4×10^{38}	4 bytes	Números Decimais	System.Single
double	5.0×10^{-324} a 1.7×10^{308}	8 bytes	Números Decimais	System.Double
decimal	1.0×10^{-28} a 1.0×10^{28}	16 bytes	Números Decimais	System.Decimal
bool	true ou false	1 byte	Verdadeiro ou Falso	System.Boolean

Tabela 2. Tipos Simples da Linguagem C#

A seção “Structs”, no capítulo “Recursos Exclusivos da Linguagem C#”, apresentará mais detalhes a respeito dos tipos de dados de alocação por valor da linguagem C#.

5.2 Arranjos

Arranjos são estruturas de dados simples utilizadas para armazenar um conjunto seqüencial de dados de tamanho fixo. Em ambas as linguagens analisadas neste estudo, os arranjos são suportados e utilizados de maneira semelhante, porém com algumas pequenas diferenças.

Em ambas as linguagens, arranjos são *Reference Types*, ou seja, objetos, e podem conter, como elementos, instâncias de tipos primitivos ou objetos, no caso da linguagem Java, e *structs* ou objetos, no caso da linguagem C#.

Ao contrário do que permite a linguagem Java, na linguagem C#, os colchetes usados na declaração dos arranjos devem suceder, obrigatoriamente, o nome do tipo dos elementos do arranjo que se deseja criar. Colchetes sucedendo o nome da variável não são aceitos pela linguagem.

Em Java, cada arranjo criado é associado a uma classe, criada e utilizada implicitamente pelo compilador e definida conforme o tipo dos elementos do arranjo. Essa classe é um tipo anônimo e estende diretamente a classe `Object`.

De modo semelhante, há uma classe anônima associada a cada arranjo também na linguagem C#, porém essa classe não estende diretamente a classe `System.Object`, mas sim a classe abstrata `System.Array`, a qual provê métodos estáticos úteis ao trabalho com arranjos e pode até mesmo ser usada na declaração de variáveis às quais se pretende passar instâncias de arranjos.

Assim como ocorre com o Java, o C# permite a criação de arranjos de arranjos, também chamados de arranjos aninhados (*nested arrays*, *ragged arrays* ou *jagged arrays*). Além disso, um outro tipo de arranjo é suportado pela linguagem C#: os arranjos multidimensionais, que ao contrário dos arranjos de arranjos, permitem o acesso direto a um elemento através dos índices referentes a cada uma das dimensões que o compõem.

Exemplos (C#):

```
//Define um arranjo unidimensional contendo 3 arranjos unidimensionais de inteiros.
int[][] a = new int[3][];
//Define um arranjo bidimensional de 3 x 2 inteiros.
int[,] b = new int[3,2];
//Define um arranjo tridimensional de 3 x 2 x 5 inteiros.
int[,,] c = new int[3,2,5];
//Define um arranjo unidimensional contendo 3 arranjos tridimensionais de inteiros.
int[][][] d = new int[3][,,];
//Acesso a um elemento de um arranjo bidimensional.
int e = x[3,4];
```

Exemplo 7. Arranjos

Uma outra diferença entre as duas linguagens, referente aos arranjos, que merece ser destacada são as interfaces implementadas por cada classe de arranjo.

Em Java, as interfaces implementadas pelos arranjos são: `Cloneable`, que permite que um arranjo possa ser clonado e `java.io.Serializable`, que permite que um arranjo seja serializado (SUN, 2001).

Em C#, as interfaces implementadas pelos arranjos são: `ICloneable`, `IList`, `ICollection` e `IEnumerable`, além da interface genérica `IList<T>` que permite a conversão de um `T[]` diretamente para um `IList<T>`. Além de implementarem essas interfaces, os arrays também são definidos com os atributos `Serializable`, que permite que sejam serializados, e `ComVisible`, que permite que sejam acessados por componentes COM³.

³ Para mais informações acesse: <http://www.microsoft.com/com/default.msp>.

5.3 Strings

`String` é um tipo de dado utilizado para representar seqüências de caracteres de texto. Em linguagens como C, *strings* são representadas através de arranjos de elementos do tipo `char` que recebem tratamento especial pelos compiladores para tornar o seu uso mais prático e produtivo. Nesses casos, a linguagem faz uso do caractere `\0`, que representa o *byte* 0, para indicar o fim da *string*.

Nas linguagens Java e C#, o tipo `string` é um tanto mais elaborado, sendo representado por uma classe específica: `java.lang.String`, no caso da linguagem Java, e `System.String`, no caso da linguagem C#.

Em ambas as linguagens, o tipo `string` conta com tratamento especial, por parte dos ambientes de desenvolvimento, para que possa ser usada de modo mais prático, podendo ser instanciado diretamente através de seqüências de caracteres iniciadas e terminadas com o caractere aspas (`"`). O uso do caractere finalizador de *strings*, `\0`, usado nas linguagens C e C++, também não se aplica.

Outra característica que merece ser destacada é o fato de, em ambas as linguagens, *strings* serem objetos imutáveis, ou seja, sempre que se associa um valor a uma variável do tipo *string*, uma nova `string` é criada e passada para a variável, sendo as instâncias anteriores abandonadas. Essa característica cria a impressão de que *strings* são tipos simples, ou primitivos, e não objetos.

Exemplo (C#):

```

// Value Type:
int a = 0;
int b = a;
a = 1;
Console.Write(b);           //Imprime 0

// Reference Type:
public class MyClass {
    public MyClass(int f) {
        Field = f;
    }
    public int Field { get; set; };
}
MyClass a = new MyClass(0);
MyClass b = a;
a.Field = 1;
Console.Write(b.Field);     //Imprime 1

//String:
string a = "Hello World";
string b = a;
a = "Good Bye World";
Console.Write(b);           //Imprime "Hello World"

```

Exemplo 8. Strings

Em ambas as linguagens, o caractere `\` é reservado para a representação de caracteres especiais dentro das *strings*, como por exemplo: `\t`, que representa uma tabulação ou `\n`, que representa uma quebra de linha. Sendo assim, quando se deseja incluir o próprio caractere `\` em uma string, deve-se utiliza-lo em dobro, `\\`, para que o compilador entenda que não se trata do uso de um caractere especial. A linguagem C# oferece um outro recurso para os casos em que caracteres `\` estiverem presentes em uma determinada string, o operador `@`. *Strings* precedidas desse operador, não reconhecem o caractere `\` como indicador de caractere especial, utilizam dois caracteres aspas em seqüência (`""`) para representarem um único caractere aspas (`"`) e reconhecem as quebras de linha utilizadas.

Exemplos:

```
string a = "Happy birthday, Joel";           // Happy birthday, Joel
string b = @"Happy birthday, Joel";         // Happy birthday, Joel
string c = "hello \t world";                // hello world
string d = @"hello \t world";               // hello \t world
string e = "Joe said \"Hello\" to me";      // Joe said "Hello" to me
string f = @"Joe said \"Hello\" to me";     // Joe said "Hello" to me
string g = "\\server\share\file.txt";       // \\server\share\file.txt
string h = @"\\server\share\file.txt";     // \\server\share\file.txt
string i = "one\r\two\r\nthree";           // one
                                           // two
                                           // three

string j = @"one
two
three";                                     // one
                                           // two
                                           // three
```

Exemplo 9. Strings

Como já foi dito, *strings* são objetos. Uma consequência imediata disso, é o fato de que utilizar o operador igual (==) para verificar se duas *strings* representam a mesma sequência de caracteres, somente retornará verdadeiro se ambas as *strings* forem representadas pelo mesmo objeto em memória, o que o exemplo abaixo mostra não ser algo tão obvio:

Exemplo (Java):

```
public static void main(String[] args) throws IOException {
    String a = getHello() + getWorld();      //a = "Hello World"
    String b = getHello() + getWorld();      //b = "Hello World"
    System.out.println(a == b);              //False
    System.in.read();
}
private static String getHello() {
    return "Hello ";
}
private static String getWorld() {
    return "World";
}
```

Exemplo 10. Strings

De fato, esse comportamento é observado na linguagem Java, porém o mesmo não ocorre na linguagem C#.

Exemplo (C#):

```
public static void Main(string[] args) {
    string a = GetHello() + GetWorld(); //a = "Hello World"
    string b = GetHello() + GetWorld(); //b = "Hello World"
    Console.WriteLine(a == b);         //True
    Console.ReadKey();
}
private static string GetHello() {
    return "Hello ";
}
private static string GetWorld() {
    return "World";
}
```

Exemplo 11. Strings

Isso não ocorre na linguagem C# devido ao fato de a classe `System.String` ter sido implementada utilizando um recurso, denominado sobrecarga de operadores, através do qual o operador igual (`==`) foi modificado de modo a retornar o resultado da comparação entre os valores representados por cada objeto `string`, e não o resultado da comparação de seus endereços de memória.

Na linguagem Java, para se comparar o conteúdo de duas *strings*, deve-se utilizar o método `equals()`, ao passo que na linguagem C#, para se comparar os endereços de memória de dois objetos *strings*, deve-se utilizar o método estático `ReferenceEquals()` da classe `System.Object`.

Exemplo (Java):

```
public static void main(String[] args) throws IOException {
    String a = getHello() + getWorld(); //a = "Hello World"
    String b = getHello() + getWorld(); //b = "Hello World"
    System.out.println(a.equals(b));    //True - Mesmo valor
    System.in.read();
}
private static String getHello() {
    return "Hello ";
}
private static String getWorld() {
    return "World";
}
```

Exemplo 12. Strings

Exemplo (C#):

```
public static void Main(string[] args) {
    string a = GetHello() + GetWorld(); //a = "Hello World"
    string b = GetHello() + GetWorld(); //b = "Hello World"
    Console.WriteLine(
        String.ReferenceEquals(a, b)); //False - Referências distintas
    Console.ReadKey();
}
private static string GetHello() {
    return "Hello ";
}
private static string GetWorld() {
    return "World";
}
```

Exemplo 13. Strings

É necessário destacar também que ambas as linguagens são otimizadas para reconhecer strings literais coincidentes e reaproveitar as instâncias criadas em memória, o que explica o uso dos métodos privados dos exemplos anteriores.

Exemplo (Java):

```
String a = "Hello World"; //Instancia "Hello World"
String b = "Hello World"; //Reutiliza instância "Hello World"
System.out.println(a == b); //Imprime True
```

Exemplo 14. Strings

Exemplo (C#):

```
string a = "Hello World"; //Instancia "Hello World"
string b = "Hello World"; //Reutiliza instância "Hello World"
Console.WriteLine(
    String.ReferenceEquals(a,b)); //Imprime True
```

Exemplo 15. Strings

Além da classe `java.lang.String`, no caso da linguagem Java, e da classe `System.String`, no caso da linguagem C#, essas linguagens ainda fornecem um outro modo de trabalhar com *strings*, o qual se mostra mais eficiente em operações nas quais o volume de alterações de conteúdo das strings é grande. Na linguagem Java, a classe a ser usada nesses

casos é a `java.lang.StringBuilder`, sendo que na linguagem C#, a classe em questão é a `System.Text.StringBuilder`.

5.4 Enumerações

Independente de qual seja a linguagem de programação utilizada, em muitos momentos durante a implementação de um software, o uso de constantes simbólicas é recorrente. Em linguagens como o Pascal (ISO, 1990), é possível definir tipos de dados que representam um conjunto de valores constantes relacionados no seu contexto. Por exemplo, podemos definir um tipo de dado denominado `Naipes` para representar os possíveis valores para uma variável que armazene o naipe de uma carta de baralho, nesse caso, a definição desse tipo de dado se daria conforme o exemplo abaixo.

Exemplo:

```
Naipes = (Paus, Ouros, Copas, Espadas);
```

Exemplo 16. Enumerações

Esses tipos de dados são conhecidos como enumerações, ou simplesmente *enums*.

Desde o seu lançamento, a linguagem C# já suporta a definição de *enums*, já a linguagem Java só passou a dar suporte a esses tipos de dados a partir de sua versão 1.5.

Apesar de serem utilizados para propósitos semelhantes e de modos semelhantes, *enums* são implementados de maneira muito diferente pelas linguagens Java e C#.

5.4.1 Enums na Linguagem Java:

Na linguagem Java, sempre que um *enum* é declarado, implicitamente uma nova classe é criada para representá-lo.

Para simples entendimento, podemos supor o modelo apresentado no exemplo a seguir como sendo uma classe criada para representar o *enum* Naipes do exemplo anterior.

Exemplo:

Declaração do *enum*:

```
enum Naipes { PAUS, OUROS, COPAS, ESPADAS }
```

Uso:

```
Naipes naipes = Naipes.PAUS; //Constante acessada como um membro estático do enum.
```

Esboço da classe criada:

```
class Naipes extends Enum<Naipes> {  
    public static final Naipes PAUS = new Naipes();  
    public static final Naipes OUROS = new Naipes();  
    public static final Naipes COPAS = new Naipes();  
    public static final Naipes ESPADAS = new Naipes();  
}
```

Exemplo 17. Enums na linguagem Java

Algumas considerações importantes devem ser feitas com relação aos *enums* da linguagem

Java:

- As únicas instâncias dos *enums* são aquelas representadas por suas constantes. Não é permitida a criação explícita de outras instâncias.
- É possível declarar campos, métodos e até mesmo construtores para *enums*.

Exemplo:

```
public enum Coin {
    PENNY(1),
    NICKEL(5),
    DIME(10),
    QUARTER(25);
    Coin(int value) {
        this.value = value;
    }
    private final int value;
    public int value() {
        return value;
    }
}
```

Exemplo 18. Enums na linguagem Java

- Instâncias de *enums* são aceitos em blocos `switch`, sendo que nesse caso, a constante avaliada pode ser acessada diretamente pelo seu nome.

Exemplo:

```
Naipes naipes = getNaipes();
switch(naipes) {
    case PAUS: System.out.print("Carta de paus"); break;
    case OUROS: System.out.print("Carta de ouros"); break;
    case COPAS: System.out.print("Carta de copas"); break;
    case ESPADAS: System.out.print("Carta de espadas");
}
```

Exemplo 19. Enums na linguagem Java

5.4.2 Enums na Linguagem C#:

De maneira semelhante ao que acontece na linguagem Java, sempre que um *enum* é declarado, uma nova classe é criada para representá-lo, porém isso é feito de maneira bastante diferente.

O modelo apresentado no exemplo a seguir pode ser usado para ilustrar como seria essa classe na linguagem C#, considerando ainda o exemplo do *enum* `Naipes`.

Exemplo:

Declaração do *enum*:

```
enum Naipe { Paus, Ouros, Copas, Espadas }
```

Uso:

```
Naipe naipe = Naipe.Paus;
```

Esboço da classe criada:

```
class Naipe : Enum {
    public const int Paus = 0;
    public const int Ouros = 1;
    public const int Copas = 2;
    public const int Espadas = 3;
}
```

Exemplo 20. Enums na linguagem C#

Com esses exemplos, nota-se que a classe criada implicitamente pela linguagem C# para representar o *enum* em pouco se parece com a classe criada pela linguagem Java.

As seguintes observações devem ser feitas a respeito dos *enums* da linguagem C#:

- Todo *enum* possui um tipo interno (*underlying type* em inglês), o qual é usado para representar as constantes do *enum*. Esse tipo interno pode ser qualquer tipo inteiro, com exceção do tipo *char*, sendo que caso esse tipo não seja explicitamente especificado, o tipo *System.Int32* será usado implicitamente.

Exemplo:

```
enum MaioresInteiros : ulong {
    Byte = 255UL,
    UInt32 = 4294967296UL,
    UInt64 = 18446744073709551616UL
}
```

Exemplo 21. Enums na linguagem C#

- As classes criadas para representarem os *enums*, assim como no caso da linguagem Java, não permitem a criação de instâncias de um *enum*, sendo que nesse caso nenhuma instância jamais será criada, ficando disponíveis apenas as constantes estáticas de tal classe e os métodos herdados da classe `System.Enum`.
- Não é permitida a inclusão de campos, métodos ou construtores, nem mesmo estáticos, na declaração dos *enums*, porém a partir da versão 3.0 da linguagem, é possível criar métodos de extensão para *enums*, permitindo que tais métodos sejam associados às constantes dos *enums*, produzindo um efeito semelhante ao que se obtém através da atribuição de métodos aos *enums* da linguagem Java.

Exemplo:

```
enum Naipe { Paus, Ouros, Copas, Espadas }
static class NaipeExtension {
    public static Naipe Identidade(this Naipe naipe) {
        return naipe;
    }
}
Naipe identidade = Naipe.Paus.Identidade();
```

Exemplo 22. Enums na linguagem C#

Métodos de extensão é um dos assuntos tratados no próximo capítulo.

- Como as constantes dos *enums*, na linguagem C#, são instâncias de tipos inteiros, é natural que possam ser utilizadas em blocos `switch`, porém ao contrário do que ocorre na linguagem Java, tais constantes devem ser acessadas como campos estáticos do *enum*.

Exemplo:

```
Naipe naipe = GetNaipe();
switch(naipe) {
    case Naipe.Paus: Console.Write("Carta de paus"); break;
    case Naipe.Ouros: Console.Write("Carta de ouros"); break;
    case Naipe.Copas: Console.Write("Carta de copas"); break;
    case Naipe.Espadas: Console.Write("Carta de espadas");
}
}
```

Exemplo 23. Enums na linguagem C#

- Da mesma forma, é natural que seja possível realizar a conversão (*cast*) das constantes de um *enum* para seu tipo interno, ou a qualquer tipo que possa ser convertido para ele.

Exemplo:

```
enum Naipe : sbyte {
    Paus = -1,
    Ouros = 0,
    Copas = 1,
    Espadas = 2
}
enum Cedula {
    Um = 1,
    Dois = 2,
    Cinco = 5,
    Dez = 10,
    Vinte = 20,
    Cinquenta = 50,
    Cem = 100
}

byte b = (byte)Naipe.Paus;
Naipe n = (Naipe)b;

Cedula c = Cedula.Um;
Naipe m = (Naipe)c;
```

Exemplo 24. Enums na linguagem C#

- Uma outra consequência da natureza das constantes dos *enums* é a possibilidade de estourar os valores esperados. As atribuições do exemplo abaixo são válidas, embora não haja nenhuma constante do *enum* *Naipe* associada aos valores 5 e 50.

Exemplo:

```
Naipe n = (Naipe)5;
Naipe m = (Naipe)Cedula.Cinquenta
```

Exemplo 25. Enums na linguagem C#

- É possível criar *enums* que possam representar conjuntos de valores, utilizando-se o atributo `Flags` e seguindo-se algumas convenções, como definir uma constante `Nenhum` (*None*) associada ao valor zero e utilizar potências de dois como valores para as demais constantes. Nesse caso os operadores `&` e `|` podem ser usados para ler e escrever, respectivamente, múltiplos valores em uma instância do *enum*.

No exemplo abaixo, ao se atribuir o valor `OS.Windows | OS.Solaris` à variável `os`, na verdade o valor armazenado na variável é o valor 9, que corresponde à soma dos valores passados, por isso é necessário o uso de valores equivalentes a potências de dois e a exclusão do valor zero.

Exemplo:

```
[Flags]
enum OS {
    Nenhum = 0
    Windows = 1,
    Linux = 2,
    MacOS = 4,
    Solares = 8
}

OS os = OS.Windows | OS.Solaris;
Console.Write( (os & OS.Solaris) == OS.Solaris ); //Imprime verdadeiro
Console.Write( (os & OS.Linux) == OS.Linux);      //Imprime falso
```

Exemplo 26. Enums na linguagem C#

5.4.3 Operações com Enums:

Ambas as linguagens fornecem diversas operações destinadas a simplificar o uso dos *enums*.

Na linguagem Java, essas operações podem ser encontradas na classe `java.util.EnumSet`, enquanto na linguagem C# elas se encontram na classe `System.Enum`.

5.5 Definição de Tipos

A definição de novos tipos, em ambas as linguagens, ocorre de maneira semelhante, com a ressalva de que na linguagem C# é possível definir *structs*, ou *Value Types*, que são estruturas alocadas por valor, enquanto que na linguagem Java apenas classes podem ser definidas.

Os mecanismos de herança, polimorfismo, implementação de interfaces e atribuição de modificadores de acesso serão apresentados nas seções seguintes.

5.5.1 Membros de um Tipo.

A definição de um novo tipo é realizada através da escolha de um nome, uma lista de tipos base e um conjunto de membros. Um membro de um tipo pode ser um construtor, um método, um campo, uma propriedade ou um evento, sendo que esses dois últimos só se aplicam à linguagem C#.

As subseções desta seção apresentaram as diferenças com relação à definição de membros de tipos nas duas linguagens.

5.5.1.1 *Membros estáticos e membros de instância*

Uma definição que deve ser feita antes de se analisar os tipos de membros que compõem um tipo, é o conceito de membros estáticos.

Em geral, os membros de um tipo são definidos como membros de instância, mas em alguns casos é necessário definirmos membros estáticos, o que é feito através do uso do modificador `static`.

Há dois modelos mentais que podem ser utilizados para se definir o que são membros estáticos:

5.5.1.1.1 Primeiro modelo mental para definição de membros estáticos

Membros estáticos são membros compartilhados por todas as instâncias de um tipo.

Exemplo (Java):

```
class Test {
    static int x = 0;
}

Test a = new Test();
Test b = new Test();
a.x = 1;
System.out.println(b.x);           //Imprime 1 pois o valor é compartilhado pelas
                                   //instâncias do tipo Test
```

Exemplo 27. Primeiro modelo mental para definição de membros estáticos

Também é possível usar o tipo diretamente, para acessar os membros estáticos.

Exemplo:

```
class Test {
    static int x = 0;
}

Test.x = 1;
```

Exemplo 28. Primeiro modelo mental para definição de membros estáticos

Esse modelo mental se aplica melhor à linguagem Java, uma vez que o acesso aos membros estáticos através de instâncias não é permitido pela linguagem C#.

Exemplo (C#):

```
class Test {
    static int x = 0;
}

Test a = new Test();
Test b = new Test();
Test.x = 1;           //OK
a.x = 1;             //ERRO DE COMPILAÇÃO
```

Exemplo 29. Primeiro modelo mental para definição de membros estáticos

5.5.1.1.2 Segundo modelo mental para definição de membros estáticos

Membros estáticos são membros pertencentes ao tipo e não às instâncias do tipo.

Esse modelo ilustra melhor o modo como tais membros são tratados pela linguagem C# e também pode ser usado para a linguagem Java.

Uma ressalva deve ser feita para o caso de classes aninhadas estáticas da linguagem Java, que apesar de serem denominadas estáticas, não se enquadram nas definições anteriores.

