



**PUC**  
CAMPINAS  
PONTIFÍCIA UNIVERSIDADE CATÓLICA

# **Centro de Ciências Exatas, Ambientais e de Tecnologias**

*Faculdade de Análise de Sistemas  
Curso de Sistemas de Informação*

## **Programação Orientada a Objetos**

*Fundamentos de POO com Java  
1º Semestre de 2018*

**Prof. André Luís dos R.G. de Carvalho**

# Índice

<b>ÍNDICE .....</b>	<b>2</b>
<b>CAPÍTULO I: O PARADIGMA DE ORIENTAÇÃO A OBJETOS.....</b>	<b>6</b>
INTRODUÇÃO .....	7
MOTIVAÇÃO.....	7
INSTÂNCIAS .....	8
CLASSES.....	9
OBJETOS .....	9
MEMBROS DE CLASSE E DE INSTÂNCIA .....	9
ENCAPSULAMENTO .....	10
HERANÇA .....	10
POLIMORFISMO.....	11
SOBRECARGA .....	11
ASSINATURA .....	12
HERANÇA MÚLTIPLA .....	12
CLASSES ABSTRATAS .....	13
CONCLUSÃO .....	13
REFERÊNCIAS .....	14
BIBLIOGRAFIA.....	15
<b>CAPÍTULO II: ORIENTAÇÃO A OBJETOS NA LINGUAGEM DE PROGRAMAÇÃO JAVA .....</b>	<b>16</b>
INTRODUÇÃO .....	17
CARACTERÍSTICAS ENQUANTO LINGUAGEM DE PROGRAMAÇÃO .....	17
O JAVA DEVELOPMENT KIT (JDK).....	20
IDENTIFICADORES .....	21
CONVENÇÕES.....	21
PALAVRAS RESERVADAS .....	22
OPERADORES .....	22
OUTROS SEPARADORES .....	22

<b>COMENTÁRIOS.....</b>	<b>23</b>
<b>SAÍDA DE DADOS.....</b>	<b>23</b>
<b>CLASSE.....</b>	<b>23</b>
<b>COMPILAÇÃO E EXECUÇÃO .....</b>	<b>24</b>
<b>TIPOS BÁSICOS .....</b>	<b>24</b>
<b>CLASSES WRAPPER.....</b>	<b>26</b>
A CLASSE NUMBER.....	26
A CLASSE INTEGER .....	26
A CLASSE LONG.....	26
A CLASSE FLOAT.....	26
A CLASSE DOUBLE.....	26
A CLASSE BOOLEAN.....	27
A CLASSE CHARACTER.....	27
<b>CONSTANTES.....</b>	<b>27</b>
CONSTANTES LITERAIS.....	27
CONSTANTES SIMBÓlicas .....	29
<b>CADEIAS DE CARACTERES.....</b>	<b>29</b>
A CLASSE STRING .....	29
A CLASSE STRINGBUFFER.....	30
A CLASSE STRINGTOKENIZER .....	30
<b>FUNÇÕES OU MÉTODOS .....</b>	<b>30</b>
<b>VARIÁVEIS OU ATRIBUTOS.....</b>	<b>31</b>
<b>VARIÁVEIS LOCAIS E GLOBAIS .....</b>	<b>32</b>
<b>ACESSIBILIDADE DE MEMBROS .....</b>	<b>32</b>
<b>ENTRADA DE DADOS.....</b>	<b>33</b>
<b>A CLASSE MATH.....</b>	<b>37</b>
<b>EXPRESSÕES.....</b>	<b>37</b>
OPERADORES ARITMÉTICOS CONVENCIONAIS.....	38
OPERADORES DE INCREMENTO E DECREMENTO .....	39
OPERADORES RELACIONAIS .....	40
OPERADORES LÓGICOS .....	40
OPERADORES DE BIT.....	40
A Classe BitSet .....	41
CONVERSÕES DE TIPO .....	42
OPERADORES COMBINADOS COM ATRIBUIÇÃO .....	42
EXPRESSÕES CONDICIONAIS.....	43
<b>COMANDOS.....</b>	<b>44</b>
O COMANDO DE ATRIBUIÇÃO .....	44
BLOCOS DE COMANDO.....	45
O COMANDO IF .....	45
O COMANDO SWITCH.....	46
O COMANDO WHILE.....	49