Classes aninhadas serão tratadas separadamente na seção “Tipos Aninhados”.

5.5.1.2 Campos

Campos são variáveis ou constantes definidas como membros de um tipo, ou seja, que podem ser acessadas a partir das instâncias do tipo no qual foram definidas, ou diretamente a partir do tipo, caso tenham sido declaradas como estáticas.

Em geral adota-se a convenção, em ambas as linguagens, de não se criar campos com visibilidade pública, exceto em casos justificáveis, ou quando se tratarem de campos de valor constante.

Na linguagem C#, durante a inicialização de campos estáticos, é possível referenciar outros campos estáticos declarados posteriormente no mesmo tipo, o que não é permitido pela linguagem Java.

Exemplo (Java):

```
class Test {  
    static int a = 2 * b + 7;           //ERRO DE COMPILAÇÃO  
    static int b = 5;  
}
```

Exemplo 30. Campos

Exemplo (C#):

```
class Test {
    static int a = 2 * b + 7;           //OK
    static int b = 5;
}
```

Exemplo 31. Campos

5.5.1.2.1 Campos constantes e somente-leitura

Tanto a linguagem C# quanto a linguagem Java permitem a definição de campos de valor constante. A linguagem C# ainda permite a definição de campos somente-leitura, que apresentam um comportamento levemente diferenciado dos campos constantes.

5.5.1.2.1.1 Campos constantes na linguagem Java

Para a definição de um campo constante na linguagem Java, basta utilizar o modificador `final` antes da declaração do campo.

Exemplo:

```
class Test {
    final int a = 2;
    public Test() {
        a = 3;           //ERRO DE COMPILAÇÃO
    }
}
```

Exemplo 32. Campos constantes na linguagem Java

Também é possível utilizar esse modificador para declarar variáveis locais de valor constante.

Exemplo:

```
public void foo() {
    final int b = 2;
    b = 3;           //ERRO DE COMPILAÇÃO
}
```

Exemplo 33. Campos constantes na linguagem Java

5.5.1.2.1.2 Campos constantes e somente-leitura na linguagem C#

A definição de campos e variáveis locais de valor constante na linguagem C# é possível através do uso do modificador `const`.

Exemplo:

```
class Test {
    const int a = 2;
    public Test() {
        a = 3; //ERRO DE COMPILAÇÃO
    }

    public void Foo() {
        const int b = 2;
        b = 3; //ERRO DE COMPILAÇÃO
    }
}
```

Exemplo 34. Campos constantes e somente-leitura na linguagem C#

Já os campos somente-leitura são definidos utilizando-se o modificador `readonly`.

Exemplo:

```
class Test {
    readonly int a = 2;
    public Test() {
        a = 3; //OK
    }
    public void Foo() {
        a = 3; //ERRO DE COMPILAÇÃO
    }
}
```

Exemplo 35. Campos constantes e somente-leitura na linguagem C#

Campos somente-leitura diferem dos campos constantes ao permitirem que seu valor seja informado no construtor do tipo. O uso desse modificador permite a semântica de campos constantes a campos de tipos complexos.

Exemplo:

```
class Test {
    readonly MyClass a = null;
    public Test() {
        a = MyFactory.GetMyClass();           //OK
    }
    public void Foo() {
        a = null;                             //ERRO DE COMPILAÇÃO
    }
}
```

Exemplo 36. Campos constantes e somente-leitura na linguagem C#

5.5.1.3 Métodos

Métodos são as funções a serem executadas pelo tipo ou por suas instâncias.

Há poucas diferenças com relação à definição de métodos de um tipo nas duas linguagens, sendo que a maioria delas é abordada nas demais seções deste trabalho.

Um aspecto em especial, que deve ser mencionado, é o fato de que em ambas as linguagens, nas versões mais recentes, é possível passar implicitamente um arranjo variável de parâmetros para um método, e isso é feito de maneira levemente diferente pelas duas linguagens.

Exemplo:

```
class Test {
    A a = new A();
    void Foo() {
        a.Bar(1);           //Ao invés de a.Bar(new int[] { 1 });
        a.Bar(1, 2);        //Ao invés de a.Bar(new int[] { 1, 2 });
        a.Bar(1, 2, 3);     //Ao invés de a.Bar(new int[] { 1, 2, 2 });
    }
}
```

Exemplo 37. Métodos

Para permitir essa flexibilidade, o método em questão deve ser definido de modo que o seu último parâmetro seja um arranjo e utilize uma sintaxe especial. Esse recurso só pode ser utilizado para o último parâmetro do método.

5.5.1.3.1.1 Arranjos de parâmetros na linguagem Java

Para definir um arranjo de parâmetros em um método na linguagem Java, utiliza-se o operador reticências (. . .) após o tipo do parâmetro, sem utilizar a notação de arranjo.

Exemplo:

```
class A {
    void bar(int... args) {
        for (int arg : args) {
            //Do something
        }
    }
}
```

Exemplo 38. Arranjos de parâmetros na linguagem Java

5.5.1.3.1.2 Arranjos de parâmetros na linguagem C#

No caso da linguagem C#, o parâmetro deve ser definido como um arranjo, porém recebendo o modificador `params` antes de sua declaração.

Exemplo:

```
class A {
    void Bar(params int[] args) {
        foreach (int arg in args) {
            //Do something
        }
    }
}
```

Exemplo 39. Arranjos de parâmetros na linguagem C#

5.5.1.4 Propriedades

Propriedades são membros semelhantes a campos, mas implicitamente dotados de métodos utilizados para ler e escrever o seu valor.

5.5.1.4.1 Propriedades na linguagem Java:

Na linguagem Java, não há o conceito explícito de propriedade, sendo que tal funcionalidade é obtida através do uso de um padrão composto por um campo privado ou protegido e dois

métodos, um denominado *getter* e outro denominado *setter*, destinados a permitir a leitura e a escrita sobre esse campo, respectivamente.

Exemplo:

```
class Test {
    private int x;
    int getX() {
        return x;
    }
    void setX(int x) {
        this.x = x;
    }
}

Test test = new Test();
test.setX(1);
int x = test.getX();
```

Exemplo 40. Propriedades na linguagem Java

Em situações onde for conveniente, apenas um desses métodos pode existir, ou podem ser utilizados modificadores de acesso diferenciados para cada um deles, de modo a controlar melhor o acesso ao campo. É possível também que não haja um campo e que o valor da propriedade seja calculado ao ser lido.

Um dos recursos amplamente discutidos para serem ou não incluídos nas versões futuras da linguagem Java é justamente o suporte explícito ao uso de propriedades, de maneira semelhante ao que ocorre na linguagem C#, porém essa inclusão poderia trazer problemas de compatibilidade com as versões anteriores da linguagem, além de receber forte rejeição de grande parte da comunidade de desenvolvedores da linguagem.

5.5.1.4.2 Propriedades na linguagem C#

Na linguagem C# o conceito de propriedades faz parte da linguagem desde a sua definição, sendo que é possível ter um controle completo sobre a visibilidade de leitura e escrita dessas propriedades, sem que seja necessário o uso de métodos *getter* e *setter* explícitos, que são apontados como os grandes poluidores de código na linguagem Java.

Exemplo:

```
class Test {
    private int x;
    public int X {
        get {
            return x;
        }
        set {
            x = value;
        }
    }
}

Test test = new Test();
test.X = 1;
int x = test.X;
```

Exemplo 41. Propriedades na linguagem C#

A partir da versão 3.0 da linguagem C#, o uso de propriedades se tornou ainda mais prático, através da inclusão do recurso propriedades automáticas, que permite que os blocos `get` e `set` de tais propriedades, além do campo privado utilizado para armazenar seu valor, sejam definidos implicitamente pelo compilador.

Exemplo:

```
class Test {
    public int X { get; set; } //Define a propriedade X automaticamente
}

Test test = new Test();
test.X = 1;
int x = test.X;
```

Exemplo 42. Propriedades na linguagem C#

Note que é possível, embora não recomendado, que se utilizem apenas métodos e campos para definir propriedades na linguagem C#, exatamente da mesma forma como é feito na linguagem Java.

5.5.1.4.3 Propriedades indexadas

A linguagem C# permite a definição de uma propriedade indexada em cada tipo. Tal propriedade faz com que o tipo se comporte como uma coleção de itens e permite o acesso a seus itens de modo semelhante a um arranjo.

Exemplo (C#):

```
//Define lista de inteiros
class MyIntList {
    //Armazena inteiros
    int[] itens = new int[0];
    //Define propriedade indexada da lista
    public int this[int index] {
        get {
            //Obtém item ou lança exceção
            if (index < itens.Length) {
                return itens[index];
            }
            throw new ArgumentException("index");
        }
        set {
            //Atribui item
            if (index >= itens.Length) {
                int[] n = new int[index+1];
                for (int i = 0; i < itens.Length; i++ ) {
                    n[i] = itens[i];
                }
                itens = n;
            }
            itens[index] = value;
        }
    }
    //Retorna número de itens da lista
    public int Count {
        get {
            return itens.Length;
        }
    }
}

MyIntList m = new MyIntList();           //Instancia novo MyListInt em m
m[3] = 0;                                 //Altera tamanho para 3 e atribui valor 0
m[1] = m[3];                              //Atribui valor da posição 3 na posição 1
m[5] = 2;                                 //Altera tamanho para 5 e atribui valor 2
```

Exemplo 43. Propriedades indexadas

Apesar de apenas uma propriedade indexada ser permitida em cada tipo, é possível sobrecarregá-la:

Exemplo (C#):

```
class MyClassList {
    public MyClass this[int index] { ... }
    public MyClass this[string firstname] { ... }
    public MyClass this[string firstname, string lastname] { ... }
}

MyClassList l = GetMyClassList();
MyClass x = l[0];
MyClass y = l["Rafael"];
MyClass y = l["Rafael", "Romão"];
```

Exemplo 44. Propriedades indexadas

A linguagem Java não oferece nenhum recurso semelhante, porém utiliza, como convenção, métodos `get()` e `set()` para implementar tal funcionalidade.

Exemplo (Java):

```
//Define lista de inteiros
class MyIntList {
    //Armazena inteiros
    int[] itens = new int[0];
    //Define método get
    public int get(int index) {
        //Obtém item ou lança exceção
        if (index < itens.length) {
            return itens[index];
        }
        throw new ArgumentException("index");
    }
    //Define método set
    public void set(int index, int value) {
        //Atribui item
        if (index >= itens.length) {
            int[] n = new int[index+1];
            for (int i = 0; i < itens.length; i++) {
                n[i] = itens[i];
            }
            itens = n;
        }
        itens[index] = value;
    }
    //Retorna número de itens da lista
    public int getCount() {
        return itens.length;
    }
}

MyIntList m = new MyIntList(); //Instancia novo MyListInt em m
m.set(3,0); //Altera tamanho para 3 e atribui valor 0
m.set(1,m.get(3)); //Atribui valor da posição 3 na posição 1
m.set(5,2); //Altera tamanho para 5 e atribui valor 2
```

Exemplo 45. Propriedades indexadas

5.5.1.5 *Eventos*

Eventos são membros especiais utilizados para permitir que observadores externos sejam notificados quando determinados eventos ocorrerem dentro de um tipo. Eventos só podem ser considerados membros de tipo na linguagem C#.

Esse recurso será detalhado na seção “Tratamento de Eventos”.

5.5.2 **Tipos Estáticos**

O conceito de membros estáticos está presente em ambas as linguagens, como já descrito, porém existe também, em ambas as linguagens, o conceito de tipos estáticos, ou mais especificamente classes estáticas, uma vez que esse conceito não se aplica aos structs da linguagem C#. Porém, o conceito de classe estática na linguagem Java, em nada se assemelha com o conceito de classe estática da linguagem C#.

Na linguagem Java, o termo classe estática é utilizado para se referir a classes aninhadas estáticas. Tais classes são ditas aninhadas por serem definidas dentro de uma outra classe, e estáticas por terem acesso somente aos membros estáticos da classe dentro da qual está definida.

Na linguagem C#, o termo classe estática é utilizado para se referir a classes, aninhadas ou não, que tenham sido definidas utilizando-se o modificador `static`. Tais classes devem, obrigatoriamente, possuir apenas membros estáticos, inclusive os construtores, tendo como consequência, a impossibilidade de serem instanciadas.

Exemplo:

```
static class Test1 {
    int x; //ERRO DE COMPILAÇÃO
}

static class Test2 {
    static int x; //OK
}

Test2 test = new Test2(); //ERRO DE COMPILAÇÃO
```

Exemplo 46. Tipos estáticos

5.6 Interfaces

Interfaces são contratos de definição de tipos que estabelecem um conjunto de métodos, propriedades e eventos que devem, obrigatoriamente, ser implementado pelas classes que desejam cumprir tal contrato.

Exemplo (Java):

```
interface Intf {
    void foo();
    void bar();
}

class Test implements Intf {
    void foo() { ... } //Deve existir, obrigatoriamente
    void bar() { ... } //Deve existir, obrigatoriamente
}
```

Exemplo 47. Interfaces

O principal aspecto referente a interfaces que difere as duas linguagens é a possibilidade, na linguagem C#, de se realizar a implementação de um método para uma interface específica, o que é conhecido como implementar a interface explicitamente.

Exemplo (C#):

```
interface Intf1 {
    void Foo();
}

interface Intf2 {
    void Foo();
}

class Test1 : Intf1, Intf2 {
    void Foo() { ... } //Implementa Foo() de ambas as interfaces
}

class Test2 : Intf1, Intf2 {
    void Intf1.Foo() { ... } //Implementa Foo() da interface Intf1
    void Intf2.Foo() { ... } //Implementa Foo() da interface Intf2
}
```

Exemplo 48. Interfaces

As conseqüências disso são:

- É possível dar tratamento diferenciado a um método quando sua chamada for feita através de uma interface específica.
- Métodos implementados dessa forma não podem ser invocados caso não seja feita uma conversão para a interface desejada.

Exemplo:

```
interface Intf1 {
    void Foo();
}

interface Intf2 {
    void Foo();
}

class Test1 : Intf1, Intf2 {
    void Foo() { ... } //Implementa Foo() de ambas as interfaces
}

class Test2 : Intf1, Intf2 {
    void Intf1.Foo() { ... } //Implementa Foo() da interface Intf1
    void Intf2.Foo() { ... } //Implementa Foo() da interface Intf2
}

Test1 a = new Test1();
a.Foo(); //Chama método Test1.Foo()

Test2 b = new Test2();

Intf1 x = (Intf1)b;
x.Foo(); //Chama método Intf1.Foo()

Intf2 y = (Intf2)b;
y.Foo(); //Chama método Intf2.Foo()

b.Foo(); //ERRO DE COMPILAÇÃO
```

Exemplo 49. Interfaces

5.7 Herança e Polimorfismo

Herança e polimorfismo são conceitos inerentes às linguagens orientadas a objetos. De modo resumido, podemos dizer que o termo herança se refere à capacidade de se criar um tipo Y que herde os membros definidos em um outro tipo X, de modo que podemos dizer que o tipo Y estende o tipo X. Já o mecanismo de polimorfismo permite, entre outras coisas, que se utilizem instâncias de um tipo Y, tal que Y estenda X, em pontos onde se espera o uso de instâncias de um tipo X.

Exemplo (Java):

```
class Animal {
    void andar() {
        System.out.println("Andei!");
    }
}
class Cachorro extends Animal {
    void latir() {
        System.out.println("Au Au!");
    }
}
Animal pet = new Cachorro(); //Instancia um Cachorro e atribui a uma
                             //variável do tipo Animal
pet.andar();                //Pede que o animal ande
pet.latir();                //Erro de compilação, pois pet foi definido
                             //como Animal e por isso os membros de
                             //Cachorro não são diretamente acessíveis
```

Exemplo 50. Herança e Polimorfismo

Com relação a essas duas características, tanto a linguagem Java quanto a linguagem C# fornecem um suporte semelhante, mesmo que diferentemente implementados em alguns pontos. Ambas as linguagens não suportam herança múltipla, possuem as mesmas possibilidades de uso polimórfico e dão um suporte muito semelhante ao uso de interfaces.

Uma sutil diferença que deve ser destacada, com relação à implementação de herança de tipos nas duas linguagens, é o modo como se define qual tipo será estendido.

5.7.1 Estendendo um Tipo na Linguagem Java:

Para estender um tipo na linguagem Java, utiliza-se o comando `extends`, para estender uma classe, ou o comando `implements`, para se implementar uma ou mais interfaces.

Exemplo:

```
class Cachorro extends Animal implements IAnimal, IBicho {
    /*...*/
}
```

Exemplo 51. Estendendo um tipo na linguagem Java

O comando `extends` deve sempre preceder o comando `implements` e apenas um tipo pode ser especificado após o comando `extends`.

5.7.2 Estendendo um Tipo na Linguagem C#:

Na linguagem C#, o comando utilizado para especificar herança de tipos é o comando dois pontos (:), tanto para especificar o tipo base como as interfaces a serem implementadas.

Exemplo:

```
class Cachorro : Animal, IAnimal, IBicho {
    /*...*/
}
```

Exemplo 52. Estendendo um tipo na linguagem C#

O tipo base deve sempre ser o primeiro parâmetro especificado após o operador dois pontos (:).

Um outro fator a ser destacado é o fato de apenas classes suportarem herança. Apesar disso, *structs* podem implementar interfaces, o que é definido utilizando-se a mesma sintaxe.

Exemplo:

```
struct Animal : IAnimal, IBicho {
    /*...*/
}
```

Exemplo 53. Estendendo um tipo na linguagem C#

Structs também não podem ser utilizados como tipo base para classes.

5.7.3 Herança e Sobrescrita de Métodos

Ao se definir um tipo que estende um tipo base, todos os membros, campos, métodos e propriedades, que tenham sido definidos com escopo superior a privado, são herdados pelo novo tipo, ou seja, podem ser utilizados como se fossem parte da definição desse tipo.

Uma dos principais recursos disponibilizados através de herança e polimorfismo é a possibilidade de se definir diferentes implementações para um mesmo método no tipo base e

nos seus tipos derivados. Esse recurso é conhecido como sobrescrita de métodos (*method overriding* em inglês).

Para melhor analisarmos esse recurso, é conveniente separarmos os métodos em categorias, conforme sua situação em relação à sobrescrita.

5.7.3.1 Métodos finais, ou selados

Métodos finais, na linguagem Java, ou selados, na linguagem C#, são métodos que não podem ser sobrescritos pelos tipos derivadas.

5.7.3.1.1 Métodos finais na linguagem Java

Métodos finais na linguagem Java são definidos através do uso do modificador `final` imediatamente antes do tipo de retorno do método.

Exemplo:

```
class Animal {
    public final void andar() {                //Não pode ser sobrescrito
        System.out.println("Andei!");
    }
}
```

Exemplo 54. Métodos finais na linguagem Java

Além dos métodos definidos com o modificador `final`, aqueles que forem definidos com o escopo privado também não poderão ser sobrescritos.

Exemplo:

```
class Animal {
    private void andar() {                    //Não pode ser sobrescrito
        System.out.println("Andei!");
    }
}
```

Exemplo 55. Métodos finais na linguagem Java

5.7.3.1.2 Métodos selados na linguagem C#

Na linguagem C#, todo método é selado por padrão.

Exemplo:

```
class Animal {
    public void Andar() {
        Console.WriteLine("Andei!");
    }
}
```

//Não pode ser sobrescrito

Exemplo 56. Métodos selados na linguagem C#

5.7.3.2 Métodos virtuais

Métodos virtuais são métodos que podem ser sobrescritos pelos tipos derivados. Esses métodos recebem um tratamento diferenciado pelo ambiente de execução para possibilitar a verificação de qual versão do método deve ser executada quando uma chamada polimórfica é realizada.

5.7.3.2.1 Métodos virtuais na linguagem Java

Por padrão, todo método é virtual na linguagem Java.

Exemplo:

```
class Animal {
    public void andar() {
        System.out.println("Andei!");
    }
}
```

//Pode ser sobrescrito

Exemplo 57. Métodos virtuais na linguagem Java

5.7.3.2.2 Métodos virtuais na linguagem C#

Para tornar um método virtual na linguagem C#, o modificador `virtual` deve ser utilizado imediatamente antes do tipo de retorno do método.

Exemplo:

```
class Animal {
    public virtual void Andar() {
        Console.WriteLine("Andei!");
    }
}
```

Exemplo 58. Métodos virtuais na linguagem C#

Métodos de escopo privado não podem ser definidos como virtuais.

Exemplo:

```
class Animal {
    private virtual void Andar() {
        Console.WriteLine("Andei!");
    }
}
```

Exemplo 59. Métodos virtuais na linguagem C#

5.7.3.3 Métodos abstratos

Métodos abstratos são métodos que devem, obrigatoriamente, ser sobrescritos pelos tipos derivados, a menos que os mesmos também sejam definidos como abstratos. Além disso, métodos abstratos só podem ser definidos em tipos que também tenham sido definidos como abstratos.

Para definir um tipo como abstrato, basta utilizar o modificador `abstract` imediatamente antes do comando `class`, ao passo que para definir um método como abstrato, basta utilizar o modificador `abstract` imediatamente antes do tipo de retorno do método.

Métodos abstratos não possuem corpo.

Exemplo (Java):

```
abstract class Animal {
    public abstract void andar();
}
```

Exemplo 60. Métodos abstratos

Exemplo (C#):

```
abstract class Animal {
    public abstract void Andar();
}
```

Exemplo 61. Métodos abstratos

Métodos de escopo privado não podem ser definidos como abstratos.

Todo método abstrato é também virtual.

5.7.3.4 Métodos sobrescritos

Métodos sobrescritos são métodos que foram definidos em um tipo e modificados na definição de um tipo derivado. Nesses casos, é possível que o método definido no tipo base também seja executado quando conveniente.

5.7.3.4.1 Métodos sobrescritos na linguagem Java

Na linguagem Java, para que um método seja considerado sobrescrito, basta que haja um outro método com mesmo nome e mesmo conjunto de parâmetros na classe base do tipo.

Exemplo:

```
class Animal {
    public void andar() {
        System.out.println("Andei como um animal!");
    }
}

class Cachorro extends Animal {
    public void andar() {
        System.out.println("Andei como um cachorro!");
    }
}
```

Exemplo 62. Métodos sobrescritos na linguagem Java

Para invocar o método sobrescrito no tipo base, utiliza-se o comando `super`.

Exemplo:

```
class Animal {
    public void andar() {
        System.out.println("Andei como um animal!");
    }
}

class Cachorro extends Animal {
    public void andar() {
        super.andar();
        System.out.println("Andei como um cachorro!");
    }
}
```

Exemplo 63. Métodos sobrescritos na linguagem Java

A partir da versão 1.5 da linguagem Java, foi incluída a anotação `Override`. Essa anotação foi projetada para ser usada como um marcador para métodos sobrescritos, de modo que, caso a mesma seja usada em um método que não possua um equivalente não privado na classe base, o ambiente de desenvolvimento notificará o desenvolvedor sobre tal situação no momento da compilação.

Exemplo:

```
class Animal {
    public void andar() { ... }
}

class Cachorro extends Animal {
    @Override
    public void andar() { ... } //Correto. Método sobrescrito
    @Override
    public void nadar() { ... } //Errado. Método não sobrescrito
}
```

Exemplo 64. Métodos sobrescritos na linguagem Java

Anotações serão apresentadas posteriormente na seção “Anotações ou Atributos”.

5.7.3.4.2 Métodos sobrescritos na linguagem C#

Na linguagem C#, para que um método seja considerado sobrescrito, é necessário que haja um método com mesmo nome e conjunto de parâmetros no tipo base e que tal método tenha sido

definido com o modificador `virtual`. Além disso, deve se utilizar o modificador `override` imediatamente antes do tipo de retorno do método sobrescrito.

Exemplo:

```
class Animal {
    public virtual void Andar() { //Método virtual
        Console.WriteLine("Andei como um animal!");
    }
}

class Cachorro : Animal {
    public override void Andar() { //Método sobrescrito
        Console.WriteLine("Andei como um cachorro!");
    }
}
```

Exemplo 65. Métodos sobrescritos na linguagem C#

Caso seja definido um método sobrescrito sem que haja um método virtual equivalente na classe base, ou um método selado para o qual exista um método virtual ou selado equivalente na classe base, ocorrerá um erro de compilação.

Exemplo:

```
class Animal {
    public virtual void Andar() { //Método virtual
        Console.WriteLine("Andei como um animal!");
    }
    public void Nadar() { //Método selado
        Console.WriteLine("Nadei como um animal!");
    }
}

class Cachorro : Animal {
    public void Andar() { //ERRO DE COMPILAÇÃO
        Console.WriteLine("Andei como um cachorro!");
    }
    public override void Nadar() { //ERRO DE COMPILAÇÃO
        Console.WriteLine("Nadei como um cachorro!");
    }
}
```

Exemplo 66. Métodos sobrescritos na linguagem C#

Para invocar o método sobrescrito na classe base, utiliza-se o comando `base.`

Exemplo:

```
class Animal {
    public void Andar() {
        Console.WriteLine("Andei como um animal!");
    }
}

class Cachorro : Animal {
    public override void Andar() {
        base.Andar();
        Console.WriteLine("Andei como um cachorro!");
    }
}
```

Exemplo 67. Métodos sobrescritos na linguagem C#

Todo método sobrescrito é também virtual, a menos que o mesmo seja definido também como selado.

Exemplo:

```
class Animal {
    public virtual void Andar() {
        Console.WriteLine("Andei como um animal!");
    }
}

class Cachorro : Animal {
    public sealed override void Andar() {
        Console.WriteLine("Andei como um cachorro!");
    }
}
```

Exemplo 68. Métodos sobrescritos na linguagem C#

5.7.3.5 Métodos reintroduzidos

Métodos reintroduzidos são métodos que, apesar de encontrados no tipo base, não foram definidos como virtuais e ainda assim necessitam ser sobrescritos. É um recurso que viola os interesses dos construtores do tipo base, porém em alguns casos pode ser necessário.

A linguagem Java não suporta métodos reintroduzidos, já a linguagem C# disponibiliza tal recurso através do modificador `new`.

Exemplo:

```
class Animal {
    public void Andar() {
        Console.WriteLine("Andei como um animal!");
    }
}

class Cachorro : Animal {
    public new void Andar() {
        Console.WriteLine("Andei como um cachorro!");
    }
}
```

Exemplo 69. Métodos reintroduzidos

É importante destacar a diferença prática entre sobrescrever um método virtual e reintroduzir um método não virtual. O exemplo abaixo ilustra essa situação.

Exemplo:

```
class Animal {
    public virtual void Correr() {
        Console.WriteLine("Corri como um animal!");
    }
    public void Andar() {
        Console.WriteLine("Andei como um animal!");
    }
}

class Cachorro : Animal {
    public override void Correr() {
        Console.WriteLine("Corri como um cachorro!");
    }
    public new void Andar() {
        Console.WriteLine("Andei como um cachorro!");
    }
}

Animal animal = new Cachorro();
animal.Correr(); //Imprime: "Corri como um cachorro!"
animal.Andar(); //Imprime: "Andei como um animal!"

Cachorro cachorro = new Cachorro();
cachorro.Correr(); //Imprime: "Corri como um cachorro!"
cachorro.Andar(); //Imprime: "Andei como um cachorro!"
```

Exemplo 70. Métodos reintroduzidos

5.8 Construtores e Finalizadores

Frequentemente é interessante que o desenvolvedor, ao implementar um novo tipo, tenha um modo de executar rotinas de inicialização sempre que uma nova instância desse tipo for

criada. Em outros casos menos comuns, é também interessante que haja um modo de se executar rotinas de finalização, sempre que uma instância é liberada da memória.

Ambas as linguagens, C# e Java, permitem que tais rotinas sejam implementadas.

5.8.1 Construtores

Em ambas as linguagens, construtores são implementados através de métodos especiais, os quais não possuem a especificação de um tipo de retorno e cujo nome deve ser, obrigatoriamente, o mesmo nome do tipo. Dentro do corpo de um método construtor é possível realizar operações de inicialização dos membros da nova instância criada, receber parâmetros para serem usados nessas operações e invocar a execução de um outro construtor da mesma classe ou de uma classe base a qual o tipo em questão estenda.

Para todo tipo definido, ao menos um construtor sempre estará presente, sendo que mais de um construtor, desde que com parâmetros diferentes, podem ser definidos. Nos casos em que nenhum construtor seja explicitamente definido pelo desenvolvedor, um construtor padrão será criado implicitamente pelo compilador, o qual não possuirá parâmetros e terá como única atribuição invocar o construtor padrão da classe base do tipo. Caso um ou mais construtores sejam definidos, o construtor padrão não será criado e como consequência disso, outros tipos que vierem a estender o tipo em questão deverão, obrigatoriamente, implementar algum construtor explicitamente.

5.8.1.1 Construtores na linguagem Java

No que se refere a uma comparação com os construtores da linguagem C#, o único aspecto relevante é o modo como é feita a chamada a um outro construtor, seja ele do tipo em questão ou do tipo base.

5.8.1.1.1 *Invocando um construtor do tipo base*

Para invocar um construtor do tipo base, utiliza-se o comando `super()`, o qual deve ser, obrigatoriamente, o primeiro comando a ser executado pelo construtor e deverá receber um conjunto de parâmetros equivalente àqueles esperados pelo construtor da classe base o qual se deseja invocar.

Exemplo:

```
class Animal {
    int codigo;
    Animal(int codigo) {
        this.codigo = codigo;
    }
}
class Cachorro extends Animal {
    Cachorro() {
        super(1);           //Invoca o construtor da classe Animal passando o
        /*...*/             //inteiro 1 como parâmetro
    }
}
```

Exemplo 71. Invocando um construtor do tipo base

5.8.1.1.2 *Invocando um outro construtor do tipo em questão*

Para invocar um outro construtor do tipo em questão, utiliza-se o comando `this()`, o qual deve ser usado seguindo-se as mesmas regras de uso do comando `super()`, porém o conjunto de parâmetros deve ser equivalente ao conjunto de parâmetros esperado por um dos demais construtores do tipo, o qual se deseja invocar.

Exemplo:

```
class Animal {
    int codigo;
    Animal(int codigo) {
        this.codigo = codigo;
    }
    Animal() {
        this(0);           //Invoca um outro construtor da classe Animal
        /*...*/           //passando o inteiro 0 como parâmetro
    }
}
```

Exemplo 72. Invocando um outro construtor do tipo em questão

5.8.1.2 Construtores na linguagem C#

Os construtores na linguagem C# diferem dos construtores da linguagem Java apenas no modo como se deve implementar a chamada para um outro construtor do tipo em questão e para um construtor do tipo base.

5.8.1.2.1 Invocando um construtor do tipo base na linguagem C#

Para invocar um construtor do tipo base, utiliza-se o comando `base()`, o qual deverá ser precedido pelo operador `:` e ser informado antes da abertura do bloco do construtor, diferentemente do comando `super()` da linguagem Java. Esse comando deverá receber um conjunto de parâmetros equivalentes àqueles esperados pelo construtor da classe base que se deseja invocar.

Exemplo:

```
class Animal {
    int codigo;
    Animal(int codigo) {
        this.codigo = codigo;
    }
}
class Cachorro : Animal {
    Cachorro()
        : base(1) {           //Invoca o construtor da classe Animal passando o
        /*...*/              //inteiro 1 como parâmetro
    }
}
```

Exemplo 73. Invocando um construtor do tipo base na linguagem C#

5.8.1.2.2 *Invocando um outro construtor do tipo em questão*

Para invocar um outro construtor do tipo em questão, utiliza-se o comando `this()`, o qual deve ser usado seguindo-se as mesmas regras de uso do comando `base()`, porém o conjunto de parâmetros deve ser equivalente ao conjunto de parâmetros esperado por um dos demais construtores do tipo, assim como na linguagem Java.

Exemplo:

```
class Animal {
    int codigo;
    Animal(int codigo) {
        this.codigo = codigo;
    }
    Animal()
        : this(0) {           //Invoca um outro construtor da classe Animal
        /*...*/              //passando o inteiro 0 como parâmetro
    }
}
```

Exemplo 74. Invocando um outro construtor do tipo em questão

5.8.1.3 *Construtores estáticos*

Em alguns cenários, é necessário que os campos estáticos de um tipo sejam inicializados quando esse tipo for carregado para a memória. Para suprir tal necessidade, a linguagem Java

oferece um recurso conhecido como bloco estático (static block em inglês), enquanto o C# fornece a possibilidade de se implementar um construtor estático para cada tipo.

5.8.1.3.1 Bloco estático na linguagem Java

Para definir um bloco de comandos a serem executados no momento em que um tipo for carregado para a memória, utiliza-se o comando `static` como se fosse um método sem nome, parâmetros ou tipo de retorno.

Exemplo:

```
class A {  
    static {  
        System.out.println("Inicializando tipo A");  
    }  
}
```

Exemplo 75. Bloco estático na linguagem Java

5.8.1.3.2 Construtores estáticos na linguagem C#

Para obter o mesmo resultado na linguagem C#, utiliza-se construtores estáticos, os quais são definidos de modo semelhante a um construtor normal, porém utilizando-se o modificador `static`.

Exemplo:

```
class A {  
    static A() {  
        Console.WriteLine("Inicializando tipo A");  
    }  
}
```

Exemplo 76. Construtores estáticos na linguagem C#

Construtores estáticos não podem receber parâmetros e apenas um construtor estático é permitido em cada tipo.

5.8.2 Finalizadores

Finalizadores são métodos especiais, os quais são executados automaticamente pelo ambiente de execução quando uma instância de um tipo é removida da memória.

Em ambas as linguagens, quando todas as referências a uma instância de um tipo perdem o escopo, essa instância passa a ser considerada lixo de memória, que posteriormente poderá ser recolhido pelo mecanismo coletor de lixo (*garbage collector* em inglês).

5.8.2.1 Finalizadores na linguagem Java

Os finalizadores, na linguagem Java, são definidos através de um método virtual implementado na classe `Object` e que, quando sobrescrito, executa a função de finalizador.

Exemplo:

```
class Animal {
    Animal() {
        /*...*/
    }
    public override void finalize() {
        /*...*/
        super.finalize();
    }
}
```

Exemplo 77. Finalizadores na linguagem Java

Quando uma instância de um tipo para o qual um finalizador tenha sido implementado for recolhida da memória pelo mecanismo coletor de lixo, tal finalizador será executado.

Caso uma nova referência para a instância corrente seja atribuída dentro do seu método finalizador, esse método não mais será chamado pelo coletor de lixo.

Exemplo:

```
class Animal {
    public static Animal referencia = null;
    public override void finalize() {
        Animal.referencia = this; //Ressucita a instância e impede
                                   //que o finalizador seja
                                   //executado novamente
        super.finalize();         //Invoca o finalizador da classe base
    }
}
```

Exemplo 78. Finalizadores na linguagem Java

5.8.2.2 Finalizadores na linguagem C#

Há duas formas de implementar um finalizador na linguagem C#. Uma delas é através do uso da interface `IDisposable`.

Essa interface possui um único método, denominado `Dispose`, que é invocado pelo ambiente de execução no momento em que as instâncias dos tipos que implementam tal interface são removidas da memória pelo coletor de lixo.

Exemplo:

```
class Animal : IDisposable {
    Animal() {
    }
    Dispose() { //Método Dispose()
    }
}
```

Exemplo 79. Finalizadores na linguagem C#

A outra forma de implementar um finalizador é usar um bloco de finalização, também conhecido como destrutor.

Exemplo:

```
class Animal {
    Animal() {
    }
    ~Animal() { //Bloco de finalização
    }
}
```

Exemplo 80. Finalizadores na linguagem C#

O bloco de finalização do tipo base é executado automaticamente após a execução do bloco de finalização do tipo em questão e apenas um bloco de finalização pode ser definido para cada tipo.

Por traz dos panos, o que o compilador C# realmente faz para implementar os blocos de finalização é sobrescrever o método `Finalize()` do tipo `System.Object`, de modo semelhante ao que ocorre na linguagem Java, porém a sobrescrita ou a chamada direta desse método não é permitida.

Nenhuma das duas linguagens suporta finalizadores estáticos.

5.9 Modificadores de Acesso

Modificadores de acesso são palavras reservadas da linguagem utilizadas para modificar o comportamento de um tipo ou dos membros desse tipo.

Alguns dos modificadores de acesso, relacionados à herança de métodos, já foram apresentados na seção anterior. Nesta seção serão apresentados aqueles utilizados para modificar a visibilidade de um tipo ou membros de um tipo.

5.9.1 Modificadores de Acesso da Linguagem Java

Os modificadores de acesso aplicáveis às declarações de tipos e membros de um tipo, na linguagem Java, são:

5.9.1.1 *Modificador public*

O modificador `public` define que o tipo ou membro ao qual se aplica não possui nenhum tipo de restrição de visibilidade.

Exemplo:

Arquivo: Planta.java

```
package plantas;

public class Planta {
    public void florescer() {
        System.out.println("Floresci!");
    }
}
```

Arquivo: Animal.java

```
package bichos;

public class Animal {
    public Planta alimentacaoVegetal;           //Planta está acessível
    public void andar() {
        System.out.println("Andei!");
    }
}
```

Exemplo 81. Modificador public

5.9.1.2 *Modificador padrão*

Ao se definir um tipo ou membro de um tipo sem informar um modificador de acesso, o modificador padrão é utilizado. Esse modificador define que o tipo ou membro pode ser acessado apenas de dentro do mesmo pacote.

Exemplo:

Arquivo: Animal.java

```
package bichos;

class Animal {
    void andar() {
        System.out.println("Andei!");
    }
}
```

Arquivo: Cachorro.java

```
package bichos;

class Cachorro extends Animal { //Animal está acessível
    void latir() {
        System.out.println("Au Au!");
    }
}
```

Exemplo 82. Modificador padrão

5.9.1.3 Modificador *protected*

O modificador `protected` define que um membro de um tipo pode ser acessado de qualquer ponto de dentro do mesmo pacote ou a partir de tipos derivados do tipo em questão. Esse modificador não se aplica a tipos, apenas a membros de tipos.

Exemplo:

Arquivo: Cachorro.java

```
package bichos;

public class Cachorro extends Animal {
    protected void latir () {
        System.out.println("Au Au!");
    }
}
```

Arquivo: Cachorro.java

```
package bichos;

public class Poodle extends Cachorro {
    public void latirMuito() {
        latir(); //Protected na classe
    }
    base
        latir();
        latir();
}
}
```

Arquivo: Gato.java

```
package bichos;

public class Gato extends Animal {
    public void fugirDe(Cachorro cachorro) {
        cachorro.latir(); //Latir está acessível
        System.out.println("Fugi do cachorro!");
    }
}
```

Exemplo 83. Modificador protected

5.9.1.4 Modificador private

O modificador `private` define que um membro de um tipo não pode ser acessado de nenhum ponto de fora do tipo ao qual pertence. Esse modificador não se aplica a tipos, apenas a membros de tipos.

Exemplo:

Arquivo: Cachorro.java

```
package bichos;

public class Cachorro extends Animal {
    private void latir () {                //Membro privado
        System.out.println("Au Au!");
    }
}
```

Arquivo: Cachorro.java

```
package bichos;

public class Poodle extends Cachorro {
    public void latirMuito() {
        latir();
        latir();
        latir();
    }
}
```

Exemplo 84. Modificador private

Como já foi dito na seção anterior, definindo-se um método como privado, na linguagem Java, o mesmo torna-se por consequência um método final, ou seja, não virtual.

5.9.2 Modificadores de Acesso da Linguagem C#

Os modificadores de acesso aplicáveis às declarações de tipos e membros de um tipo, na linguagem C#, são:

5.9.2.1 Modificador public

O modificador `public` define que o tipo ou membro ao qual se aplica não possui nenhum tipo de restrição de visibilidade.

Exemplo:

```
namespace Plantas {
    public class Planta {
        public void Florescer() {
            Console.WriteLine("Floresci!");
        }
    }
}

namespace Bichos {
    public class Animal {
        public Planta AlimentacaoVegetal; //Planta está acessível
        public void Andar() {
            Console.WriteLine("Andei!");
        }
    }
}
```

Exemplo 85. Modificador public

5.9.2.2 Modificador internal

O modificador `internal` define que o tipo ou membro pode ser acessado de qualquer ponto dentro do mesmo *namespace*.

Exemplo:

```
namespace Bichos {
    internal class Animal {
        internal void Andar() {
            Console.WriteLine("Andei!");
        }
    }

    internal class Cachorro : Animal { //Animal está acessível
        internal void Latir() {
            Console.WriteLine("Au Au!");
        }
    }
}
```

Exemplo 86. Modificador internal

O modificador `internal` é o modificador padrão em definições de tipos, ou seja, caso nenhum modificador seja explicitamente declarado, o mesmo é assumido pelo compilador.

O exemplo a seguir é equivalente ao anterior.

Exemplo:

```
namespace Bichos {
    class Animal {
        internal void Andar() {
            Console.WriteLine("Andei!");
        }
    }

    class Cachorro : Animal { //Animal está acessível
        internal void Latir() {
            Console.WriteLine("Au Au!");
        }
    }
}
```

Exemplo 87. Modificador internal

5.9.2.3 Modificador protected

O modificador `protected` define que um membro de um tipo pode ser acessado apenas de dentro do tipo em questão ou a partir de tipos derivados. Esse modificador não se aplica a tipos, apenas a membros de tipos.

Exemplo:

```
namespace Bichos {
    public class Cachorro : Animal {
        protected void Latir() {
            Console.WriteLine("Au Au!");
        }
    }

    public class Poodle : Cachorro {
        public void LatirMuito() {
            Latir();
            Latir();
            Latir();
        }
    }
}
```

Exemplo 88. Modificador protected

5.9.2.4 Modificador protected internal

A combinação dos modificadores `protected` e `internal` gera o modificador `protected internal`, equivalente ao modificador `protected` da linguagem Java, ou seja, define que

um membro de um tipo pode ser acessado de qualquer ponto de dentro do mesmo pacote ou a partir de tipos derivados do tipo em questão. Esse modificador não se aplica a tipos, apenas a membros de tipos.

Exemplo:

```
namespace Bichos {
    public class Cachorro : Animal {
        protected internal void Latir() {
            Console.WriteLine("Au Au!");
        }
    }

    public class Poodle : Cachorro {
        public void LatirMuito() {
            Latir(); //Protected internal na classe base
            Latir();
            Latir();
        }
    }

    public class Gato extends Animal {
        public void FugirDe(Cachorro cachorro) {
            cachorro.Latir(); //Latir está acessível
            Console.WriteLine("Fugi do cachorro!");
        }
    }
}
```

Exemplo 89. Modificador protected internal

5.9.2.5 Modificador private

O modificador `private` define que um membro de um tipo não pode ser acessado de nenhum ponto de fora do tipo ao qual pertence. Esse modificador não se aplica a tipos, apenas a membros de tipos.

Exemplo:

```
namespace Bichos {
    public class Cachorro : Animal {
        private void Latir() {
            Console.WriteLine("Au Au!");
        }
    }

    public class Poodle : Cachorro {
        public void LatirMuito() {
            Latir();
            Latir();
            Latir();
        }
    }
}
```

//ERRO DE COMPILAÇÃO

Exemplo 90. Modificador private

O modificador `private` é o modificador padrão em definições de membros de tipos, ou seja, caso nenhum modificador seja explicitamente declarado, o mesmo é assumido pelo compilador.

O exemplo a seguir é equivalente ao anterior:

Exemplo:

```
namespace Bichos {
    public class Cachorro : Animal {
        void Latir() {
            Console.WriteLine("Au Au!");
        }
    }

    public class Poodle : Cachorro {
        public void LatirMuito() {
            Latir();
            Latir();
            Latir();
        }
    }
}
```

//ERRO DE COMPILAÇÃO

Exemplo 91. Modificador private

5.9.2.6 Assemblies amigáveis

Além dos modificadores descritos acima, na linguagem C# é possível usar o atributo `System.Runtime.CompilerServices.InternalsVisibleTo` para especificar que determinados *assemblies* podem ter acesso às declarações internas do *assembly* em questão.

Exemplo:

Assembly Plantas.dll:

```
namespace Plantas {
    [InternalsVisibleTo("Bichos")]
    internal class Planta { ... }
    internal class Capim : Planta { ... }
}
```

Assembly Bichos.dll:

```
namespace Bichos {
    internal class Ruminante {
        //Acessa tipos internos ao assembly Plantas
        internal void Ruminar(Capim capim) { ... }
    }
}
```

Exemplo 92. Assemblies amigáveis

5.10 Inicialização dos Membros de um Tipo

No momento da criação de uma instância de um tipo, os valores de seus campos são inicializados conforme a sua definição. Essa inicialização difere levemente nas duas linguagens.