O COMANDO DO WHILE.....	50
O COMANDO FOR.....	50
O COMANDO CONTINUE .....	53
O COMANDO BREAK.....	54
<b>MEMBROS DE CLASSE E DE INSTÂNCIA .....</b>	<b>56</b>
<b>CONSTRUTORES DE CLASSE.....</b>	<b>57</b>
<b>EXCEÇÕES.....</b>	<b>58</b>
<b>GETTERS E SETTERS .....</b>	<b>60</b>
<b>CRIAÇÃO DE INSTÂNCIAS.....</b>	<b>64</b>
<b>CONSTRUTORES DE INSTÂNCIA .....</b>	<b>65</b>
<b>MÉTODOS CANÔNICOS.....</b>	<b>65</b>
TOSTRING.....	65
EQUALS .....	65
HASHCODE .....	66
COMPARETO .....	68
CLONE .....	69
CONSTRUTOR DE CÓPIA .....	69
<b>ORGANIZAÇÃO DE PROGRAMA.....</b>	<b>78</b>
MODULARIZAÇÃO .....	78
PACOTES .....	79
<i>Classes Públicas.....</i>	79
<i>Membros Default (ou de Pacote).....</i>	79
<b>JAVADOC .....</b>	<b>89</b>
JARs .....	93
<b>SOBRECARGA .....</b>	<b>105</b>
<b>CRIAÇÃO E DESTRUÇÃO.....</b>	<b>105</b>
<b>VETORES.....</b>	<b>119</b>
CONSTANTES VETOR.....	119
TAMANHO DAS DIMENSÕES DE UM VETOR .....	119
<b>HERANÇA .....</b>	<b>129</b>
MAIS SOBRE CONVERSÕES DE TIPO.....	130
VERIFICAÇÃO DE TIPO EM JAVA.....	131
MEMBROS PROTEGIDOS .....	131
<b>A CLASSE VECTOR .....</b>	<b>145</b>
<b>INTERFACES .....</b>	<b>145</b>
<b>CLASSEN MEMBRO .....</b>	<b>146</b>
HERANÇA MÚLTIPLA.....	169
<b>A CLASSE JAVA.LANG.OBJECT .....</b>	<b>170</b>
<b>CLASSEN ABSTRATAS .....</b>	<b>170</b>

<b>CLASSES GENÉRICAS (TEMPLATES) .....</b>	<b>186</b>
<b>RECURSÃO .....</b>	<b>196</b>
<b>EXERCÍCIOS .....</b>	<b>197</b>
<b>I. CLASSES E OBJETOS .....</b>	<b>197</b>
<b>II. PACOTES.....</b>	<b>204</b>
<b>III. HERANÇA.....</b>	<b>205</b>
<b>III. CLASSES GENÉRICAS (TEMPLATES).....</b>	<b>210</b>
<b>BIBLIOGRAFIA.....</b>	<b>211</b>

# **Capítulo I:**

# **O Paradigma de**

# **Orientação a Objetos**

## Introdução

Apesar de ser um paradigma relativamente novo, o paradigma de orientação a objetos já pode ser considerado um clássico.

Seu uso vem aceleradamente aumentando com o tempo, tendo se firmado praticamente como um “modismo” dos dias atuais. Parece a concretização da previsão anunciada em [Takahashi88], que dizia que a orientação a objetos viria a ser nos anos 80 e 90 aquilo que programação estruturada foi nos anos 70.

Como praticamente todos os paradigmas de programação, o paradigma de orientação a objetos também surgiu juntamente com o desenvolvimento de uma linguagem de programação.

A primeira linguagem a implementar conceitos de orientação a objetos foi a linguagem Símula, proposta por Dahl e Nygaard em 1966. Posteriormente o conceito foi refinado e desenvolvido na linguagem SmallTalk, proposta por Alan Kay em 1972.

A linguagem SmallTalk não somente norteou o desenvolvimento do paradigma, mas, ainda hoje, é considerada a linguagem que suporta de forma mais completa e pura os seus conceitos.

## Motivação

Podemos dizer que um programa de computador implementa uma solução computacional para um problema do mundo real.

Assim sendo, construir um programa de computador envolve vários processos de abstração e concretização entre dois mundos, o real (onde existe o problema, a solução, e o procedimento para se chegar do problema à solução) e o virtual (onde existe uma abstração do problema, uma abstração da solução, e um procedimento abstrato para chegar de uma na outra).

Naturalmente, existe uma distância conceitual inevitável entre este dois mundos. Diminuir esta distância é justamente uma das principais propostas deste paradigma.

Para tanto, o paradigma parte da premissa de que o mundo real não é composto nem por dados (como pretendem os paradigmas orientados a dados), nem por processos (como pretendem os

---

paradigmas orientados a processos), e sim por entidades semi-autônomas que trabalham e interagem cooperativamente.

Tais entidades são os elementos constituintes fundamentais do paradigma e recebem o nome de objeto.

## Instâncias

Todo instância possui um estado interno para registrar e se lembrar do efeito de sua operação. O estado interno de uma instância é representado por uma área de memória local ao instância, que é inacessível e indevassável.

Todo instância possui ainda um comportamento, que é representado por um repertório de operações que o instância dispõe para responder a mensagens externas ou mesmo internas à própria instância.

O resultado da operação de uma instância depende não só da mensagem que ele recebeu, mas também de seu estado interno.

Mensagens são enviadas de uma instância a outro com a finalidade de que a instância receptora produza algum resultado desejado.

A natureza das operações que a instância receptora executa para produzi-lo é ela quem determina, e poderá provocar alterações em seu estado interno, o envio de novas mensagens para outras instâncias e mesmo para si própria, e até a criação de novas instâncias.

Parâmetros podem acompanhar as mensagens dirigidas a uma instância. Tais parâmetros, por sua vez, também são instâncias, e poderão ter algum efeito sobre as operações que a instância receptora executará quando receber a mensagem que elas acompanham.

A descrição das operações que uma instância executa quando recebe uma mensagem é chamada método. O conceito de método é muito parecido com o conceito de procedimento.

A relação que existe entre mensagens e métodos em uma instância é sempre biunívoca, i.e., uma mesma mensagem somente poderia resultar em métodos diferentes quando enviada para instâncias diferentes.

---

A associação entre mensagens e métodos é sempre pré-definida e estática, i.e., não pode ser alterada em tempo de execução.

## Classes

Uma classe é um modelo de instância, i.e., consiste de descrições de estado, métodos, e associações entre mensagens e métodos, que todas as instâncias pertencentes àquela classe irão possuir.

O conceito de classe é muito parecido com o conceito de tipo abstrato de dados, na medida que uma classe define uma estrutura interna e um conjunto de operações que todas as instâncias daquela classe irão possuir.

Classes também podem receber mensagens, ter métodos e ter um estado interno.

## Objetos

Todo objeto tem uma classe e serve ao propósito de armazenar instâncias da classe à qual pertence.

Instâncias podem ser criadas a partir de uma classe e, no ato de sua criação, ser solicitadas através de uma mensagem a prestar algum serviço, o que será feito pelas execuções do método associado àquela mensagem.

Instâncias somente podem ser solicitadas através de uma mensagem a prestar algum serviço, em um momento posterior àquele de sua criação, quando estiver armazenada em um objeto, que lhe provê, com seu nome, uma forma de ser referenciada.

Assim sendo, se não estiverem armazenadas em um objeto, as instâncias têm uma sobrevida breve, já que perdem a utilidade logo após sua criação.

## Membros de Classe e de Instância

Todos os dados e métodos definidos em uma classe podem ser ditos membros daquela classe. Podemos classificar os membros de uma classe em (1) dados e métodos de classe; e (2) dados e métodos de instância.

---

Entendemos que membros de classe, sejam eles dados ou métodos, são membros relativos à uma classe como um todo e não a nenhum instância individual da classe. Membros de classe expressam propriedades e ações aplicáveis a toda uma classe de instâncias e não a uma instância específica.

Entendemos que membros de instância de classe, sejam eles dados ou métodos, são membros relativos à instâncias individuais e não à classe como um todo. Membros de instância expressam propriedades e ações que são aplicáveis às instâncias de uma classe individualmente e não à classe de modo geral.

## Encapsulamento

O conceito de encapsulamento de certa forma resume tudo o que foi dito até agora sobre o modelo de computação proposto pelo paradigma.