Exemplo (Java):

```
class Base {
    Base() {
        metodo();
    }
    void metodo() {
        return;
    }
}

class Derived extends Base {
    private int campo = 1;
    void metodo () {
        System.out.println("Derived.metodo() => campo = " + campo);
    }
}

class Program {
    public static void main(String[] args) throws IOException {
        Derived d = new Derived("d");
        d.metodo();
        System.in.read();
    }
}
```

Listagem impressa (Java):

```
(Derived)d.metodo() => campo = 0
(Derived)d.metodo() => campo = 1
```

Exemplo 93. Inicialização dos membros de um tipo

Exemplo (C#):

```
class Base {
    internal Base() {
        Metodo();
    }
    internal virtual void Metodo() {
        return;
    }
}

class Derived : Base {
    private int campo = 1;
    internal override void Metodo() {
        Console.WriteLine("Derived.Metodo() => campo = " + campo);
    }
}

class Program {
    public static void Main(string[] args) {
        Derived d = new Derived("d");
        d.Metodo();
        Console.ReadKey();
    }
}
```

Listagem impressa (C#):

```
(Derived)d.Metodo() => campo = 1
(Derived)d.Metodo() => campo = 1
```

Exemplo 94. Inicialização dos membros de um tipo

Como pode ser notado nos exemplos acima, na linguagem Java, ao executar o método `metodo()` a partir do construtor do tipo, o valor inicial do campo `campo` ainda não havia sido atribuído. Isso acontece porque, ao contrário do que ocorre na linguagem C#, na linguagem Java, o construtor da classe base é invocado antes da inicialização dos campos do tipo.

5.11 Escopo de Variáveis Locais

Variáveis locais são variáveis declaradas dentro de blocos de códigos, como métodos ou blocos `if` e `for`, ao contrário dos campos, que são variáveis declaradas diretamente na definição de um tipo.

Escopo pode ser definido como o espaço no qual uma variável está acessível. Membros de tipos como campos e métodos têm escopo definido conforme os modificadores de acesso utilizados em sua definição, já as variáveis locais tem como escopo o bloco de código no qual foram declaradas.

5.11.1 Escopo de Variáveis Locais na Linguagem Java

A linguagem Java permite que sejam definidas variáveis e parâmetros de métodos com o mesmo nome dado a campos definidos na declaração do tipo.

Exemplo:

```
class Test {
    double x; //Define um campo com o nome x
    double y; //Define um campo com o nome y
    void foo(boolean b, double y) { //Define um parâmetro com o nome y
        this.y = y; //Acessa o campo e o parâmetro y
        x = 1.0; //Acessa o campo x
        if (b) {
            int x = 1; //Define uma "variável local", de escopo
                //apenas dentro do bloco if,
                //também denominada x
            this.x = 2.0; //Acessa o "campo" x
        }
    }
}
```

Exemplo 95. Escopo de variáveis locais na linguagem Java

5.11.2 Escopo de Variáveis Locais na Linguagem C#

A linguagem C# só permite que sejam definidas variáveis locais com o mesmo nome dado a campos definidos na declaração do tipo caso o bloco de código exterior àquele no qual tais variáveis forem declaradas não acesse os campos de mesmo nome.

Exemplo:

```
class Test1 {
    double x; //Define um campo com o nome x
    double y; //Define um campo com o nome y
    void foo(boolean b, double y) { //Define um parâmetro com o nome y
        this.y = y; //Acessa o campo e o parâmetro y
        x = 1.0; //Acessa o campo x
        if (b) {
            int x = 1; //ERRO DE COMPILAÇÃO
        }
    }
}

class Test2 {
    double x; //Define um campo com o nome x
    double y; //Define um campo com o nome y
    void foo(boolean b, double y) { //Define um parâmetro com o nome y
        this.y = y; //Acessa o campo e o parâmetro y
        if (b) {
            x = 1.0;
        } else {
            int x = 1; //Não há acesso ao campo x
        } //no bloco exterior
    }
}
```

Exemplo 96. Escopo de variáveis locais na linguagem C#

5.12 Tipos Aninhados

Tipos aninhados são tipos definidos dentro de outros tipos.

Exemplo:

```
class Test {
    /*Membros de Test*/
    public class Nested {
        /*Membros de Nested*/
    }
}
Test.Nested n = new Test.Nested();
```

Exemplo 97. Tipos aninhados

Alguns aspectos interessantes podem ser observados num comparativo dos tipos estáticos das duas linguagens. Tais aspectos serão apresentados nas subseções seguintes.

5.12.1 Tipos Aninhados na Linguagem Java

Como a linguagem Java não permite a definição de structs, os tipos definidos pelo desenvolvedor na linguagem Java são sempre classes. Sendo assim, os tipos aninhados na linguagem Java são mais conhecidos como Classes Aninhadas.

Tais classes se dividem em dois tipos:

5.12.1.1 Classes aninhadas estáticas

Classes aninhadas estáticas (*static nested classes*, em inglês) é o termo usado para se referir àquelas classes aninhadas que não possuem permissão para acessar os membros não estáticos da classe na qual estão definidas, por isso o uso do termo estático para identificar essas classes.

Exemplo:

```
class Test {
    int x = 0;
    static class Nested {
        Nested() {
            this.x = 0;           //ERRO DE COMPILAÇÃO
        }
    }
}
```

Exemplo 98. Classes aninhadas estáticas

5.12.1.2 Classes aninhadas não estáticas

Classes aninhadas não estáticas (*Inner Classes* em inglês) são classes aninhadas que possuem permissão para acessar os membros não estáticos da classe na qual estão definidas.

Exemplo:

```
class Test {
    int x = 0;
    class Nested {
        Nested() {
            this.x = 0;           //OK
        }
    }
}
```

Exemplo 99. Classes aninhadas não estáticas

Como consequência, algumas restrições se aplicam às classes internas:

- Uma classe interna não pode conter nenhum membro estático.
- Uma instância de uma classe interna só pode existir dentro de uma instância da classe que a contém, ao contrário de uma classe aninhada estática, que pode ser instanciada de fora da classe que a contém.

Exemplo:

```
class Test1 {
    class Nested1 {                //Classe interna
    }
}

class Test2 {
    static class Nested2 {        //Classe aninhada estática
    }
}

Test1.Nested1 n1 = new Test1.Nested1(); //ERRO DE COMPILAÇÃO

Test1 test1 = new Test1();
Test1.Nested1 n2 = test1.new Nested1(); //OK

Test2.Nested2 n3 = new Test2.Nested2(); //OK
```

Exemplo 100. Classes aninhadas não estáticas

Além das classes internas definidas como membros de uma outra classe, há mais dois tipos de classes internas:

5.12.1.2.1 Classes internas locais

Classes internas locais (*Local inner classes* em inglês) são classes internas definidas dentro do corpo de um método, e cujo escopo é o mesmo de uma variável local.

Exemplo:

```
class Test {
    int foo() {
        class LocalInner {                //Classe interna local
            /*Membros de InnerLocal*/
        }
        LocalInner li = new LocalInner();
        li.DoSomething();
    }
}
```

Exemplo 101. Classes internas locais

5.12.1.2.2 Classes internas anônimas

Classes internas anônimas (*Anonymous Inner Classes* em inglês) são classes internas locais definidas sem um nome.

Esse tipo de classe interna será trabalhado melhor na seção “Tipos Anônimos”.

5.12.2 Tipos Aninhados na Linguagem C#

Ao contrário do que ocorre na linguagem Java, há apenas um tipo de classe aninhada na linguagem C#, que seria equivalente às classes aninhadas estáticas da linguagem Java, ou seja, não possuem acesso aos membros não estáticos da classe que a contém, porém não se utiliza o modificador `static` para a definição de tais classes.

Exemplo:

```
class Test {
    int x = 0;
    class Nested {
        Nested() {
            this.x = 0;           //ERRO DE COMPILAÇÃO
        }
    }
}
```

Exemplo 102. Tipos aninhados na linguagem C#

Não há nenhum modo de definir tipos com comportamento semelhante ao das classes aninhadas não estáticas da linguagem Java na linguagem C#.

Quando uma instância de um tipo é utilizada a partir de um tipo aninhado, todos os membros do tipo principal se tornam acessíveis, sendo assim, uma alternativa para os casos em que um tipo aninhado precisa ter acesso aos membros não estáticos do tipo que o contém é passar a instância atual desse tipo, usando o comando `this`, como parâmetro no construtor do tipo aninhado, e a partir de então utiliza-la quando necessário.

Exemplo:

```
class Test {
    public static void Main() {
        Test a = new Test();
        a.nested.Foo();           //OK
        Console.WriteLine(a.nested.x); //ERRO, x é privado
    }
    private int x = 0;           //Campo privado
    public Nested nested;
    private Test() {
        nested = new Nested(this); //Passa this para o nested
    }
    public class Nested {
        private Test parent;
        public Nested(Test parent) {
            this.parent = parent;
        }
        public void Foo() {
            Console.WriteLine(parent.x); //OK, mesmo x sendo privado
        }
    }
}
```

Exemplo 103. Tipos aninhados na linguagem C#

Uma observação importante deve ser feita a respeito do uso do modificador `static` na definição de classes aninhadas: Uma vez que esse modificador tem significados diferentes nas duas linguagens quando utilizado na definição de classes, classes aninhadas definidas de maneira idêntica podem produzir efeitos completamente distintos.

Exemplo:

Em Java:

```
class Test {
    public static class Nested {           //Define uma classe aninhada estática
    }
}
Test.Nested t = new Test.Nested();      //OK - Instancia um novo Test.Nested
```

Em C#:

```
class Test {
    public static class Nested {           //Define uma classe estática aninhada
    }
}
Test.Nested t = new Test.Nested();      //ERRO DE COMPILAÇÃO - Tipos
                                         //estáticos não podem ser instanciados
```

Exemplo 104. Tipos aninhados na linguagem C#

Além de classes aninhadas, também podem ser declarados *structs* aninhados, em outros *structs*, ou em classes.

Exemplo:

```
class TestClass {
    class NestedClass {
    }
    struct NestedStruct {
    }
}

struct TestStruct {
    class NestedClass {
    }
    struct NestedStruct {
    }
}
```

Exemplo 105. Tipos aninhados na linguagem C#

Assim como os demais membros de um tipo, caso nenhum modificador de acesso seja usado para o tipo aninhado, a visibilidade considerada será a equivalente ao modificador `private`.

5.13 Tipos Anônimos

Tipos anônimos estão presentes em ambas as linguagens, porém com diferentes propósitos.

5.13.1 Tipos Anônimos na Linguagem Java:

Tipos anônimos, na linguagem Java, são classes aninhadas que apresentam as seguintes características:

- Não possuem um nome;
- São fortemente tipados, sendo que um tipo é criado implicitamente pelo compilador para representá-lo;
- São classes aninhadas não estáticas (*Inner Classes*);
- Devem implementar uma interface ou estender a um tipo conhecido;
- Podem conter campos e métodos ou tipos aninhados como membros;
- Não podem conter campos estáticos;
- São utilizadas para implementar padrões de projeto como o *Observer* (GAMMA, 1994).

O exemplo a seguir ilustra a definição e o uso de um tipo anônimo na linguagem Java.

Exemplo:

```
class Programa {
    Programa() {
        Button btn = new Button("Antes");
        btn.addActionListener(
            new ButtonActionAdapter() {
                void onClick() {
                    btn.setText("Depois!");
                }
            }
        )
    }
}
```

Exemplo 106. Tipos anônimos na linguagem Java

No exemplo acima, um tipo anônimo é definido e uma instância desse tipo é criada e passada como parâmetro para o método `addActionListener()` da classe `Button`. Esse tipo anônimo estende uma classe denominada `ButtonActionAdapter` e sobrescreve o método `onClick()` dessa classe.

5.13.2 Tipos Anônimos na Linguagem C#:

Tipos anônimos, na linguagem C#, são classes aninhadas que apresentam as seguintes características:

- Não possuem um nome;
- São fortemente tipados, sendo que um tipo é criado implicitamente pelo compilador para representá-lo;
- Conseguem acessar variáveis e métodos visíveis no método dentro do qual foram definidos, de modo semelhante aos tipos aninhados não estáticos da linguagem Java.
- Não podem implementar interfaces ou estender um outro tipo. Por padrão todos estendem o tipo `System.Object`;

- Não podem conter métodos ou tipos aninhados como membros. Todos os seus membros devem ser campos;
- Não podem conter campos estáticos;
- Todos os campos devem ser inicializados;
- Todos os campos são definidos como somente leitura;
- Os tipos dos campos não devem ser informados. São inferidos pelo compilador;
- As definições dos campos devem ser separadas por vírgulas;
- São *Reference Types*, ou seja, são alocados por referência;
- Tem o operador `==` e o método `Equals()` sobrescritos de modo a se comportarem como *Value Types* nas operações de comparação.

Exemplo:

```
class Programa {
    int a = 5;
    Programa() {
        int b = 2;
        var p = new {
            x = a,
            y = b
        };
        Console.WriteLine("p.x = {0}, p.y = {1}", p.x, p.y);
    }
}
```

Exemplo 107. Tipos anônimos na linguagem C#

Note que, como o tipo é anônimo e não pode implementar interfaces ou estender outros tipos, para acessar seus campos, suas instâncias devem ser armazenadas em variáveis definidas por inferência de tipos, utilizando-se o comando `var` no lugar do tipo da variável. Inferência de Tipos é um recurso que será apresentado no capítulo seguinte.

Note que o exemplo a seguir não define nenhum tipo anônimo.

Exemplo:

```
class MyClass {
    internal int x;
    internal int y;
}

class Programa {
    int a = 5;
    Programa() {
        int b = 2;
        var p = new MyClass {
            x = a,
            y = b
        };
        Console.WriteLine("p.x = {0}, p.y = {1}", p.x, p.y);
    }
}
```

Exemplo 108. Tipos anônimos na linguagem C#

Nesse caso, uma nova instância do tipo `MyClass` será criada e inicializada, conforme os parâmetros passados. Inicialização de objetos é um recurso que será apresentado no capítulo seguinte.

5.14 Identificação e Conversão de Tipos

Em ambas as linguagens, existem basicamente duas maneiras de identificar um tipo durante a execução do programa: Através de comandos de linguagem específicos para esse fim, ou através de programação reflexiva. Programação reflexiva é um assunto apresentado posteriormente neste capítulo.

Já a conversão de um tipo em outro, de formato compatível, pode ser feita de três maneiras: Através de uma conversão implícita, através de uma conversão explícita ou através de métodos de conversão.

Esta seção apresentará as opções disponíveis para identificação e conversão de tipos nas linguagens Java e C#.

5.14.1 Identificação de Tipos na Linguagem Java

A linguagem Java oferece o operador `instanceof`, que permite saber se uma dada instância pertence a um dado tipo.

Exemplo:

```
Object p = new Point();
boolean b = p instanceof Point;
System.out.println(b); //Imprime "true"
```

Exemplo 109. Identificação de tipos na linguagem Java

As seguintes características do operador `instanceof` devem ser observadas:

- Retorna verdadeiro sempre que a instância testada puder ser referenciada por uma variável do tipo testado, ou seja, o tipo da instância testada e o tipo testado não precisam ser o mesmo, basta que sejam compatíveis;
- Não é aplicável a tipos primitivos;

Através do uso de programação reflexiva (*reflection*) é possível simular o operador `instanceof` obtendo a classe a ser testada e utilizando o método `isInstance()` da mesma.

Exemplo:

```
Class type = Class.forName("Point");
Object p = new Point();
boolean b = type.isInstance(p);
System.out.println(b); //Imprime "true"
```

Exemplo 110. Identificação de tipos na linguagem Java

5.14.2 Identificação de Tipos na Linguagem C#

A linguagem C# disponibiliza o operador `is`, que desempenha função semelhante ao operador `instanceof` da linguagem Java.

Exemplo:

```
object p = new Point();
bool b = p is Point;
Console.WriteLine(b); //Imprime "true"
```

Exemplo 111. Identificação de tipos na linguagem C#

As seguintes características do operador `is` devem ser observadas:

- Retorna verdadeiro sempre que a instância testada puder ser referenciada por uma variável do tipo testado, ou seja, o tipo da instância testada e o tipo testado não precisam ser o mesmo, basta que sejam compatíveis;
- É aplicável tanto a *Reference Types* quanto a *Value Types*.

Utilizando programação reflexiva, o operador `is` pode ser simulado, assim como o operador `instanceof`, na linguagem Java.

Exemplo:

```
Type type = typeof(Point);
object p = new Point();
bool b = type.IsInstanceOfType(p);
System.out.println(b); //Imprime "true"
```

Exemplo 112. Identificação de tipos na linguagem C#

5.14.3 Conversão de Tipos na Linguagem Java

Conversão de tipos é um assunto que deve ser estudado com muito cuidado em ambas as linguagens, porém para efeitos de comparação com a linguagem C#, o principal ponto a ser destacado na linguagem Java é a ausência de um operador de conversão segura.

Para realizar uma conversão explícita, na linguagem Java, deve-se informar o tipo de destino entre parênteses, precedendo a instância que se deseja converter.

Exemplo:

```
Object o = new Point();
Point p = (Point)o; //Converte o de Object para Point
```

Exemplo 113. Conversão de tipos na linguagem Java

Essa conversão é considerada não segura porque caso a instância convertida não seja de um tipo compatível com o tipo utilizado na conversão, uma exceção será lançada.

Além do método de conversão acima, é possível utilizar métodos específicos para conversão de alguns tipos. Os mais comuns são os métodos `parse()` utilizados para converter *strings* em representações numéricas.

5.14.4 Conversão de Tipos na Linguagem C#

Além da conversão explícita não segura, semelhante àquela disponível na linguagem Java, a linguagem C# também disponibiliza o operador `as`, que permite uma conversão segura, ou seja, que não lançará uma exceção caso falhe. Nesse caso a conversão retornará uma referência nula.

Exemplo:

```
object o = new Point();
Point p = (Point)o; //Converte o de Object para Point
Rectangle r = (Rectangle)o; //Erro de execução, o não é um Rectangle
Triangle t = o as Triangle //Retorna null, pois o não é um Triangle
```

Exemplo 114. Conversão de tipos na linguagem C#

Em ambas as linguagens, tanto as conversões explícitas quanto as conversões implícitas são verificadas pelo compilador para evitar conversões que sempre falham. Caso o tipo utilizado na conversão não seja compatível com o tipo da instância utilizada na conversão, um erro de compilação será gerado.

Exemplo:

```
Type o = new Type();
Point p = (Point)o;           //ERRO DE COMPILAÇÃO, Type não é compatível com Point
```

Exemplo 115. Conversão de tipos na linguagem C#

Assim como acontece na linguagem Java, algumas conversões podem ser realizadas utilizando métodos específicos dos tipos em questão. Porém, além disso, a linguagem C# conta com o tipo estático `System.Convert`, que fornece métodos para conversões entre os principais tipos nativos da linguagem, além de um método `ChangeType()`, que permite uma conversão forçada de um tipo para outro, caso tais tipos sejam compatíveis.

Conversões implícitas são semelhantes em ambas as linguagens.

Conversões numéricas serão estudadas na seção “Operações Numéricas”, posteriormente neste capítulo.

5.15 Comandos de Iteração

Ambas as linguagens disponibilizam comandos de iteração semelhantes:

- **While:** O comando `while` permite a repetição da execução de um bloco de código enquanto uma dada condição for verdadeira, sendo que tal condição será testada antes de cada execução do bloco.

Exemplo (C#):

```
int i = 1;
while (i < 10) {
    i++;
}
Console.WriteLine(i);           //Imprime 10
```

Exemplo 116. Comandos de iteração

- Do While: O comando `do while` permite a repetição da execução de um bloco de código enquanto uma dada condição for verdadeira, sendo que tal condição será testada após cada execução do bloco.

Exemplo (C#):

```
int i = 1;
do {
    i++;
}
while (i < 10);
Console.WriteLine(i);           //Imprime 10
```

Exemplo 117. Comandos de iteração

- For: O comando `for` permite a repetição da execução de um bloco de código enquanto uma dada condição for verdadeira, porém além dessa condição, esse comando possui um bloco de inicialização, executado antes da primeira execução e um bloco de iteração, executado após cada execução.

Exemplo:

```
int j = 1;
for (int i = 1; i < 10; i++) {
    j++;
}
Console.WriteLine(j);           //Imprime 10
```

Exemplo 118. Comandos de iteração

Nesse comando, podem ser notadas algumas diferenças nas duas linguagens. Enquanto a linguagem Java permite que o bloco de inicialização, a condição de parada ou o bloco de iteração sejam omitidos, a linguagem C# só admite o uso desse comando caso as três condições estejam presentes.

Exemplo:

Java:

```
for (;;) //Execução infinita, pois não há condição de parada
```

C#:

```
for (;;) //ERRO DE COMPILAÇÃO
```

Exemplo 119. Comandos de iteração

- For Each: O comando `foreach`, na linguagem C#, permite a execução de um bloco de código para cada um dos elementos de uma dada coleção. Na linguagem Java esse comando recebe o nome de `for`, porém com uma estrutura diferente do bloco `for` comum.

Exemplo:

Java:

```
int[] inteiros = {1, 2, 3, 4, 5};
for (int i : inteiros) {
    System.out.println(i);
}
```

C#:

```
int[] inteiros = {1, 2, 3, 4, 5};
foreach (int i in inteiros) {
    Console.WriteLine(i);
}
```

Exemplo 120. Comandos de iteração

Enquanto na linguagem Java a coleção passada deve ser uma instância da interface `java.lang.Iterable`, ou um arranjo, na linguagem C# tal coleção deve ser uma instância de qualquer tipo que possua um método, público e não estático, denominado `GetEnumerator()`, que retorne um objeto que, por sua vez, possua uma propriedade,

pública e não estática, denominada `Current`, que retorne o item corrente da coleção e um método, público e não estático, denominado `MoveNext()`, que atribua o próximo item da coleção à propriedade `Current` e retorne `true`, ou retorne `false`, caso não haja mais itens na coleção.

Existem interfaces utilizadas para esse fim, porém as mesmas não são obrigatórias. Tais interfaces serão apresentadas na seção "Iteradores", posteriormente neste capítulo.