Ele determina que uma instância pode ser vista como o encapsulamento de seu estado interno e de seu comportamento. O mesmo ocorre com uma classe.

## Herança

O conceito de herança se baseia em uma estrutura hierárquica de classes. Em tal estrutura de classes, cada classe pode ter zero ou mais subclasses e zero ou mais superclasses.

Uma dada subclass herda todos os componentes (representação de estado interno e comportamento) de suas superclasses. A classe, em adição às propriedades que herda, pode definir componentes próprios, além de poder redefinir componentes que recebeu por herança.

Vale observar que os componentes de uma subclass recebidos por herança que não forem redefinidos, comportar-se-ão exatamente como componentes definidos na própria classe, e isto sem a necessidade de nenhuma definição ou indicação especial.

Esta propriedade encoraja a definição de novas classes, sem no entanto, requerer uma extensiva duplicação de código [Dershem90].

---

No paradigma imperativo, o conceito mais próximo de herança é o de subtipos. Da mesma forma que uma subclasse herda as propriedades de suas superclasses, um subtipo herda as de seu supertipo.

O conceito de subtipo, no entanto, é muito mais limitado do que o conceito de herança, porque é aplicável somente a um conjunto limitado de tipos básicos, em geral, aos tipos escalares.

O paradigma imperativo não generaliza esta definição para contemplar tipos abstratos de dados, o que faz do conceito de herança uma grande contribuição do paradigma de orientação a instâncias.

## Polimorfismo

Polimorfismo é a propriedade que possibilita que se envie uma mesma mensagem a diferentes instâncias, provocando em cada uma delas uma operação diferente.

Isto acontece porque cada uma das diversas instâncias receptoras potenciais de uma determinada mensagem possui um método próprio para responder à chegada daquela mensagem, dependendo da classe à qual ela pertence.

É importante ressaltar que, ao se enviar uma mensagem, não há a necessidade de se saber nem a classe da instância receptora, nem a forma que esta utilizará para responder à mesma.

No paradigma imperativo, o conceito mais próximo de polimorfismo é o de sobrecarga de operadores e de procedimento.

O conceito de polimorfismo estabelece que a determinação do método a ser invocado, quando da recepção de uma mensagem por uma instância, é feita com base na classe à qual a instância receptora pertence.

## Sobrecarga

O conceito de sobrecarga estabelece a possibilidade de se definir 2 ou mais funções com o mesmo nome de forma que, quando ocorrer a chamada de uma função, dentre as diversas

---

funções de mesmo nome, vai-se determinar aquele que será executado pelo número, tipo e ordem dos parâmetros reais fornecidos por ocasião de sua chamada.

A diferença entre este conceito e o conceito de polimorfismo reside no tempo em que é feita a associação do procedimento a ser executado com a chamada de procedimento.

No polimorfismo, a associação ocorre dinamicamente, uma vez que a classe à qual um instância pertence não é conhecida senão em tempo de execução. Na sobrecarga, tudo se passa como se o número e o tipo dos parâmetros constituíssem uma extensão do nome do procedimento, e a associação pode ser estática.

## **Assinatura**

O nome de um método, juntamente com o conjunto ordenados dos tipos dos parâmetros que ele recebe, costuma ser chamado de assinatura .

## **Herança Múltipla**

Herança Múltipla é um recurso previsto pelo paradigma de orientação a objetos cuja finalidade é permitir que classe herde características não de uma, mas de várias outras classes.

Embora não seja um recurso do qual se precise lançar mão com tanta freqüência, trata-se de um recurso que pode por vezes ser muito útil.

Por exemplo: mesmo dispondo de um conjunto amplo de classes base desenvolvidas, pode ser que, num dado momento, nos vejamos confrontados com a necessidade de um novo tipo de objeto.

Pode ser ainda que este novo tipo de objeto guarde muita semelhança, não com uma, mas com muitas das classes que já temos. Trata-se de um caso típico a ser resolvido com herança múltipla.

Quando uma classe é derivada de mais de uma classe, ela pode ser usada em todos os lugares onde for esperada uma de suas classes base. Trata-se de uma extensão do conceito de polimorfismo.

---

## Classes Abstratas

Numa situação convencional de herança, tem-se que classes bases compartilham com suas classes derivadas, tanto a seu comportamento, quanto sua implementação.

Entretanto, podem haver situações nas quais quase não exista, de fato, o compartilhamento de implementação, embora haja um efetivo compartilhamento de comportamento.

Geralmente não é possível escrever código para muitos dos métodos dessas classes, que, apesar de declarados, não são de fato definidos, existindo sem um corpo onde ocorra sua implementação.

Chamamos este tipo de método de Métodos Abstratos e classes que contêm um ou mais métodos abstratos são chamadas de Classes Abstratas.

Não é permitido criar instâncias de classes abstratas, mas, apesar disso, elas podem ser muito úteis por propiciarem o polimorfismo.

## Conclusão

As linguagens orientadas a objetos nos apresentam uma nova forma de organizar e estruturar programas e isto leva à necessidade de desenvolver uma nova forma de pensar em resolução computacional de problemas.

O Paradigma de Orientação a Objetos conduz naturalmente à implementação de módulos fortemente coesos e fracamente acoplados. Isto torna os programas orientados a objetos mais manuteníveis e suas partes mais reusáveis, facilitando o que em Engenharia de Software chamamos de Peça de Software.

Por tudo isso, cada vez mais encontramos o Paradigma de Orientação a Objetos incorporado nos mais diversos ambientes de computação de uso contemporâneo.

---

Anexo I

## Referências

Dershem, H.L.; and Jipping, M.J., "Programming Languages: Structures and Models", Wadsworth, Inc., 1990.

Takahashi, T., "Introdução à Programação Orientada a Objetos", III EBAI - Escola Brasileiro-Argentina de Informática, 1988.

Anexo II

## Bibliografia

Sebesta, R.W., "Concepts of Programming Languages", The Benjamin/Cummings Publishing Company, Inc., 1989.

Ghezzi, Java., Jazayeri, M., "Programming Languages Concepts", John Wiley & Sons, Inc., 1982.

Dershem, H.L.; and Jipping, M.J., "Programming Languages: Structures and Models", Wadsworth, Inc., 1990.

Pratt, T.W., Programming Languages: Design and Implementation.

Takahashi T.; e Liesemberg, H.K.; "Programação Orientada a Objetos"

Entsminger, G.; "The TAO of Objects: A Beginner's Guide to Object Oriented Programming"

Cox, B.J., Programação Orientada para Instância.Morrison, M.; December, John; et alii, "Java Unleashed", Sams.net Publishing, 1996.

Richardson, J.E.; Schulz, R.C.; e Berard, E.V.; "A Complete Object Oriented Design Example"

---

# **Capítulo II: Orientação a Objetos na Linguagem de Programação Java**

## Introdução

Em meados dos anos 90 a WWW transformava o mundo online. Com um sistema de hipertexto, usuários da WWW se tornaram capazes de selecionar e visualizar informações de toda parte do mundo.

Apesar permitir a seus usuários um alto nível de seletividade de informação, a WWW se limitava a distribuir arquivos de texto, imagem e som, não lhes fornecendo condições para interagir adequadamente com a informação recebida.

Em outras palavras, faltava interatividade verdadeira, em tempo real, dinâmica e visual entre usuários e aplicações WWW.

Java vem suprir esta lacuna, tornando possível a distribuição de aplicações ativas através da WWW, aplicações que podem interagir continuamente com os usuários através do mouse e do teclado e dar-lhes respostas imediatas.

## Características enquanto Linguagem de Programação

É sabido que linguagens de programação que encorajam o desenvolvimento de programas robustos, habitualmente, impõem ao programador boa dose de disciplina na escrita de código fonte. A linguagem C é notoriamente falha neste sentido.

A linguagem C++, que deriva da linguagem C, foi projetada para ser menos permissiva que sua predecessora (e de fato é) mas ainda guarda uma influência excessiva desta linguagem. Isso é um problema e, neste sentido, a linguagem Java é mais rigorosa.

Sabe-se ainda que C++ é uma poderosa linguagem de programação orientada a objetos e, como acontece com a maioria das linguagens projetadas para serem poderosas, ela apresenta uma série de características que propiciam condições favoráveis para o programador desatento cometer erros de programação.

---

Apesar de se basear na linguagem C++, a Java é uma linguagem de programação estruturalmente bastante mais simples que sua predecessora, já que de seu projeto foram eliminadas características que em C++ eram raramente usadas (ou mal usadas) e que podiam levar o programador a incorrer em erros de programação.