Devido a essa flexibilidade, construções como a seguinte são permitidas:

```
class MyIterator {
    int[] ints = {1, 2, 3, 4, 5};
    int counter = 0;
    public bool MoveNext() {
        return counter++ < ints.Length;
    }
    public int Current {
        get { return ints[counter-1]; }
    }
}

class MyEnumerable {
    public MyIterator GetEnumerator() {
        return new MyIterator();
    }
}

MyEnumerable inteiros = new MyEnumerable();
foreach (int i in inteiros) {
    Console.WriteLine(i);
}

//Imprime:
//1
//2
//3
//4
//5
```

Exemplo 121. Comandos de iteração

Uma outra observação interessante, a respeito dos comandos de iteração, é o fato de a linguagem Java permitir o uso de *labels*, em conjunto com o comando `continue`, para desviar o fluxo de execução entre comandos `for` aninhados, embora o mesmo

comportamento possa ser obtido, na linguagem C#, através do uso de *labels* em conjunto com o comando `goto`.

Exemplo:

Java:

```
outerLoop: for(String a : foo) {
    for (String b : bar) {
        if (b.equals("buzz")) {
            continue outerLoop;
        }
    }
    // other code comes here...
}
```

C#:

```
foreach (string a in foo) {
    foreach (string b in bar) {
        if (b == "buzz") {
            goto end_of_inner_loop;
        }
    }
    end_of_inner_loop: {}
    // other code comes here...
}
```

Exemplo 122. Comandos de iteração

5.16 O Comando Switch

À primeira vista o comando `switch` de ambas as linguagens é bastante similar, porém uma breve análise já revela significativas diferenças:

- A linguagem C# permite que variáveis do tipo `string` sejam testadas, ao contrário da linguagem Java, que só admite variáveis de tipos numéricos ou *enums*.

Exemplo (C#):

```
string s = "A";
switch (s) {
    case "A": Console.WriteLine("A"); break;
    case "B": Console.WriteLine("B"); break;
    default: Console.WriteLine(s); break;
}
```

Exemplo 123. O comando switch

- A linguagem C# só permite que o fluxo de execução passe para o bloco case seguinte caso o bloco case em execução esteja vazio. Caso contrário, o comando break ou o comando goto case devem ser usados explicitamente.

Exemplo (C#):

```
string s = "A";
switch (s) {
    case "A": s = "a"; break;           //OK, interrompe o fluxo
    case "B":                               //OK, executa o mesmo bloco para B e C
    case "C": goto case "A";           //OK, passa o fluxo para o case "A"
    case "D": s = "d";                 //ERRO DE COMPILAÇÃO
    case "E": s = "e"; break;
}
```

Exemplo 124. O comando switch

A linguagem Java, assim como as linguagens C e C++, permite que o fluxo de execução passe para o bloco case seguinte, sem testar novamente a condição.

Exemplo (Java):

```
int i = 1;
switch (i) {
    case 0: System.out.println(0);
    case 1: System.out.println(1);
    case 2: System.out.println(2);
    case 3: System.out.println(3); break;
    case 4: System.out.println(4);
}

//Imprime:
//1
//2
//3
```

Exemplo 125. O comando switch

5.17 Operações Numéricas

Embora os computadores possam realizar operações numéricas em um curtíssimo tempo, o que foi fundamental para sua evolução, a precisão dos seus resultados ainda deixa a desejar, e por isso deve ser tratada com atenção.

Essa seção destacará algumas características das linguagens Java e C# ao realizarem operações numéricas.

5.17.1 Os Operadores Checked e Unchecked

Em ambas as linguagens, operações de conversão de valores entre tipos inteiros, onde a conversão é feita para um tipo de menor representatividade, ou seja, que admita um intervalo de valores que não comporta o valor a ser convertido, o valor é truncado.

A linguagem C# fornece os operadores `checked` e `unchecked` que permitem controlar quando esse comportamento será admitido. Caso o operador `checked` seja usado, tais situações não serão admitidas e resultarão em uma exceção. Caso o operador `unchecked` seja usado esse comportamento será admitido.

Exemplo (C#):

```
byte x = 255;
byte y = 255;
byte a = (byte)(x + y);           //255 + 255 = 254
a = checked ((byte)(x + y));     //OVERFLOW EXCEPTION
a = (byte)(255 + 255);           //ERRO DE COMPILAÇÃO
a = unchecked ((byte)(255 + 255)); //255 + 255 = 254
unchecked {
    a = (byte)(x + y);           //255 + 255 = 254
    a = (byte)(255 + 255);     //255 + 255 = 254
}
checked {
    a = (byte)(x + y);           //OVERFLOW EXCEPTION
    a = (byte)(255 + 255);     //ERRO DE COMPILAÇÃO
}
int b = x + y;                   //255 + 255 = 510
```

Exemplo 126. Os comandos `checked` e `unchecked`

5.17.2 O Modificador `strictfp`

Em operações numéricas do tipo, $x = y * z / w$, sendo x , y , z e w variáveis dos tipos `float` ou `double`, caso o resultado da expressão $y * z$ seja um número maior que o limite superior do tipo, o resultado dessa expressão será truncado, o que resultará em um resultado impreciso.

Alguns processadores resolvem esse problema utilizando registradores especiais de tamanho maior que os 32 *bits*, usados para o tipo `float`, e que os 64 *bits*, usados para o tipo `double`, para armazenar cálculos temporários, o que permite um resultado mais preciso.

Como nem todos os processadores oferecem essa característica, o mesmo cálculo poderia resultar em diferentes resultados, dependendo do computador utilizado para executá-lo. Para evitar essa discrepância de resultados, as versões iniciais da linguagem Java truncavam os valores dos cálculos temporários mesmo quando o processador utilizado oferecia a capacidade de realizar cálculos mais precisos. Tal comportamento gerou um descontentamento na comunidade de desenvolvedores Java, o que fez com que, a partir da versão 1.2 da linguagem, o comportamento padrão fosse alterado para sempre utilizar as otimizações dos processadores, mesmo ocasionando discrepâncias de resultados entre computadores diferentes.

Como alternativa, para aqueles que desejassem o truncamento dos valores intermediários das operações, foi criado o modificador `strictfp`.

Esse modificador pode ser utilizado em classes e métodos, sendo que quando usado em classes, torna todas as operações de ponto flutuante, realizadas nos métodos e construtores dessa classe, restritas a ponto flutuante, ou seja, operações que não utilizam as otimizações do processador utilizado.

Não há nenhum modo de forçar o truncamento das operações intermediárias na linguagem C#. O resultado das operações de ponto flutuante sempre retornam o resultado mais preciso possível.

5.17.3 O Tipo `System.Decimal`

A linguagem C# fornece o tipo `decimal` para operações onde a precisão deve ser grande, mas a escala de valores não precisa ser tanto, como em operações financeiras, por exemplo.

Enquanto a escala de valores de um `double` varia de 5.0×10^{-324} a 1.7×10^{208} com precisão de 15 a 16 dígitos, a escala de valores do tipo `decimal` varia de 1×10^{-28} a 1×10^{28} com precisão de pelo menos 28 dígitos. Além disso, números fracionários decimais como 0.1 são representados com exatidão, ao contrário do que ocorre com os tipos `float` e `double`.

5.18 Tratamento de Eventos

Tratamento de eventos é um recurso essencial em linguagens utilizadas para o desenvolvimento de interfaces gráficas de interação com o usuário do software. Ferramentas de desenvolvimento como o Delphi obtiveram grande sucesso por terem sabido tirar proveito desse recurso.

Em ambas as linguagens deste estudo o tratamento de eventos está presente, porém de maneiras bem distintas. Enquanto na linguagem C# os eventos fazem parte da própria linguagem, na linguagem Java utiliza-se o padrão de projeto *Observer* para suprir tal necessidade.

5.18.1 Tratamento de Eventos na Linguagem Java

A definição de um evento na linguagem Java não depende de nenhuma estrutura particular da linguagem, mas sim do uso de determinadas convenções para a implementação de tal funcionalidade.

Para definir um evento deve-se:

- Definir uma interface que declare os métodos destinados a tratar cada um dos eventos disponibilizados pelo tipo;
- Criar uma classe que forneça uma implementação vazia da interface descrita anteriormente, para que possa ser sobrescrita de modo a fornecer o comportamento desejado a seus métodos;
- Criar um campo e um método *setter* para associar uma instância de tal interface à instância do tipo que expõe os eventos;
- Implementar chamadas, no momento em que o evento deva ser disparado, para o método equivalente na instância da interface, caso haja uma.

Exemplo:

```
interface ButtonListener {
    void onClick();
}

class ButtonAdapter {
    void onClick() { }
}

class Button {
    ButtonListener listener = null;
    void setListener(ButtonListener listener) {
        this.listener = listener;
    }
    void doClick() {
        if (listener != null) {
            listener.onClick();
        }
    }
    /* ... */
}
```

Exemplo 127. Tratamento de eventos na linguagem Java

Para utilizar um evento, deve-se:

- Criar uma instância de um tipo anônimo (*Anonymous Inner Class*) que estenda a classe vazia criada para implementar a interface descritora dos eventos e sobrescrever os métodos desejados com o código que deseja executar no momento de cada evento;
- Utilizar o método criado no tipo para associar a instância criada anteriormente ao campo do tipo da interface. Desse modo, os métodos desta instância serão invocados quando o evento ocorrer.

Exemplo:

```
Button btn = new Button();
btn.setListener(
    new ButtonAdapter() {
        void onClick() {
            btn.setText("Clicado!");
        }
    }
);
```

Exemplo 128. Tratamento de eventos na linguagem Java

5.18.2 Tratamento de Eventos na Linguagem C#

Eventos são partes integrantes da linguagem C#, sendo que a principal diferença com relação aos eventos da linguagem Java é a maior praticidade no seu uso e definição.

Para definir um evento deve-se:

- Criar o evento no tipo;
- Implementar chamadas para o evento, no momento em que o mesmo deva ser disparado, passando os devidos parâmetros;
- Inicializar o evento com um valor vazio, no construtor do tipo, ou implementar checagem se o evento está nulo antes de fazer a chamada ao mesmo.

Exemplo:

```
class Button {
    public Button() {
        OnClick += delegate { };
    }
    public event EventHandler OnClick();
    void DoClick() {
        OnClick(this, new EventArgs());
    }
    /* ... */
}
```

Exemplo 129. Tratamento de eventos na linguagem C#

Para utilizar um evento deve-se:

- Atribuir um método ao evento.

Exemplo:

```
Button btn = new Button();
btn.OnClick = (o, e) => btn.Text = "Clicado!";
```

Exemplo 130. Tratamento de eventos na linguagem C#

Os exemplos acima utilizam as convenções e recursos mais recentes da linguagem, como por exemplo a expressão lambda utilizada como método atribuído ao evento.

Além desses novos recursos, os eventos sofreram significativas evoluções desde a versão 1.0 da linguagem, o que tornou a sua utilização muito mais prática.

Maiores detalhes sobre a definição de eventos na linguagem C# serão apresentados na seção "Delegates, Métodos Anônimos e Expressões Lambda", no capítulo seguinte.

5.19 Tratamento de Exceções

Ambas as linguagens disponibilizam um modelo de tratamento de exceções muito semelhante, com blocos `try`, `catch` e `finally`, identificação da exceção e relançamento quando conveniente.

Exemplo:

```
try {  
    //...  
} catch (MyException me) {  
    throw new MyOtherException(me);  
} finally {  
    //...  
}
```

Exemplo 131. Tratamento de exceções

A principal diferença está no fato de não existirem exceções checadas na linguagem C#, ou seja, não é possível declarar quais são as exceções lançadas por um determinado método, sendo que a documentação da biblioteca é o único recurso para que o usuário do mesmo possa conhecer tais exceções.

A linguagem Java permite e requer tal declaração sempre que o método permitir que alguma exceção seja propagada para fora do mesmo. Essa declaração é realizada utilizando-se o comando `throws`.

Exemplo (Java):

```
void myMethod() throws MyException { //OK
    if ( ... ) throw new MyException();
}

void myMethod { //OK
    try {
        if ( ... ) throw new MyException();
    } catch (MyException me) {
        //Trata exceção
    }
}

void myMethod() { //ERRO DE COMPILAÇÃO
    if ( ... ) throw new MyException();
}
```

Exemplo 132. Tratamento de exceções

As vantagens e desvantagens da existência ou não de exceções “checadas” é um tópico que gera acaloradas discussões nos fóruns e blogs especializados, na internet. No momento da definição da linguagem C#, optou-se pela não inclusão dessa característica.

Uma outra diferença está no fato de a linguagem C# permitir que sejam declarados blocos `try catch` sem que se declare uma variável para armazenar a exceção capturada. Nesses casos, é possível especificar que tipo de exceção deseja-se capturar ou simplesmente capturar todo tipo de exceção.

Exemplo (C#):

```
try {
    //...
} catch (MyException me) {           //Armazena excessão capturada
    throw new MyOtherException(me);
} catch (Exception) {               //Apenas especifica excessão a capturar
    throw;
} catch {                             //Captura qualquer excessão
    //...
}
```

Exemplo 133. Tratamento de exceções

5.20 Tipos Genéricos

Tipos genéricos são um grande avanço nas linguagens de programação modernas, aclamado pela maioria dos desenvolvedores que as utilizam. Trata-se de tipos adaptáveis, que podem ter um ou mais tipos agregados, normalmente denominados parâmetros genéricos, definidos no momento de sua utilização, ao invés de o serem no momento de sua definição.

O uso mais comum para tipos genéricos é a definição de coleções cujo tipo de dados de seus itens é um parâmetro genérico, permitindo que a mesma implementação da coleção possa ser usada para diferentes tipos de itens.

Exemplo (C#):

```
public class List<T> {                //Define lista genérica

    public void Add(T item) { ... }   //Adiciona ítem genérico à
lista
    public void RemoveAt(int index) { ... } //Remove ítem da lista
    public T Retrieve(int index) { ... }  //Obtém ítem da lista
    public int IndexOf(T item) { ... }   //Obtém o índice de um ítem
}
List<int> inteiros = new List<int>();    //Utiliza como lista de int
List<string> strings = new List<string>(); //Utiliza como lista de string
List<MyClass> myobjects = new List<MyClass>(); //Utiliza como lista de MyClass
```

Exemplo 134. Tipos genéricos

Os tipos genéricos foram introduzidos na linguagem C# em sua versão 2.0, sendo que pouco tempo depois também foram disponibilizados para a linguagem Java, com o lançamento de sua versão 1.5.

Apesar de ter o mesmo objetivo, esse recurso foi implementado de maneira muito diferente nas duas linguagens.

5.20.1 Tipos Genéricos na Linguagem Java

Normalmente, ao definir um tipo genérico na linguagem Java, o que ocorre por traz dos panos é a definição de um tipo não genérico cujos parâmetros genéricos são todos do tipo `Object`.

Exemplo:

O tipo descrito abaixo:

```
public class List<T> {
    public void add(T item) { ... }
    public void removeAt(int index) { ... }
    public T retrieve(int index) { ... }
    public int indexOf(T item) { ... }
}
```

é compilado como:

```
public class List {
    public void add(Object item) { ... }
    public void removeAt(int index) { ... }
    public Object retrieve(int index) { ... }
    public int indexOf(Object item) { ... }
}
```

Exemplo 135. Tipos genéricos na linguagem Java

Desse modo, para obter o comportamento de tipo genérico, o compilador insere conversões do tipo `Object` para o tipo utilizado como parâmetro genérico na declaração da variável em todos os pontos onde for necessário.

Exemplo:

O trecho de código abaixo:

```
List<Integer> inteiros = new List<Integer>();  
Integer i = inteiros.retrieve(0);
```

é compilado como:

```
List inteiros = new List();  
Integer i = (Integer)inteiros.retrieve(0);
```

Exemplo 136. Tipos genéricos na linguagem Java

A linguagem Java permite que algumas restrições sejam aplicadas aos parâmetros dos tipos genéricos. Tais restrições podem modificar o comportamento acima e permitir um menor grau de abstração dos tipos dos parâmetros genéricos.

Exemplo:

O tipo descrito abaixo:

```
public class MyList<T extends MyClass> {           //Indica que T deve estender MyClass
    public void add(T item) { ... }
    public void removeAt(int index) { ... }
    public T retrieve(int index) { ... }
    public int indexOf(T item) { ... }
}
```

é redefinido como:

```
public class MyList {
    public void add(MyClass item) { ... }
    public void removeAt(int index) { ... }
    public MyClass retrieve(int index) { ... }
    public int indexOf(MyClass item) { ... }
}
```

e o seguinte trecho de código:

```
List<MySubClass> myobjects = new List<MySubClass>();
MySubClass i = myobjects.retrieve(0);
MyClass j = myobjects.retrieve(0);
```

passa a ser compilado como:

```
List myobjects = new List();
MySubClass i = (MySubClass)myobjects.retrieve(0);
MyClass j = myobjects.retrieve(0);
```

Exemplo 137. Tipos genéricos na linguagem Java

Alguns efeitos colaterais desta forma de definir tipos genéricos são:

- Não é possível determinar, em tempo de execução, quais são os tipos dos parâmetros genéricos de um tipo;
- É possível declarar variáveis de tipos genéricos sem informar os parâmetros genéricos.

Exemplo:

```
public class List<T> {
    public void add(T item) { ... }
    public void removeAt(int index) { ... }
    public T retrieve(int index) { ... }
    public int indexOf(T item) { ... }
}
List<Integer> lista1 = new List<Inteiros>(); //Parâmetro genérico definido
List lista2 = new List(); //Parâmetro genérico indefinido
```

Exemplo 138. Tipos genéricos na linguagem Java

5.20.2 Tipos Genéricos na Linguagem C#

Na linguagem C#, os tipos genéricos são tipos especiais que são usados como gabaritos para tipos especializados, criados no momento em que o tipo genérico é utilizado.

Quando uma instância de um tipo genérico é criada, um novo tipo, não genérico, é criado, sendo que, nesse novo tipo, todas as referências aos parâmetros genéricos são substituídas por referências aos tipos especificados.

Exemplo:

O tipo descrito abaixo:

```
public class List<T> {
    public void Add(T item) { ... }
    public void RemoveAt(int index) { ... }
    public T Retrieve(int index) { ... }
    public int IndexOf(T item) { ... }
}
```

quando instanciado:

```
List<int> inteiros = new List<int>();
```

gera um novo tipo especializado:

```
public class ListInt {
    public void Add(int item) { ... }
    public void RemoveAt(int index) { ... }
    public int Retrieve(int index) { ... }
    public int IndexOf(int item) { ... }
}
```

e a instanciação da variável é substituída por:

```
ListInt inteiros = new ListInt();
```

Exemplo 139. Tipos genéricos na linguagem C#

Sendo assim, o efeito prático do uso de tipos genéricos na linguagem C# é o mesmo de se criar um tipo especializado para cada conjunto de parâmetros genéricos que se deseja utilizar.

Em tempo de execução, informações referentes a esses tipos especializados construídos com base nos tipos genéricos são disponibilizadas através de instâncias do tipo `System.Type`, assim como acontece para os tipos não genéricos, porém disponibilizando informações referentes aos parâmetros genéricos utilizados.

O modo como especificar restrições aos parâmetros de um tipo genérico na linguagem C# também é um pouco diferente do modo como isso acontece na linguagem Java.

Exemplo:

```
public class MyList<T> where T : MyClass { //Indica que T deve estender MyClass
    public void Add(T item) { ... }
    public void RemoveAt(int index) { ... }
    public T Retrieve(int index) { ... }
    public int IndexOf(T item) { ... }
}
```

Exemplo 140. Tipos genéricos na linguagem C#

Além disso, o uso de tipos genéricos sem a informação de quais devem ser os parâmetros genéricos utilizados não é permitido.

Exemplo:

```
public class List<T> {
    public void Add(T item) { ... }
    public void RemoveAt(int index) { ... }
    public T Retrieve(int index) { ... }
    public int IndexOf(T item) { ... }
}
List<int> lista1 = new List<int>(); //Parâmetro genérico definido
List lista2 = new List(); //Busca por tipo List não genérico
```

Exemplo 141. Tipos genéricos na linguagem C#

5.21 Coleções

Nas versões atuais, tanto a plataforma Java quanto o .NET Framework fornecem ao desenvolvedor um amplo conjunto de tipos conhecidos como coleções, por permitirem o agrupamento e a manipulação de coleções de instâncias de diversos tipos.

Os tipos de coleção podem ser divididos em genéricos e específicos, sendo que os genéricos permitem que se defina o tipo a ser colecionado no momento da instanciação da coleção, enquanto os específicos geralmente aceitam apenas o tipo object.

5.21.1 Coleções na Linguagem Java

Devido ao modo como os tipos genéricos foram introduzidos na linguagem Java, as coleções existentes puderam continuar sendo utilizadas de modo não genérico, ou seja, utilizando apenas o tipo `object`, e serem utilizadas também de modo genérico, informando-se um tipo específico no momento de sua declaração ou instanciação.

O conjunto de coleções da linguagem Java que será apresentado aqui é conhecido como Java Collections Framework (JCF) (SUN, 2008). Esse framework separa as coleções grupos, definidos pelas seguintes interfaces:

- `Collection`: A interface `java.util.Collection` é usada como base para três dos outros quatro tipos de coleções existentes: `Set`, `List` e `Queue`;
- `Set`: A interface `java.util.Set` é usada para definir coleções que não admitam itens duplicados;

Exemplos:

- `HashSet`: O tipo `java.util.HashSet` armazena seus itens em uma tabela de dispersão, com excelente desempenho, mas sem permitir ordenação dos itens;
- `TreeSet`: O tipo `java.util.TreeSet` armazena seus itens em uma estrutura de árvore, é sensivelmente mais lento que o tipo `HashSet` mas ordena seus itens de acordo com seus valores;
- `LinkedHashSet`: O tipo `java.util.LinkedHashSet`, assim como o `HashSet`, armazena seus itens em uma tabela de dispersão, porém implementada através de uma

lista encadeada, o que permite a ordenação dos itens de acordo com a seqüência de inserções e apresenta desempenho pouco inferior ao tipo `HashSet`.