Especificamente, Java difere de C++ nos seguintes pontos:

1. Java não suporta structs, unions e ponteiros;
2. Java não suporta typedef nem #define;
3. Java apresenta diferenças em certos operadores e não suporta sobrecarga de operadores;
4. Java não suporta herança múltipla;
5. Java lida com parâmetros da linha de comando de forma diversa daquela utilizada por C ou por C++;
6. Java tem *strings* como parte do pacote Java.lang, o que difere dos vetores terminados com null das linguagens C e C++;
7. Java tem um sistema automático para alocar e liberar memória (coleta de lixo), o que torna desnecessário o uso de funções de alocação e desalocação de memória como é preciso em C e C++.

A linguagem Java suporta multithreading, i.e., oferece facilidades para o programador implementar aplicações que apresentam diversas linhas de execução (aplicações nas quais podem ser encontradas diversas tarefas em processo de execução num mesmo instante).

Diferentemente das linguagens C e C++, a linguagem Java foi especificamente projetada para trabalhar em ambiente de rede, possuindo uma vasta biblioteca de classes para comunicação através de protocolos para Internet da família TCP/IP (HTTP, FTP, etc).

Por esta razão, Java é capaz de manipular recursos via URLs com tão facilmente quanto linguagens como C ou C++ acessam um sistema de arquivos local.

É importante ressaltar que, ao mesmo tempo que essas características representam facilidades para a programação de sistemas distribuídos, representam também uma intensa

---

preocupação com a questão da segurança computacional no sentido mais amplo da expressão.

Java é uma linguagem de arquitetura neutra e é implementada através de uma técnica que leva o nome de interpretação impura.

Tal técnica de implementação de linguagens de programação se caracteriza, principalmente, por lançar mão de um compilador e de um interpretador articulados da seguinte forma: o compilador compila o programa fonte mas, em vez de gerar código executável por uma máquina real, ele gera código para uma máquina virtual implementada em software, ou seja, pelo interpretador.

Em outras palavras, para poder ser executado, o código gerado pelo compilador deverá ser interpretado por um interpretador que implementa a máquina virtual para a qual o compilador gerou código.

Assim, o código compilado pode rodar em qualquer máquina real, naturalmente, desde que haja para tal máquina um interpretador que implemente nela a máquina virtual da linguagem Java.

A linguagem intermediária, na qual o compilador escreve o programa a ser interpretado, chama-se ByteCode. E virtude do processo de interpretação, um programa em Java nem sempre tem o mesmo desempenho que um programa equivalente compilado para uma particular plataforma de hardware.

Por isso, Java prevê a possibilidade de traduzir ByteCodes para a linguagem de máquinas reais, ensejando assim condições para se alcançar a mesma eficiência de um processo de compilação tradicional.

É importante ressaltar que, dizer que uma linguagem é de arquitetura neutra, é muito mais forte e contundente do que dizer que uma linguagem é portável.

Sabemos que é possível acontecer de linguagens como C e C++ (reconhecidamente portáveis) gerarem programas que rodem de maneira ligeiramente diferente em plataformas de hardware diferentes.

---

Tal se deve, principalmente, à diferença de tamanho da palavra de memória nas diferentes arquiteturas de máquina, o que leva variáveis de um mesmo tipo, em diferentes plataformas, terem capacidades diferentes de representação de valores.

Em Java isso não acontece. Internamente, os dados são representados sempre da mesma forma, não importando a plataforma de hardware. Isso faz com que seja possível ter a certeza de que programas Java produzem sempre o mesmo resultado, não importando a plataforma na qual executem.

## O Java Development Kit (JDK)

Distribuído pela SUN Microsystems, o JDK engloba, entre outras ferramentas:

1. Um compilador Java (javac);
  2. Um interpretador de ByteCode que interpreta aplicações para console (java);
  3. Um interpretador de ByteCode que interpreta aplicações para ambiente de janelas (javaw);
  4. Um interpretador de ByteCode que interpreta aplicações integradas em uma página da WEB (appletviewer);
  5. Um depurador (jdb);
  6. Um compactador (jar);
  7. Um utilitário para extrair informações sobre dados e funções, públicos ou não, de um arquivo de ByteCode (javap);
  8. Um utilitário gerador de header files (.h) para integrar ByteCodes em aplicações feitas em linguagem C ou C++ (javah);
  9. Um gerador de documentação (javadoc).
-

## Identificadores

Identificadores introduzem nomes no programa. Em Java podem ser uma seqüência arbitrariamente longa de letras, dígitos, sublinhados (\_) e cífrões (\$). O primeiro caractere de um identificador deve necessariamente ser uma letra, sublinhado (\_) ou cífrão (\$).