- `List`: A interface `java.util.List` é usada para definir coleções que admitam itens duplicados, permite manipular o seu conteúdo de acordo com a posição dos seus itens, realizar pesquisas, iterações e operações que afetam um subconjunto de seus itens;

Exemplos:

- `ArrayList`: O tipo `java.util.ArrayList` é a principal implementação da interface `List`;
- `LinkedList`: O tipo `java.util.LinkedList` é uma implementação em formato de lista encadeada, o que proporciona melhor desempenho nas aplicações em que as listas encadeadas são indicadas;
- `Vector`: O tipo `java.util.Vector` é um tipo existente desde as versões iniciais da linguagem Java que fornece uma implementação mais simples para o caso em que arranjos redimensionáveis são desejados. Após a criação do JCF a classe `Vector` foi modificada para também implementar a interface `List`;
- `Queue`: A interface `java.util.Queue` é usada para definir coleções que se comportam como filas, nas quais o primeiro elemento a entrar na fila deve ser o primeiro a sair;

Exemplo:

- `PriorityQueue`: O tipo `java.util.PriorityQueue` é a principal implementação da interface `Queue`;

- Map: A interface `java.util.Map` difere das demais interfaces de coleção pelo fato de permitir a inserção de itens indexados por uma chave;

Exemplos:

- HashMap: O tipo `java.util.HashMap` é a principal implementação da interface `Map`, e pode ser comparada ao tipo `HashSet`;
- TreeMap: O tipo `java.util.TreeMap` é análogo ao tipo `TreeSet`;
- LinkedHashMap: O tipo `java.util.HashMap` é a implementação, em formato de lista encadeada, do tipo `HashMap`;
- Iterable: A interface `java.util.Iterable` é usada para definir coleções que suportem iterações. A seção “Iteradores” apresentará mais detalhes sobre essa interface, implementada por todas as coleções disponíveis no JCF;
- Iterator: A interface `java.util.Iterator` é usada para definir os iteradores retornados pela interface `Iterable`.

A breve descrição apresentada acima já é suficiente para efeitos de comparação com a linguagem C#. Uma cobertura mais detalhada do JCF foge ao escopo deste trabalho.

5.21.2 Coleções na Linguagem C#

Ao contrário do que ocorre na plataforma Java, as coleções disponíveis no .NET Framework são divididas em coleções específicas e coleções genéricas, sendo que as últimas são basicamente implementações genéricas das primeiras.

Para fins de comparação, serão apresentadas apenas algumas das coleções genéricas do .NET Framework:

- `ICollection<T>`: Assim como a interface `Collection` da plataforma Java, a interface `System.Collections.Generic.ICollection<T>`, do .NET Framework, é usada como base para os principais tipos de coleção existentes;
- `IEnumerable<T>`: A interface `System.Collections.Generic.IEnumerator<T>` é usada para especificar tipos que suportam iterações, assim como a interface `Iterable` da plataforma Java;
- `IEnumerator<T>`: A interface `System.Collections.Generic.IEnumerator<T>` é usada para definir os iteradores fornecidos pela interface `IEnumerable<T>`;
- `IList<T>`: A interface `System.Collections.Generic.IList<T>` é usada para definir coleções que suportem o acesso direto a seus itens através de índices, de modo semelhante à interface `List` da plataforma Java;
- `IDictionary<TKey, TValue>`: A interface `System.Collections.Generic.IDictionary<TKey, TValue>` pode ser considerada análoga à interface `java.util.Map`, da plataforma Java.

Há implementações padrões para as interfaces `IList<T>` e `IDictionary<T>`, assim como há implementações dessas, e das demais interfaces citadas, que definem tipos como *queues*, *stacks*, *hash sets* e *linked lists*, de modo semelhante ao que ocorre na plataforma Java, sendo que a maioria dessas implementações utiliza mais de uma interface para obter o comportamento desejado e todas elas implementam a interface `IEnumerable<T>` e sua versão não genérica, `IEnumerable`.

Os arranjos da linguagem C# são construídos de modo a implementarem as interfaces `IEnumerable<T>` e `IEnumerable`, tornando-os polimorficamente compatíveis com tais interfaces.

O .NET Framework, a partir de sua versão 3.5, ainda fornece um conjunto de métodos de extensão, aplicáveis a todas as classes que implementam as interfaces `IEnumerable<T>` e `IEnumerable`, que permitem diversas funções de uso comum a qualquer coleção, como obtenção de valores máximos, mínimos, somatórios, contagem, conversões para outros tipos de coleções dentre várias outras. Consulte os tópicos “Métodos de Extensão” e “LINQ”, do próximo capítulo, para mais informações a respeito.

5.22 Iteradores

Iteradores são objetos capazes de percorrer uma coleção de objetos de modo iterativo, ou seja, sem que seja necessário conhecer previamente todos os itens ou quantos itens compõem tal coleção.

Nas linguagens Java e C#, os iteradores são definidos pelas interfaces `java.util.Iterator` e `System.Collections.IEnumerator`, respectivamente.

5.22.1 Iteradores na Linguagem Java

A interface `java.util.Iterator` define os seguintes métodos:

- `next`: O método `next()` retorna o próximo elemento da coleção;
- `hasNext`: O método `hasNext()` retorna verdadeiro caso haja mais algum item na coleção a ser lido;

- remove: O método `remove()`, que não é de uso obrigatório durante as operações de iteração, remove o último elemento retornado pelo método `next()` da coleção.

Normalmente é adotada a convenção de se implementar uma classe interna à classe de coleção que implemente a interface `Iterator`, e retornar uma instância de tal classe através de um método denominado `iterator()`, conforme definido na interface `Iterable`.

Exemplo:

```
class MyList implements Iterable {
    Iterator iterator() {
        return new MyIterator();
    }
    static class MyIterator implements Iterator {
        int[] itens = {0, 1, 2};
        int index = 0;
        public boolean hasNext() {
            return index < itens.length;
        }
        public Object next() {
            return itens[index++];
        }
        public void remove() { }
    }
}

MyList m = new MyList();
Iterator i = m.iterator();
while (i.hasNext()) {
    System.out.println(i.next());
}
System.in.read();

//Imprime
//0
//1
//2
```

Exemplo 142. Iteradores na linguagem Java

5.22.1.1 O comando *for estendido*

O comando `for` estendido, incluído a partir da versão 1.5 da plataforma Java, permite a iteração direta sobre os itens de uma coleção, assim como apresentado anteriormente na seção “Comandos de Iteração”. Para que uma coleção possa ser iterada, utilizando-se tal comando, a mesma deve implementar a interface `Iterable`.

Exemplo:

A iteração apresentada no exemplo anterior,

```
MyList m = new MyList();
Iterator i = m.iterator();
while (i.hasNext()) {
    System.out.println(i.next());
}
```

pode ser alterada para

```
MyList m = new MyList();
for (Object o : m) {
    System.out.println(o);
}
```

Exemplo 143. O comando for estendido

5.22.2 Iteradores na Linguagem C#

A interface `IEnumerator` do .NET Framework define os seguintes membros:

- `Current`: A propriedade `Current` armazena o item atual da iteração;
- `Reset()`: O método `Reset()` limpa a propriedade `Current` e reinicia a iteração;
- `MoveNext()`: O método `MoveNext()` armazena o próximo item da iteração na propriedade `Current` e retorna `true`, ou retorna `false` caso não haja mais nenhum item a ser iterado.

A interface `IEnumerable` define o método `GetEnumerator()`, que é equivalente ao método `iterator()` da classe `java.util.Iterable`.

Exemplo:

```

class MyList : IEnumerable {
    public IEnumerator GetEnumerator() {
        return new MyEnumerator();
    }
    class MyEnumerator : IEnumerator {
        int[] itens = { 0, 1, 2 };
        int index = 0;
        public object Current { get; private set; }
        public bool MoveNext() {
            if (index < itens.Length) {
                Current = itens[index++];
                return true;
            }
            return false;
        }
        public void Reset() {
            Current = null;
            index = 0;
        }
    }
}

MyList m = new MyList();
IEnumerator i = m.GetEnumerator();
while (i.MoveNext()) {
    Console.WriteLine(i.Current);
}
Console.ReadKey();

//Imprime
//0
//1
//2

```

Exemplo 144. Iteradores na linguagem C#

5.22.2.1 O comando foreach

Através do uso do comando foreach, a iteração do exemplo anterior pode ser alterada conforme o exemplo a seguir:

Exemplo:

```
MyList m = new MyList();
IEnumerator i = m.GetEnumerator();
while (i.MoveNext()) {
    Console.WriteLine(i.Current);
}
```

pode ser alterada para

```
MyList m = new MyList();
foreach (object i in m) {
    Console.WriteLine(i);
}
```

Exemplo 145. O comando foreach

Ao contrário do que ocorre na linguagem Java, o comando `foreach` não está restrito aos objetos que implementem uma interface específica. Como já foi dito na seção “Comandos de Iteração”, basta que o objeto passado como parâmetro para tal comando implemente um método denominado `GetEnumerator()` que retorne um objeto que, por sua vez, implemente os métodos da interface `IEnumerator`, mesmo que tal interface não tenha sido explicitamente referenciada na declaração do tipo.

Exemplo:

```

class MyList {
    public IEnumerator GetEnumerator() { //Nenhuma interface implementada
        return new MyEnumerator(); //Método GetEnumerator()
    }
    class MyEnumerator { //Nenhuma interface implementada
        int[] itens = { 0, 1, 2 };
        int index = 0;
        public int Current { //Propriedade Current
            get; private set;
        }
        public bool MoveNext() { //Método MoveNext()
            if (index < itens.Length) {
                Current = itens[index++];
                return true;
            }
            return false;
        }
        public void Reset() { //Método Reset()
            Current = null;
            index = 0;
        }
    }
}

MyList m = new MyList();
foreach (int i in m) { //Funciona, mesmo MyList não sendo IEnumerable
    Console.WriteLine(i);
}
Console.ReadKey();

//Imprime
//0
//1
//2

```

Exemplo 146. O comando foreach

Note que no exemplo anterior, o tipo da propriedade `Current` não é `object`, como seria caso a interface `IEnumerator` estivesse sendo implementada. Esse é na verdade o objetivo que levou os projetistas da linguagem a optar por não aceitar apenas instâncias da interface `IEnumerable` na comando `foreach`. Do modo como foi implementado, não há nenhuma operação de *boxing* e *unboxing* caso o tipo iterado seja um *Value Type* ou nem mesmo nenhuma conversão de tipos, o que poderia degradar o desempenho da iteração. Provavelmente se os tipos genéricos estivessem disponíveis desde a primeira versão da linguagem, o comando `foreach` seria restrito à versão genérica da interface `IEnumerable`, a

interface `IEnumerable<T>`, a qual utiliza a versão genérica da interface `IEnumerator`, a interface `IEnumerator<T>`.

5.22.2.2 Blocos de iteração

A linguagem C# ainda oferece um recurso conhecido como blocos de iteração, os quais simplificam a implementação de métodos que retornam uma lista de objetos.

Para utilizar esse recurso, deve-se definir um método que retorne uma instância da interface `IEnumerable`, ou de sua versão genérica, `IEnumerable<T>`, e utilizar o comando `yield` em conjunto com o comando `return` para retornar o resultado da iteração, item por item.

Exemplo:

```
public static void Main(string[] args) {
    //Imprime 0 2 4 6 8
    IEnumerable<int> evens1 = Enumerate1(0,10);
    foreach (int i in evens1) {
        Console.Write(i + " ");
    }
    //Imprime 12 14 16 18
    IEnumerable<int> evens2 = Enumerate2(10,20);
    foreach (int i in evens2) {
        Console.Write(i + " ");
    }
    Console.ReadKey();
}

//Enumera usado "yield return"
private static IEnumerable<int> Enumerate1(int s, int e) {
    for (int i = s; i < e; i += 2) {
        yield return i;
    }
}

//Enumera sem usar "yield return"
private static IEnumerable<int> Enumerate2(int s, int e) {
    Queue<int> result = new Queue<int>();
    for (int i = s; i < e; i += 2) {
        result.Enqueue(i);
    }
    return result;
}
```

Exemplo 147. Blocos de iteração

5.23 Paralelismo

A execução de múltiplos programas simultaneamente é um recurso que ajudou bastante na popularização dos computadores pessoais, sendo que atualmente, com o surgimento e evolução dos processadores com múltiplos núcleos, esse tipo de tarefa passou a se mostrar ainda mais satisfatória para o usuário.

Esse recurso, conhecido como multitarefa, ou paralelismo, é possível devido ao modo como os processadores e sistemas operacionais organizam as tarefas que realizam: Cada programa executa em um processo, e cada processo pode ainda ser dividido em diversas linhas de execução (*threads* em inglês), sendo que as linhas de execução de um processo são executadas simultaneamente pelo processador do computador. Quando esse processador possui múltiplos núcleos, algumas linhas de execução são executadas literalmente ao mesmo tempo em que outras, ou seja, paralelamente, permitindo que duas tarefas sejam realizadas gastando o mesmo tempo necessário para se realizar apenas uma delas, por exemplo.

A grande maioria das linguagens de programação oferecem recursos que permitem a implementação de algoritmos que façam uso de múltiplas linhas de execução, sendo que nas linguagens Java e C#, esses recursos permitem que tal tarefa seja executada de maneira extremamente simples.

Para efeitos de comparação, as principais questões relacionadas ao uso de multiplas linhas de execução que devem ser destacadas são:

- Como disponibilizar um modo de se executar uma determinada tarefa paralelamente;
- Como controlar o acesso, por múltiplas linhas de execução, a áreas comuns de memória.

5.23.1 Paralelismo na Linguagem Java

Há várias formas de se trabalhar com linhas de execução na linguagem Java, sendo que as principais são:

- Estender a classe `java.lang.Thread` e sobrescrever o seu método `run()`.

Após isso, deve-se criar uma instância dessa nova classe e invocar o seu método `start()`, o que fará com que uma nova linha de execução seja criada e o método `run()`, sobrescrito, seja executado por uma linha de execução paralela.

Exemplo:

```
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.print(i);
            Thread.sleep(10);
        }
    }
}

Thread t = new MyThread();
t.start();
for (int i = 0; i < 10; i++) {
    System.out.print(i);
    Thread.sleep(5);
}
System.in.read();

//Imprime: 00112234356478596789
```

Exemplo 148. Paralelismo na linguagem Java

- Implementar a interface `java.lang.Runnable` e passar uma instância dessa nova classe como parâmetro para o construtor de uma instância da classe `Thread`.

Após isso, ao invocar o método `start()` da *thread* criada, o método `run()` do objeto `Runnable` passado como parâmetro será executado por uma linha de execução paralela.

Exemplo:

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.print(i);
            Thread.sleep(10);
        }
    }
}

Thread t = new Thread(new MyRunnable());
t.start();
for (int i = 0; i < 10; i++) {
    System.out.print(i);
    Thread.sleep(5);
}
System.in.read();

//Imprime: 00112234356478596789
```

Exemplo 149. Paralelismo na linguagem Java

- Implementar a interface `java.lang.Runnable` e passar suas instâncias para o método `execute()` de uma instância da classe `java.util.concurrent.ThreadPoolExecutor`, o qual irá usar um conjunto de linhas de execução recicláveis, denominado *pool*, para executar as tarefas passadas conforme a disponibilidade de tais linhas.

Exemplo:

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.print(i);
            Thread.sleep(10);
        }
    }
}

BlockingQueue<Runnable> queue = new ArrayBlockingQueue<Runnable>(5);
ThreadPoolExecutor e = new ThreadPoolExecutor(2, 2, 10,
    TimeUnit.SECONDS, queue);

e.execute(new MyRunnable());

for (int i = 0; i < 10; i++) {
    System.out.print(i);
    Thread.sleep(5);
}
System.in.read();

//Imprime: 00112234356478596789
```

Exemplo 150. Paralelismo na linguagem Java

A segunda questão que deve ser considerada ao se trabalhar com múltiplas linhas de execução é o acesso a áreas comuns de memória.

Caso duas ou mais linhas de execução acessem uma mesma variável ao mesmo tempo, não é possível determinar qual valor foi escrito ou lido por cada uma das linhas de execução, sendo assim, deve ser implementado um mecanismo de controle de acesso a essas variáveis. Na linguagem Java, as principais formas de se fazer esse controle são:

- Utilizar o modificador `synchronized` na declaração de um método para informar ao compilador que os acessos à memória realizados por tal método devem ser exclusivos a uma única linha de execução, o que fará com que, quando uma segunda linha de execução tentar acessar tal método, esta deverá aguardar a conclusão da passagem da primeira por ele;

Exemplo:

```
public void synchronized asyncMethod() {
    //Código a ser executado em modo exclusivo
}
```

Exemplo 151. Paralelismo na linguagem Java

- Utilizar o modificador `synchronized` para definir um bloco de código que deva ser executado em modo exclusivo. Nesse caso será necessário informar um objeto para servir de chave para o bloqueio de acesso ao bloco.

Exemplo:

```
public void method() {
    synchronized (this) {
        //Código a ser executado em modo exclusivo
    }
}
```

Exemplo 152. Paralelismo na linguagem Java

- Utilizar o modificador `volatile`, ao definir um campo de um tipo, para especificar que o valor de tal campo deve ser mantido sempre atualizado, de modo que nenhuma otimização realizada pelo processador venha a ocasionar o acesso, por diferentes linhas de execução, a valores desatualizados.

Exemplo:

```
private volatile int sharedNumber;
public void method() {
    sharedNumber = GetValue() ; //Acessa variável em modo exclusivo
}
```

Exemplo 153. Paralelismo na linguagem Java

Muitas outros aspectos devem ser estudados para se compreender bem o uso de multiplas linhas de execução, porém tal estudo está fora do escopo deste trabalho.

5.23.2 Paralelismo na Linguagem C#

Assim como ocorre com a linguagem Java, na linguagem C# também há diversos padrões a serem seguidos para o uso de múltiplas linhas de execução, porém os princípios básicos são os mesmos.

Para definir uma linha de execução, ao invés de estender a classe `Thread`, a linguagem C# nos oferece o uso de *delegates*, que simplificam o processo, permitindo que um delegate, ou seja, um ponteiro para um método ou bloco de código, seja passado diretamente para o construtor da classe `System.Threading.Thread`.

Exemplo:

```
void ThreadStart() {
    Console.WriteLine("Hello World!");
}
Thread t = new Thread(ThreadStart);
t.Start(); //Executa o método ThreadStart em paralelo
```

Exemplo 154. Paralelismo na linguagem C#

Também é possível utilizar um `ThreadPool`, semelhante ao `ThreadPoolExecutor` da linguagem Java, porém no caso da linguagem C#, o tipo `System.Threading.ThreadPool` deve ser acessado como uma classe estática.

Exemplo:

```
void WaitCallback(object state) {
    Console.WriteLine("Hello World!");
}
ThreadPool.QueueUserWorkItem(WaitCallback); //Enfileira o método WaitCallback
```

Exemplo 155. Paralelismo na linguagem C#

Não há, na linguagem C#, um modificador semelhante ao `synchronized` da linguagem Java, que permita a declaração de métodos de execução exclusiva, porém há a cláusula `lock`, que

define blocos de código de acesso exclusivo, de modo semelhante aos definidos pelo modificador `synchronized` em sua outra forma de uso.

Exemplo:

```
public void Method() {
    lock (this) {
        //Código a ser executado em modo exclusivo
    }
}
```

Exemplo 156. Paralelismo na linguagem C#

Já o modificador `volatile` pode ser encontrado com o mesmo nome e com as mesmas funções na linguagem C#.

Exemplo:

```
private volatile int sharedNumber;
public void Method() {
    sharedNumber = ... ; //Acessa variável em modo exclusivo
}
```

Exemplo 157. Paralelismo na linguagem C#

5.24 Anotações ou Atributos

Anotações, na linguagem Java, e Atributos, na linguagem C#, são recursos equivalentes, que possibilitam a atribuição de informação a elementos de código de modo que essa informação possa ser utilizada posteriormente durante a execução do software, através de programação reflexiva, ou pelo ambiente de desenvolvimento durante a verificação e compilação do código.

5.24.1 Anotações na Linguagem Java

Na linguagem Java, anotações são identificadas através do uso do caracter @, precedendo o seu nome e seus parâmetros, antecedendo a declaração de um pacote, tipo, membro de tipo ou variável local.

Exemplo:

```
@MyAnnotation("MyClass");           //Anotação precedendo declaração de tipo
public class MyClass { ... }
```

Exemplo 158. Anotações na linguagem Java

A criação de uma anotação na linguagem Java é feita através da declaração de uma interface que obedeça às seguintes restrições:

- O modificador `interface` deve ser precedido pelo modificador @;
- A interface não pode ser genérica;
- Interfaces de extensão não podem ser informadas. Todas as anotações estendem a `interface annotation.Annotation` implicitamente.
- Os métodos declarados não podem possuir parâmetros;
- Os métodos não podem ser genéricos;
- Os métodos devem ter um tipo de retorno que seja um tipo primitivo, uma `string`, o tipo `Class`, um `enum`, uma anotação ou um arranjo de um desses tipos.
- A clausula `throws` não pode ser utilizada nos métodos;

Exemplo:

```
public @interface MyAnnotation {
    String id();
    String name();
}
```

Exemplo 159. Anotações na linguagem Java

Os métodos de uma anotação definem seus parâmetros, de modo que a interface acima pode ser utilizada conforme o exemplo a seguir:

Exemplo:

```
@MyAnnotation(id="mc", name="MyClass"); //Utiliza a anotação MyAnnotation
public class MyClass { ... } //na classe MyClass
```

Exemplo 160. Anotações na linguagem Java

Caso a anotação tenha um único parâmetro cujo nome seja `value`, a mesma pode ser utilizada sem que o nome do parâmetro seja informado.

Exemplo:

```
public @interface MyAnnotation {
    String value();
}

@MyAnnotation("MyClass");
public class MyClass { ... }
```

Exemplo 161. Anotações na linguagem Java

Valores padrões podem ser informados para os parâmetros das anotações.

Exemplo:

```
public @interface MyAnnotation {
    String id();
    String name() default "no name";
}

@MyAnnotation(id="MyClass");
public class MyClass { ... }
```

Exemplo 162. Anotações na linguagem Java

Um arranjo pode ser passado como parâmetro para uma anotação.

Exemplo:

```
public @interface MyAnnotation {
    String[] value();
}

@MyAnnotation({"A", "B", "C"});
public class MyClass { ... }
```

Exemplo 163. Anotações na linguagem Java

Um tipo pode ser passado como parâmetro para uma anotação.

Exemplo:

```
public @interface MyAnnotation {
    Class value();
}

@MyAnnotation(Integer.class);
public class MyClass { ... }
```

Exemplo 164. Anotações na linguagem Java

Restrições podem ser impostas para parâmetros do tipo Class.

Exemplo:

```
public @interface MyAnnotation {
    Class<? extends MyClass> value();
}

@MyAnnotation(Integer.class);                                     //ERRO DE COMPILAÇÃO
public class MyOtherClass { ... }
```

Exemplo 165. Anotações na linguagem Java

5.24.2 Atributos na Linguagem C#

Os atributos na linguagem C# são utilizados de maneira muito semelhante às anotações da linguagem Java, porém ao invés do caracter @, utiliza-se os caracteres [e] para sua identificação.

Exemplo:

```
[MyAttribute("MyClass")]           //Atributo precedendo a declaração de um tipo  
public class MyClass { ... }
```

Exemplo 166. Atributos na linguagem C#

A criação de um atributo na linguagem C# se dá de modo bastante distinto do modo como as anotações são criadas na linguagem Java.

Para definir um novo atributo na linguagem C#, deve-se criar um novo tipo que obedeça às seguintes regras:

- Deve estender o tipo `System.Attribute`;
- O nome do tipo deve ser terminado pelo sufixo `Attribute`, que pode ser omitido no momento de sua utilização;
- Cada parâmetro do tipo deve ser definido através de uma propriedade pública que tenha como tipo de retorno um tipo simples, uma `string`, o tipo `Type`, um `enum`, uma anotação ou um arranjo de um desses tipos;
- Devem ser criados construtores que recebam os parâmetros de uso mais freqüentes.

Exemplo:

```
public class MyAttributeAttribute : Attribute {
    public MyAttributeAttribute(string id) {
        this.Id = id;
    }
    public string Id { get; set; }
    public string Name { get; set; }
}

[MyAttribute("mc")]
public class MyClass { ... }
```

ou:

```
[MyAttribute(Id="mc",Name="MyClass")]
public class MyClass { ... }
```

ou ainda:

```
[MyAttribute("mc",Name="MyClass")]
public class MyClass { ... }
```

Exemplo 167. Atributos na linguagem C#

Um arranjo pode ser passado como parâmetro para um atributo.

Exemplo:

```
public class MyAttributeAttribute : Attribute {
    public MyAttributeAttribute(string id, string[] names) {
        this.Id = id;
        this.Names = names;
    }
    public string Id { get; set; }
    public string[] Name { get; set; }
}

[MyAttribute("mc", new string[] { "n1", "n2" })]
public class MyClass { ... }
```

Exemplo 168. Atributos na linguagem C#

Um tipo pode ser passado como parâmetro para um atributo.

Exemplo:

```
public class MyAttributeAttribute : Attribute {
    public MyAttributeAttribute(string id, Type type) {
        this.Id = id;
        this.Type = type;
    }
    public string Id { get; set; }
    public Type type { get; set; }
}

[MyAttribute("mc", typeof(string))]
public class MyClass { ... }
```

Exemplo 169. Atributos na linguagem C#

Em ambas as linguagens, é possível utilizar anotações, ou atributos, para restringir a que elementos de código uma anotação, ou atributo, pode ser utilizada.

Exemplo (C#):

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple=false, Inherited=false)]
public class MyAttributeAttribute : Attribute {
    public MyAttributeAttribute(string id) {
        this.Id = id;
    }
    public string Id { get; set; }
    public string Name { get; set; }
}
```

Exemplo 170. Atributos na linguagem C#

Na próxima seção, será visto como utilizar programação reflexiva para obter as anotações, ou atributos, associados a um elemento de código.

5.25 Programação Reflexiva

Programação reflexiva é um conjunto de recursos que permitem que o desenvolvedor tenha acesso à estrutura dos tipos disponíveis, tais como seus membros, modificadores de acesso, anotações ou atributos usados, etc.

Ambas as linguagens disponibilizam recursos de programação reflexiva, sendo que a principal diferença conceitual entre as duas é o fato de a linguagem Java oferecer tais recursos no nível de classes, enquanto a linguagem C# oferece tais recursos no nível de *assembly*.

5.25.1 Programação Reflexiva na Linguagem Java

Na linguagem Java, através do uso do tipo `Class` é possível ter acesso à definição de um tipo de três formas:

- Adicionando o comando `.class` após o tipo que se deseja obter a definição.

Exemplo:

```
Class c = boolean.class;
```

Exemplo 171. Programação reflexiva na linguagem Java

- Utilizando o método `getClass()` da classe `Object`.

Exemplo:

```
byte[] bytes = new byte[1024];
Class c = bytes.getClass();
```

Exemplo 172. Programação reflexiva na linguagem Java

- Utilizando o método estático `forName` do tipo `Class`.

Exemplo:

```
Class c = Class.forName("com.duke.MyLocaleServiceProvider");
```

Exemplo 173. Programação reflexiva na linguagem Java

Após obtido o objeto `Class` que representa o tipo desejado, pode-se usar seus métodos para obter os campos, métodos, anotações e demais definições.

Exemplo:

```
Class c = Program.class;
Field[] fields = c.getFields();
Method[] methods = c.getMethods();
Annotation[] annotations = c.getAnnotations();
```

Exemplo 174. Programação reflexiva na linguagem Java

É possível também criar novas instâncias do tipo com base em sua definição.

Exemplo:

```
Class c = MyClass.class;
Object o = c.newInstance();
```

Exemplo 175. Programação reflexiva na linguagem Java

5.25.2 Programação Reflexiva na Linguagem C#

Na linguagem C#, o acesso à definição de tipos se dá no nível de *assembly*, o que significa que para carregar a definição de um tipo, é necessário primeiro carregar o *assembly* que contém tal tipo, porém isso só é necessário quando o tipo em questão pertence a um *assembly* diferente dos *assemblies* em escopo.

A definição de um tipo é obtida através do tipo `Type`, utilizando-se uma das seguintes abordagens:

- Utilizar o comando `typeof`.

Exemplo:

```
Type t = typeof(boolean);
```

Exemplo 176. Programação reflexiva na linguagem C#

- Utilizar o método `GetType()` a partir de uma instância.

Exemplo:

```
Type t = true.GetType();
```

Exemplo 177. Programação reflexiva na linguagem C#

- Carregando o tipo através de seu nome. Nesse caso, é necessário que o assembly que contém o tipo desejado seja carregado primeiro.

Exemplo:

```
Assembly assembly = Assembly.Load("Bichos");
Type type = assembly.GetType("Bichos.Mamiferos.Cachorro");
```

Exemplo 178. Programação reflexiva na linguagem C#

Após obtido o objeto `Type`, a obtenção dos campos, propriedades, métodos, atributos e demais definições do tipo em questão pode ser feita de modo semelhante ao que ocorre na linguagem Java.

Exemplo:

```
Type type = typeof(MyClass);
FieldInfo[] fields = type.GetFields();
PropertyInfo[] properties = type.GetProperties();
MethodInfo[] methods = type.GetMethods();
object[] attributes = type.GetCustomAttributes(false);
```

Exemplo 179. Programação reflexiva na linguagem C#

Já a criação de instâncias diretamente a partir da definição de um tipo não é possível, porém o tipo estático `Activator` pode ser usado para este fim.

Exemplo:

```
Type type = typeof(MyClass);
object c = Activator.CreateInstance(type);
```

Exemplo 180. Programação reflexiva na linguagem C#

5.26 Acesso a Código Não Gerenciado

Em certas situações, pode ser necessário que um software executado em ambiente gerenciado utilize bibliotecas escritas em código nativo do sistema operacional, com bibliotecas C++ por exemplo.

Em ambas as linguagens, é possível realizar tal acesso através do uso de modificadores reservados para esse fim.

5.26.1 Acesso a Código Não Gerenciado na Linguagem Java

A linguagem Java disponibiliza uma API denominada JNI(Java Native Interface), que permite o acesso a métodos de bibliotecas de código nativo.

Essa API define que deve ser criada uma classe que contenha os métodos da biblioteca nativa que deseja-se utilizar e que faça o carregamento da biblioteca. Tais métodos devem ser declarados sem corpo e utilizando-se o modificador `native`.

Normalmente a biblioteca nativa a ser acessada não é de fato a biblioteca desejada, mas sim uma biblioteca de comunicação, escrita especificamente para esse fim.

A sequencia abaixo ilustra como deve ser feita a preparação de uma classe Java que exponha métodos nativos.

- Primeiro deve-se criar a classe Java que irá expor os métodos `native` e realizar o carregamento da biblioteca de comunicação.

Exemplo:

```
class Bridge {
    static {
        System.loadLibrary("bridge");           //Carrega a biblioteca nativa
    }

    public native String nativeMethod1(String s);
    public native void nativeMethod2();
}
```

Exemplo 181. Acesso a código não gerenciado na linguagem Java

- Feito isso, deve-se gerar um arquivo de cabeçalho para tal biblioteca, usando o aplicativo javah.

Exemplo:

```
jvah Bridge
```

Exemplo 182. Acesso a código não gerenciado na linguagem Java

- Utilizando o arquivo gerado pelo aplicativo javah, implementa-se a biblioteca de comunicação, que acessará a biblioteca nativa desejada;
- Após isso, basta que as bibliotecas nativas estejam disponíveis no caminho de busca da aplicação Java que utiliza a classe que expõe os métodos `native` para que possam ser usados normalmente, como qualquer outro método Java.

5.26.2 Acesso a Código Não Gerenciado na Linguagem C#

A mesma tarefa pode ser executada de modo similar com a linguagem C#, porém não é necessário criar uma biblioteca de comunicação, podendo a biblioteca nativa desejada ser acessada diretamente. Para isso, deve-se utilizar um conjunto de atributos projetados para informar ao ambiente de execução como mapear os métodos, parâmetros e tipos da biblioteca nativa para a biblioteca gerenciada.

Além do uso de tais atributos, o modificador `extern` deve ser utilizado na declaração dos métodos, que, assim como os métodos `native` da linguagem Java, não devem possuir corpo.

Exemplo:

```
class Bridge {
    //Usa DllImport para import a função MessageBox
    //de uma das bibliotecas do Windows.
    [DllImport("user32.dll", CharSet = CharSet.Unicode)]
    public static extern int MessageBox(
        IntPtr hWnd, string text, string caption, uint type);
}
```

Exemplo 183. Acesso a código não gerenciado na linguagem C#

O atributo `DllImport`, utilizado acima, informa ao ambiente de execução qual DLL deve ser carregada, assim como outros parâmetros opcionais, como o nome do método na DLL por exemplo.

Há ainda diversas outras opções de interoperabilidade, disponíveis através do *namespace* `System.Runtime.InteropServices`.

6 RECURSOS EXCLUSIVOS DA LINGUAGEM C#

6.1 Diretivas de Compilação

Na linguagem C#, é possível usar diretivas de compilação para passar alguns tipos de instrução ao compilador, o que não é possível com a linguagem Java.

Exemplo:

```
#define DEBUG //Define a diretiva DEBUG

using System;

class PreprocessorTest{
    public static void Main(string[] args){
        #if DEBUG //Testa a diretiva DEBUG
            Console.WriteLine("DEBUG Mode := On");
        #else
            Console.WriteLine("DEBUG Mode := Off");
        #endif
    }
}
```

Exemplo 184. Diretivas de compilação

6.2 Inicialização de Arranjos, Coleções e Objetos

A linguagem C# permite a inicialização de arranjos, coleções e objetos de maneira atômica.

Exemplo:

```
class Test { internal string f1; internal int f2; }

class Program {
    public static void Main(string[] args) {
        //Instancia e inicializa um arranjo de inteiros
        int[] x1 = new int[] { 0, 1 };
        int[] x2 = new[] { 0, 1 };
        int[] x3 = { 1, 2, 3 };
        //Instancia e inicializa uma lista de inteiros
        List<int> y = new List<int> { 1, 2, 3 };
        //Instancia e inicializa um objeto
        Test z = new Test { f1 = "a", f2 = 5 };
    }
}
```

Exemplo 185. Inicialização de arranjos, coleções e objetos

Enquanto a inicialização de coleções é bastante semelhante à inicialização de arranjos, ou seja, permite a inserção de itens no momento de sua criação, a inicialização de objetos permite que sejam informados valores para suas propriedades.

Inicialização de arranjos é um recurso também existente na linguagem Java, enquanto os demais são recursos exclusivos da linguagem C#.

6.3 Structs

Na linguagem C#, *structs* são estruturas de dados alocados por valor, ou seja, diretamente na memória *stack*, assim como os tipos primitivos da linguagem Java.

A definição de um `struct` é realizada de maneira semelhante a uma classe, porém com algumas restrições:

- Todo `struct` possui um construtor padrão sem parâmetros implícito e não é possível redefinir esse construtor explicitamente. Outros construtores, com parâmetros, podem ser definidos.
- Não é possível utilizar herança. Todo `struct` estende o tipo `System.ValueType` implicitamente. É possível implementar interfaces em um `struct`.
- Não podem ser nulos. Os tipos anuláveis foram criados para permitir que *structs* possam ser, indiretamente, anulados.
- Não é possível inicializar os campos de um `struct` no momento de sua declaração. Por padrão, todos os campos de um `struct` são inicializados com 0 ou `null`;
- Não é permitida a declaração de um destrutor.

Na prática, não há muitos casos em que a criação de um `struct` seja necessária, porém a sua existência permitiu a unificação do sistema de tipos e todas as suas conseqüentes vantagens.

Os poucos casos em que a criação de structs é recomendada são aqueles em que:

- Há a necessidade de semântica de valor ao invés de semântica de referência;
- Existem poucos campos ou propriedades a serem definidos;
- Não há necessidade de herança.

6.4 Passagem de Parâmetros por Referência

Em ambas as linguagens, parâmetros de métodos podem ser passados por valor, ou seja, uma cópia do valor da instância passada como parâmetro é enviada ao método ao invés de uma referência para a mesma instância.

Exemplo (C#):

```
void Foo(string x) {  
    x = "Foo";  
}  
string bar = "Bar";  
Foo(bar);  
Console.WriteLine(bar);
```

//Imprime "Bar"

Exemplo 186. Passagem de parâmetros por referência

Na linguagem C#, também é possível passar parâmetros de métodos por referência. Para isso, o modificar `ref` deve ser utilizado na declaração e na chamada do método.

Exemplo (C#):

```
void Foo(ref string x) {
    x = "Foo";
}
string bar = "bar";
Foo(ref bar);
Console.WriteLine(bar); //Imprime "Foo";
```

Exemplo 187. Passagem de parâmetros por referência

O exemplo abaixo ilustra uma abordagem que poderia ser utilizada para obter o mesmo comportamento na linguagem Java.

Exemplo (Java):

```
class Wrapper { String value; }
void foo(Wrapper x) {
    x.value = "Foo";
}
string bar = "bar";
Wrapper w = new Wrapper();
w.value = bar;
foo(w);
System.out.println(w.value); //Imprime "Foo";
```

Exemplo 188. Passagem de parâmetros por referência

Além do modificador `ref`, a linguagem C# também disponibiliza o modificador `out`, que funciona de modo semelhante, porém tornando obrigatória a atribuição de um novo valor ao parâmetro.

Exemplo:

```
void Foo(out string x) { //OK
    x = "Foo";
}
void Foo(out string x) { //ERRO DE COMPILAÇÃO, x não foi modificado
    return;
}
```

Exemplo 189. Passagem de parâmetros por referência

Sempre que os modificadores `ref` ou `out` forem utilizados, uma variável deve ser passada como parâmetro. A passagem de um valor diretamente não é aceita.

Exemplo:

```
Foo(ref "Bar"); //ERRO DE COMPILAÇÃO
```

Exemplo 190. Passagem de parâmetros por referência

Esses modificadores estão disponíveis tanto para *Value Types* como para *Reference Types*.

6.5 A Cláusula “unsafe”

A linguagem C# permite o uso de trechos de código não seguro, ou seja, que faça acesso direto à memória, através do uso de ponteiros. Para isso, tal código deve estar encapsulando em um bloco identificado pela cláusula `unsafe`.

Exemplo:

```
void MyMethod() {
    //Código seguro
    int i = 5;
    int p;
    p = i;
    p = 10;
    Console.WriteLine(i); //Imprime 5

    //Código não seguro
    unsafe {
        int i = 5;
        int *p;
        p = &i;
        *p = 10;
        Console.WriteLine(i); //Imprime 10
    }
}
```

Exemplo 191. A cláusula `unsafe`

6.6 A Cláusula “using”

A linguagem C# disponibiliza uma cláusula denominada `using`, a qual permite um controle mais preciso sobre a liberação de recursos não gerenciados, ou seja, que possuem dependências externas ao .NET Framework, como por exemplo arquivos ou conexões de rede.

Para utilizar tal cláusula, deve-se acessar tal recurso através de um objeto que implemente a interface `IDisposable` e que realize a liberação do recurso através do método `Dispose()` dessa interface.

O exemplo abaixo utiliza um objeto `FileStream` para abrir um arquivo e ler o seu conteúdo.

Exemplo:

```
class Program {
    public static void Main(string[] args) {
        using (FileStream fs = new FileStream("c:\\temp.txt", FileMode.Open)) {
            StreamReader reader = new StreamReader(fs);
            Console.WriteLine(reader.ReadToEnd());
        }
        Console.ReadKey();
    }
}
```

Exemplo 192. A cláusula `using`

Quando um arquivo é aberto utilizando um objeto `FileStream`, o mesmo fica bloqueado para edição por outros processos até que seja fechado pelo objeto `FileStream` que o abriu.

O fechamento de um arquivo aberto por um objeto `FileStream` pode se dar de duas formas:

- Através do método `Close()`;
- Através da liberação do objeto `FileStream` da memória, a qual invoca o seu método `Dispose()`, que por sua vez invoca o método `Close()`.

Quando a cláusula `using` é utilizada, o objeto criado em sua inicialização é liberado da memória no momento em que o seu bloco é fechado, sem que seja necessário esperar pela próxima execução do coletor de lixo.

6.7 Tipos Anuláveis

Em determinadas situações, a atribuição do valor nulo a variáveis numéricas se mostra mais interessante que a atribuição do valor padrão.

Para atender a essa necessidade, dentre outras, foi introduzido o tipo `System.Nullable<T>` na versão 2.0 da linguagem C#.

Variáveis desse tipo aceitam a atribuição de qualquer um dos valores do tipo `T`, assim como também do valor `null`, permitindo que tais variáveis se comportem como variáveis do tipo `T`, porém anuláveis.

Exemplo:

```
class Program {
    Program() {
        System.Nullable<int> x = null; //OK
        x = 0; //OK
        int y = null; //ERRO DE COMPILAÇÃO
    }
}
```

Exemplo 193. Tipos anuláveis

As seguintes características do tipo `System.Nullable<T>` merecem destaque:

- O operador `?` pode ser utilizado para simplificar a declaração de tipos anuláveis. No exemplo abaixo, as duas declarações são equivalentes.

Exemplo:

```
System.Nullable<int> x = null; //Declara um inteiro anulável
int? x = null; //Declara um inteiro anulável
```

Exemplo 194. Tipos anuláveis

- Como os *Reference Types* já são anuláveis por natureza, o parâmetro genérico `T` só aceita *Value Types*.

Exemplo:

```
string? x = null; //ERRO DE COMPILAÇÃO
```

Exemplo 195. Tipos anuláveis

- A propriedade `HasValue` e o método `GetValueOrDefault()` podem ser utilizados mesmo quando o valor `null` for atribuído à variável.

Exemplo:

```
int? x = null;
if (x.HasValue); //Executa sem erro, mesmo x sendo nulo
int y = x.GetValueOrDefault(); //Executa sem erro, mesmo x sendo nulo
```

Exemplo 196. Tipos anuláveis

- A propriedade `Value` só pode ser acessada caso o valor atribuído à variável seja diferente de `null`.

Exemplo:

```
int? x = null;
int y = x.Value; //Erro ao executar, devido a x ser nulo
```

Exemplo 197. Tipos anuláveis

- Uma conversão explícita deve ser realizada para se atribuir o valor de uma variável do tipo `System.Nullable<T>` a uma variável do tipo `T`.

Exemplo:

```
int? x = 0;
int y = x.GetValueOrValue(); //OK
int z = (int)x; //OK
int w = x.Value; //OK
int k = x; //ERRO DE COMPILAÇÃO
```

Exemplo 198. Tipos anuláveis

A linguagem Java não possui nada semelhante aos tipos anuláveis da linguagem C#, porém como para cada tipo primitivo existe uma classe equivalente, as chamadas *Wrapper Classes*, tais classes podem ser utilizadas quando se deseja anular variáveis de tipos primitivos.

Exemplo (Java):

```
int x = null; //ERRO DE COMPILAÇÃO
Integer y = null; //OK
```

Exemplo 199. Tipos anuláveis

6.8 Tipos Parciais

A linguagem C# permite que tipos tenham sua definição dividida em múltiplos arquivos. Tais tipos são conhecidos como tipos parciais (*partial types* em inglês).

Para a definição de um tipo parcial, basta acrescentar o modificador `partial` à definição do tipo, sendo que no momento da compilação, o tipo gerado será composto por ambas as partes, resultando em um tipo idêntico ao que seria gerado caso sua definição estivesse em um único arquivo.

Exemplo:

Interface qualquer:

```
public interface MyIntf {
    void Foo(); //Define método Foo numa interface
}
```

Arquivo MyClass1.cs:

```
public partial class MyClass {
    public void Foo() { } //Implementa método Foo numa parte
}
```

Arquivo MyClass2.cs:

```
public partial class MyClass : MyIntf { } //Declara a interface em outra parte
```

Gera tipo idêntico ao definido abaixo:

```
public partial class MyClass : MyIntf {
    public void Foo() { }
}
```

Exemplo 200. Tipos parciais

Além disso, a partir da versão 3.0 da linguagem, é possível declarar métodos parciais.

Um método parcial é semelhante a um método abstrato, porém sua implementação é dada em uma outra parte do tipo, ao invés de o ser em um tipo derivado.

Exemplo:

Arquivo MyClass1.cs:

```
public partial class MyClass {
    public partial void Foo();           //Define método parcial
    public MyClass() {                 //Consumo método parcial
        Foo();
    }
}
```

Arquivo MyClass2.cs:

```
public partial class MyClass {
    public partial void Foo() {
        //Corpo do método Foo consumido no construtor
        //definido na outra parte da classe MyClass
    }
}
```

Exemplo 201. Tipos parciais

Assim como os tipos parciais, após a compilação, os métodos parciais são unificados, sendo convertidos em métodos comuns.

Caso uma implementação para um método parcial não seja provida, o mesmo é gerado como um método vazio.

6.9 Sobrecarga de Operadores

A linguagem C# permite que alguns dos operadores da linguagem sejam sobrescritos na definição de um novo tipo. Os novos operadores dizem ao ambiente de execução como realizar as operações definidas por eles quando os tipos envolvidos forem equivalentes aos tipos usados na definição do novo operador. Sempre que um operador for definido, ele irá automaticamente sobrescrever a implementação padrão do mesmo operador.

Exemplo:

```
public class MyClass {
    public int X { get; set; }
    public static int operator +(int i, MyClass m) {
        return m.X + i;
    }
    public static int operator +(MyClass m, int i) {
        return m.X + i;
    }
}

MyClass m = new MyClass();
m.X = 1;
int x = m.X + 5;           //x = 6;
int y = 5 + m.X;          //y = 6;
```

Exemplo 202. Sobrecarga de operadores

Os operadores passíveis de sobrecarga são:

```
+ - ! ~ ++ -- true false
+ - * / % & | ^ << >> == != > < >= <=
```

Também é possível sobrescrever os operadores de conversão implícita e explícita de tipos.

Exemplo:

```
public class MyClass {
    public int X { get; set; }
    public static implicit operator int(MyClass m) {
        return m.X;
    }
    public static explicit operator MyClass(int i) {
        return new MyClass { X = i; };
    }
}

MyClass m = new MyClass();
m.X = 3;
int i = m;           //Faz i = 3
i = 5;
m = (MyClass)i;     //Cria novo MyClass com X = 5;
```

Exemplo 203. Sobrecarga de operadores

6.10 Geração de Tipos e Código em Tempo de Execução

6.10.1 Geração de Tipos em Tempo de Execução

Através dos serviços providos pelo *namespace* `System.Reflection.Emit`, é possível definir e compilar tipos, armazená-los em memória, salvá-los em disco e utilizá-los como se fossem tipos definidos pelo próprio compilador, tudo isso em tempo de execução, o que é muito útil quando se desenvolve ferramentas que dependem fortemente dos parâmetros passados pelo usuário, como por exemplo ferramentas de execução de expressões regulares (JARGAS, 2001).

6.10.2 Geração de Código em Tempo de Execução

Através dos serviços providos pelo *namespace* `System.CodeDom`, é possível criar geradores de código e produzir arquivos de código fonte em diversas linguagens, o que é muito útil quando se desenvolve ferramentas que interagem com o desenvolvedor ou *plugins* para o ambiente de desenvolvimento.

6.11 Inferência de Tipos

Inferência de tipos é um recurso exclusivo da linguagem C# que permite que variáveis locais sejam declaradas sem que seja necessário explicitar o tipo de dado da variável. Para isso, é necessário que a variável seja inicializada no momento da sua declaração.

Exemplo:

```
var x = 0; //Declara um int
var y = 0.0; //Declara um double
var z = "Text"; //Declara uma string
var w = new DateTime(2008,10,15); //Declara um DateTime
var k = new[] {0, 1}; //Declara um arranjo de inteiros
```

Exemplo 204. Inferência de tipos

É importante observar que o uso de inferência de tipos não implica em tipagem fraca (MICROSOFT, 2007, p. 229). No momento da compilação do código, o compilador irá inferir, de acordo com o valor usado na sua inicialização, qual o tipo de dado mais apropriado para a variável. O código, em linguagem intermediária, gerado pelo exemplo a seguir é idêntico ao gerado pelo exemplo anterior.

Exemplo (sem inferência de tipos):

```
int x = 0; //Declara um int
double y = 0.0; //Declara um double
string z = "Text"; //Declara uma string
DateTime w = new DateTime(2008,10,15); //Declara um DateTime
int[] k = new[] {0, 1}; //Declara um arranjo de inteiros
```

Exemplo 205. Inferência de tipos

A principal motivação para a introdução da inferência de tipos na linguagem C#, juntamente com os tipos anônimos, foi possibilitar a implementação de pesquisas integradas à linguagem (LINQ), que serão apresentadas no final deste capítulo.

Não há nada semelhante à inferência de tipos na versão atual da linguagem Java, porém esse é um dos tópicos discutidos em diversos forums, disponíveis na internet, a respeito das novas características a serem incluídas em versões futuras da linguagem.

6.12 Delegates, Métodos Anônimos e Expressões Lambda

Delegates, métodos anônimos e expressões *lambda* serão apresentados em conjunto devido ao fato de serem evoluções de um mesmo recurso. Talvez o principal recurso a diferir as linguagens Java e C#.

Delegates são tipos de dados especiais, os quais estendem o tipo `System.Delegate`, e que atuam como ponteiros para métodos.

Um *delegate* pode apontar para um método estático ou para um método de instância. Neste último caso, uma referência para a instância a qual o método pertence também é armazenada no *delegate*.

Na versão 1.0 da linguagem C#, o uso de *delegates* praticamente se restringia à implementação de eventos.

- Declarava-se o *delegate*;
- Declarava-se o evento utilizando o *delegate* criado como tipo;
- E utilizava-se instâncias do *delegate* para encapsular os métodos passados para o evento.

Exemplo:

```
//Declaração do delegate
delegate void MyDelegate(object sender);

//Declaração do evento
public class MyClass {
    public event MyDelegate MyEvent;
}

//Utilização do delegate
MyClass mc = new MyClass();
mc.MyEvent += new MyDelegate(MyMethod);
void MyMethod(object sender) {
    Console.WriteLine(sender);
}
```

Exemplo 206. Delegates, métodos anônimos e expressões lambda

Com a versão 2.0 da linguagem, a criação do *delegate* para encapsular o método passou a ser feita implicitamente pelo compilador.

Exemplo:

```
MyClass mc = new MyClass();
mc.MyEvent += MyMethod;
void MyMethod(object sender) {
    Console.WriteLine(sender);
}
```

Exemplo 207. Delegates, métodos anônimos e expressões lambda

Além disso, foi incluído um novo recurso, denominado métodos anônimos, que permitiu uma maior praticidade na declaração de tratadores de eventos simples.

Exemplo:

```
MyClass mc = new MyClass();
mc.MyEvent += delegate (object sender) {
    Console.WriteLine(sender);
};
```

Exemplo 208. Delegates, métodos anônimos e expressões lambda

Métodos anônimos são declarados de modo semelhante a um método normal, porém a palavra `delegate` é utilizada no lugar do nome do método e o mesmo deve ser atribuído a uma variável do tipo `delegate` no momento de sua definição.

Com a versão 3.0 da linguagem, o conceito de métodos anônimos foi amplamente generalizado, dando lugar às expressões *lambda*.

Exemplo:

```
MyClass mc = new MyClass();
mc.MyEvent += s => Console.WriteLine(s);
```

Exemplo 209. Delegates, métodos anônimos e expressões lambda

Na verdade a atribuição acima é uma simplificação da atribuição apresentada no exemplo a seguir:

Exemplo:

```
mc.MyEvent += (object s) => { Console.WriteLine(s); };
```

Exemplo 210. Delegates, métodos anônimos e expressões lambda

A sintaxe das expressões *lambda* pode ser complicada à primeira vista, mas permite uma considerável simplificação de notação, além de diversas novas possibilidades de uso, como será visto posteriormente no tópico “LINQ”.

6.13 Métodos de Extensão

Métodos de extensão foram introduzidos na versão 3.0 da linguagem C# visando disponibilizar um modo prático e simples de estender as capacidades de um tipo, sem necessitar modificar o seu conteúdo ou definir um tipo descendente. Em muitas situações, é desejável incorporar um novo método a um tipo que já se encontra amplamente usado e cujos fontes não podem ser modificados. Para tais situações, métodos de extensão podem ser bastante úteis.

Suponha que seja desejável criar um método que, dados um inteiro inicial e um inteiro final, retorne um arranjo contendo os inteiros pertencentes ao intervalo passado. Sem o uso de métodos de extensão, uma das maneiras de se realizar essa operação seria utilizar uma classe auxiliar (*Helper Class*), conforme o exemplo abaixo.

Exemplo:

```
static class Int32Helper {
    internal static int[] Interval(int from, int to) {
        int[] result = new int[to-from+1];
        for (int i = from, c = 0; i < to; i++, c++) {
            result[c] = i;
        }
        return result;
    }
}

class Program {
    public static void Main(string[] args) {
        int[] list = Int32Helper.Interval(1,5);
        foreach (int i in list) {
            Console.WriteLine(i);
        }
        Console.Read();
    }
}
```

Exemplo 211. Métodos de extensão

Através do uso de métodos de extensão, uma abordagem um pouco diferente pode ser utilizada.

Exemplo:

```
static class Int32Extension {
    internal static int[] To(this int from, int to) {
        int[] result = new int[to-from+1];
        for (int i = from, c = 0; i < to; i++, c++) {
            result[c] = i;
        }
        return result;
    }
}

class Program {
    public static void Main(string[] args) {
        int[] list = 1.To(5); //O método To() estende o tipo int
        foreach (int i in list) {
            Console.WriteLine(i);
        }
        Console.Read();
    }
}
```

Exemplo 212. Métodos de extensão

Para definir um método de extensão, deve-se criar uma classe estática com um nome qualquer, para conter o método de extensão, implementar um método estático cujo primeiro

parâmetro seja do tipo que se deseja estender e inserir o modificador `this` antes do tipo do primeiro parâmetro do método.

Feito isso, o compilador irá inserir esse método na lista de métodos de extensão disponíveis sempre que a classe utilizada para armazenar o método estiver no escopo.

Para utilizar o método de extensão, basta acessá-lo como se fosse um método não estático definido no tipo estendido. A instância utilizada para invocar o método será passada como primeiro parâmetro para o método de extensão.

Exemplo:

```
internal static int[] To(this int from, int to) { } //Definido como estático
int[] list = 1.To(5); //Utilizado como não estático
int[] list = Int32Extension.To(1, 5); //Equivalente ao anterior
```

Exemplo 213. Métodos de extensão

6.14 LINQ

Pesquisa integrada à linguagem, ou *Language INtegrated Query* (LINQ), foi o recurso mais significativo incluído na versão 3.0 da linguagem C#. Na verdade, a grande maioria dos recursos inseridos nessa versão foram inseridos para possibilitarem a implementação do LINQ.

O seguinte exemplo ilustra como pode ser realizada uma pesquisa em uma lista de produtos tal que apenas os produtos cujo preço seja inferior a 100,00 sejam retornados.

Exemplo:

```
IEnumerable<Produtos> selecionados = produtos.Where(p => p.Preco < 100.0m);
```

Exemplo 214. Linq

Caso desejássemos apenas os produtos da categoria eletrônicos, ordenados pelo preço, poderíamos fazer conforme o exemplo a seguir:

Exemplo:

```
IEnumerable<Produtos> selecionados = produtos.Where(p =>
    (p.Categoria == "Eletronicos") &&
    (p.Preco < 100.0m)
).OrderBy(p => p.Preco);
```

Exemplo 215. Linq

Os métodos utilizados nesses exemplos são métodos de extensão para a interface `IEnumerable<T>`, ou seja, estão disponíveis para todos os tipos que implementarem tal interface.

Note que o uso de expressões *lambda* como parâmetros, permite especificar critérios de filtro e ordenação complexos, customizados e fortemente tipados.

Juntamente com esse conjunto de métodos de extensão, foram incluídos também novos comandos, que são o que caracteriza o LINQ propriamente dito. Com esses novos comandos, e o uso do recurso de inferência de tipos, a pesquisa anterior poderia ser representada simplificada, conforme o exemplo a seguir:

Exemplo:

```
var selecionados = from produtos
    where p.Categoria == "Eletronicos" &&
        p.Preco < 100.0m
    order by p.Preco
    select p;
```

Exemplo 216. Linq

Como é possível notar, trata-se de uma linguagem muito semelhante à linguagem SQL, já conhecida da grande maioria dos desenvolvedores de software, e assim como ela, o LINQ foi

desenvolvido com base nos conceitos e teoremas da álgebra relacional, permitindo um altíssimo grau de refinamento nas pesquisas realizadas.

Recursos consagrados da linguagem SQL como junções, agrupamentos e agregações também são possíveis utilizando LINQ.

6.14.1 LinqToSQL

Um recurso conhecido como LinqToSQL, ou Dinamic LINQ, ou ainda DLINQ, permite que consultas LINQ sejam traduzidas diretamente para consultas SQL.

Esse recurso faz uso de uma interface denominada `IQueryable`, que traduz as consultas para consultas SQL, executando as no momento da listagem da enumeração.

O exemplo a seguir apresenta a mesma consulta usada nos exemplos anteriores, porém obtendo os dados de um banco de dados SQLServer⁴.

Exemplo:

```
var db = new ProductsDataContext();
var selecionados = from db.Produtos
                   where p.Categoria == "Eletronicos" &&
                       p.Preco < 100.0m
                   order by p.Preco
                   select p;
```

Exemplo 217. LinqToSQL

Nesse caso, há duas mudanças a serem observadas:

⁴ Para mais informações acesse: <http://www.microsoft.com/sql/default.mspx>.

- A inclusão da variável `db`, que representa o contexto de dados (*DataContext*) da consulta, que é o verdadeiro responsável pela compilação da consulta SQL e pelo acesso aos dados;
- O tipo retornado pela consulta, ou seja, o tipo inferido pelo compilador para a variável `selecionados`, que deixa de ser o tipo `IEnumerable<Produtos>`, e passa a ser o tipo `IQueryable<Produtos>`.

Assim como o tipo `IEnumerable<T>`, o tipo `IQueryable<T>`, que o estende, realiza a enumeração dos dados através de um `IEnumerator<T>`, sendo que a consulta SQL só é executada no momento dessa enumeração.

Esse recurso só está disponível para bancos de dados SQL Server, porém outras implementações estão sendo construídas para atender a outros SGBDs.

Para um melhor entendimento das possibilidades e peculiaridades do LINQ, um amplo estudo deve ser realizado. O objetivo desta seção é apenas realizar uma breve apresentação do recurso.

7 CONCLUSÃO

Este trabalho apresentou um extenso estudo comparativo entre as linguagens Java e C#, com um enfoque didático e evolutivo, de modo a permitir ao leitor, ainda que esse tenha pouco conhecimento a respeito de tais linguagens, determinar qual delas se mostra mais adequada a suas necessidades e preferências.

Enquanto a linguagem Java se destaca por ser compacta e objetiva, ainda que exigindo disciplina e uso de padrões de projeto para a implementação de tarefas comuns, como por exemplo o tratamento de eventos, a linguagem C# se mostra flexível e voltada para a produtividade do desenvolvedor, disponibilizando recursos desenvolvidos exclusivamente para tal finalidade.

Se fizermos uma análise a respeito da real necessidade dos recursos disponibilizados por essas linguagens, concluímos que a grande maioria dos recursos existentes na linguagem Java são recursos essenciais, enquanto que na linguagem C# uma grande parte dos recursos são dispensáveis.

Dizer qual delas é a melhor linguagem não é o objetivo desse trabalho, e nem poderia ser, uma vez que tal análise tem caráter muito subjetivo. O que se pode dizer, sem dúvidas, é que são duas linguagens igualmente poderosas e robustas, cada uma com suas particularidades.

REFERÊNCIAS

ALBAHARI, Joseph. Threading in C#. Disponível em: <http://www.albahari.com/threading/threading.pdf>. Acesso em: 30 de Janeiro de 2009.

AHÉ, Peter. Java SE 7 wish list. Disponível em: http://blogs.sun.com/ahe/entry/java_se_7_wish_list. Acesso em: 30 de Janeiro de 2009.

BIGONHA, Roberto, LOBATO, Mário. Métodos Nativos em Java. Disponível em: <http://dcc.ufmg.br/~bigonha/Cursos/Ap/Native/JavaNativeMethod.html>. Acesso em: 30 de Janeiro de 2009.

BODOFF, Stephanie *et al.* Tutorial do J2EE. Rio de Janeiro. Ed. Campus, 2002. 446 p.

CAMPIONE, Mary e WALRATH, Kathy. The Java Tutorial. 1996. Disponível em: <http://www.iam.ubc.ca/guides/javatut/index.html>. Acesso em: 30 de Janeiro de 2009.

ECMA International. ECMA-334: C# Language Specification. 2006. 4ª Ed. Disponível em: <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf>. Acesso em: 30 de Janeiro de 2009.

FLANAGAN, David. Java in a Nutshell. 1999. Disponível em: <http://docstore.mik.ua/oreilly/java-ent/jnut/index.htm>. Acesso em: 30 de Janeiro de 2009.

GAMMA, Erich *et al.* Design Patterns: Elements of Reusable Object Oriented Software. Ed. Addison-Wesley Professional, 1994. 416 p.

GOSLING, James *et al.* The Java Language Specification. 2005. 3ª Ed. Disponível em: <http://java.sun.com/docs/books/jls>. Acesso em 30 de Janeiro de 2009.

HUNDINGAN, Chandra. Understanding Destructors in C#. 2002. Disponível em: <http://www.c-sharpcorner.com/UploadFile/chandrahundigam/UnderstandingDestructors11192005021208AM/UnderstandingDestructors.aspx>. Acesso em: 30 de Janeiro de 2009.

ISO/IEC 14882. Programming Languages – C++. 1998. Disponível em: <http://plumber.gnu-darwin.org/home/pub/Documents/isoiec14882-c%2B%2B-standard.pdf>. Acesso em: 30 de Janeiro de 2009.

ISO/IEC 23270. Information Technology – Programming Languages – C#. 2006. Disponível em: http://webstore.iec.ch/preview/info_isoiec23270%7Bed2.0%7Den.pdf. Acesso em 30 de Janeiro de 2009.

ISO/IEC 7185. Information Technology – Programming Languages – Pascal. 1990. Disponível em: <http://pascal-central.com/docs/iso7185.pdf>. Acesso em 30 de Janeiro de 2009.

ISO/IEC 9899: TC3. Programming Languages – C. 2007. Disponível em: <http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf>. Acesso em: 30 de Janeiro de 2009.

JAGGER, Jon. C# struct/class differences. Disponível em: <http://www.jaggersoft.com/pubs/StructsVsClasses.htm> Acesso em: 30 de Janeiro de 2009.

JARGAS, Aurélio M. Expressões Regulares: Guia de Consulta Rápida. 2001. Disponível em: <http://guia-er.sourceforge.net>. Acesso em: 30 de Janeiro de 2009.

JAVA.NET. JDK7 Project. Disponível em: <https://jdk7.dev.java.net>. Acesso em: 30 de Janeiro de 2009.

JAVAC. Closures for the Java Programming Language. Disponível em: <http://www.javac.info>. Acesso em: 30 de Janeiro de 2009.

JAVACAMP. Java vs C#: Code for Code Comparison. Disponível em: <http://www.javacamp.org/javavscsharp>. Acesso em: 30 de Janeiro de 2009.

KUDALKAR, Chetan. A Tutorial on Nested Classes in C#. Disponível em: http://www.codeproject.com/KB/cs/nested_csclasses.aspx. Acesso em: 30 de Janeiro de 2009.

LIANG, Sheng. The Java Native Interface. Ed. Addison-Wesley, 1999. Disponível em: <http://java.sun.com/docs/books/jni/download/jni.pdf>. Acesso em: 30 de Janeiro de 2009.

MALKS, Alur C. Core J2EE Patterns: As melhores práticas e estratégias de design. Rio de Janeiro. Ed. Campus, 2004. 587 p.

MCCLUSKEY, Glen. Using Java Reflection. 1998. Disponível em: <http://java.sun.com/developer/technicalArticles/ALT/Reflection>. Acesso em: 30 de Janeiro de 2009.

MICROSOFT. C# Language Specification: Version 3.0. 2007. Disponível em: <http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/csharp%20language%20specification.doc>. Acesso em: 30 de Janeiro de 2009.

MOONEY, James D. Binging Portability to the Software Process. Disponível em: http://www.cs.wvu.edu/~jdm/research/portability/reports/TR_97-1.pdf. Acesso em: 30 de Janeiro de 2009.

MSDN. MSDN Library: Visual C#. Disponível em: <http://msdn.microsoft.com/en-us/library/kx37x362.aspx>. Acesso em: 30 de Janeiro de 2009.

NORTHRUP Tony, *et al.* Microsoft .NET Framework 2.0 Application Development Foundation: Self-Paced Training Kit. Ed. Microsoft Press, 2006. 1039 p.

NUNES, Leonardo, CALSAVARA, Alcides. Estudos sobre a concepção de uma linguagem de programação reflexiva e correspondente ambiente de execução. 2001. Programa de Pós-Graduação em Informática Aplicada. Pontifícia Universidade Católica do Paraná – PUCPR. Disponível em: <http://www.sumersoft.com/publicacoes/ArtigoLARVASBLP2001Final.pdf>. Acesso em 30 de Janeiro de 2009.

OBASANJO, Dare. A COMPARISON OF MICROSOFT'S C# PROGRAMMING LANGUAGE TO SUN MICROSYSTEMS' JAVA PROGRAMMING LANGUAGE. 2007. Disponível em: <http://www.25hoursaday.com/CsharpVsJava.html>. Acesso em: 30 de Janeiro de 2009.

SKEET, Jon. C# In Depth: The Beauty of Closures. Disponível em: <http://csharpindepth.com/Articles/Chapter5/Closures.aspx>. Acesso em: 30 de Janeiro de 2009.

STACKOVERFLOW. Does class need to implement IEnumerable to use Foreach? Disponível em: <http://stackoverflow.com/questions/127233>. Acesso em: 30 de Janeiro de 2009.

STACKOVERFLOW. Differences in Generics. Disponível em: <http://stackoverflow.com/questions/31693>. Acesso em: 30 de Janeiro de 2009.

STACKOVERFLOW. List major differences between C# and Java. Disponível em: <http://stackoverflow.com/questions/295224>. Acesso em: 30 de Janeiro de 2009.

STACKOVERFLOW. What are the benefits of the Iterator interface in Java? Disponível em: <http://stackoverflow.com/questions/89891>. Acesso em: 30 de Janeiro de 2009.

STACKOVERFLOW. Why foreach statements accept objects that implement the 'Collection' pattern instead of accept only objects that implement IEnumerable? Disponível em: <http://stackoverflow.com/questions/393152>. Acesso em: 30 de Janeiro de 2009.

SUN Microsystems, Inc. Technical Articles and Tips: Clonning Objects and Using the Serialization API. 2001. Disponível em: <http://java.sun.com/developer/JDCTechTips/2001/tt0306.html>. Acesso em: 30 de Janeiro de 2009.

SUN Microsystems, Inc. The Java 2 Platform Standard Edition 5.0 Overview. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/guide>. 2004. Acesso em 30 de Janeiro de 2009.

SUN Microsystems, Inc. The Java Tutorials. Disponível em: <http://java.sun.com/docs/books/tutorial/index.html>. 2008. Acesso em 30 de Janeiro de 2009.

WIKIPEDIA. Comparison of Java and .NET Platforms. Disponível em: http://en.wikipedia.org/wiki/Comparison_of_the_Java_and_.Net_platforms. Acesso em: 30 de Janeiro de 2009.

WIKIPEDIA. Comparison of Java and C Sharp. Disponível em: http://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Java. Acesso em: 30 de Janeiro de 2009.

WIKIPEDIA. Type System. Disponível em: http://en.wikipedia.org/wiki/Type_system. Acesso em: 30 de Janeiro de 2009.