É importante ressaltar que, diferentemente de outras linguagens de programação, a linguagem Java diferencia letras maiúsculas de letras minúsculas. Em Java, identificadores que são lidos da mesma forma, mas que foram escritos de forma diferente no que tange ao emprego de letras maiúsculas e minúsculas, são considerados identificadores diferentes.

## Convenções

Para manter em níveis aceitáveis a exigência cognitiva feita aos desenvolvedores, os identificadores, em Java, são nomeados de acordo com regras convencionadas. Naturalmente, é importante conhecê-las e também seguirlas. São elas:

1. Identificadores de constantes devem ser grafados exclusivamente com letras maiúsculas e, no caso de serem compostos por mais de uma palavra, as mesmas deverão ser separadas por um sublinhado (\_);
2. Identificadores de projetos, variáveis, métodos e parâmetros devem ser grafados com letras minúsculas e, no caso de serem compostos por mais de uma palavra, as mesmas deverão ser justapostas e as palavras que sucederem a primeira deverão ser grafadas com inicial maiúscula;
3. Identificadores de classes e interfaces devem ser grafados com inicial maiúscula e demais letras minúsculas e, no caso de serem compostos por mais de uma palavra, as mesmas deverão ser justapostas e seguirem a mesma regra de grafia; e
4. Identificadores de pacotes devem ser grafados exclusivamente com letras minúsculas e, no caso de serem compostos por mais de uma palavra, as mesmas deverão ser justapostas. Identificadores de subpacotes seguem as mesmas regras e devem ser

escritos diante do identificador de seu superpacote, sendo separados dele por um ponto

(•).

## Palavras Reservadas

Estas palavras são identificadores predefinidos e reservados pela linguagem Java para propósitos específicos.

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

## Operadores

Os caracteres e as combinações de caracteres abaixo são reservados para uso como operadores e não podem ser utilizados com outra finalidade. Cada um deles deve ser considerado como um único símbolo.

+	-	*	/	%	&	
^	~	&&		!	<	>
<=	>=	<<	>>	>>>	=	?
++	--	==	+=	-=	*=	/=
%=	&=	=	^=	!=	<<=	>>=
>>>=	.	[	]	(	)	

## Outros Separadores

Os caracteres abaixo são reservados para uso como sinais de pontuação e não podem ser utilizados com outra finalidade.

---

{              }            ;            ,            :

## Comentários

Em Java existem três formas de escrever comentários:

- Iniciando o comentário com os caracteres /\* e terminando com os caracteres \*/
- Iniciando o comentário com os caracteres // e terminando no final da linha
- Iniciando o comentário com os caracteres /\*\* e terminando com os caracteres \*/

## Saída de Dados

Para escrever na saída padrão, basta o envio da mensagem System.out.print (para escrever sem pular de linha) ou System.out.println (para escrever e em seguida pular para a próxima linha). Essas mensagens aceitam um único parâmetro (aíllo que desejamos escrever).

Para escrever na saída padrão de erros, basta o envio da mensagem System.err.print (para escrever sem pular de linha) ou System.err.println (para escrever e em seguida pular para a próxima linha). Essas mensagens aceitam um único parâmetro (aíllo que desejamos escrever).

Cadeias de caracteres podem ser escritas delimitadas por aspas (""). Se desejarmos escrever vários ítems com o envio de uma única mensagem, basta separá-los por um sinal de mais (+).

## Classes

Classes são definidas mencionando a palavra chave class, seguida pelo identificador da classe, seguido por uma série de definições de variáveis (que servem ao propósito de definir a estrutura necessária para representar o estado interno da classe e de suas instâncias) e funções (que servem ao propósito de definir as ações de que dispõem a classe e suas instâncias para responder mensagens que lhes sejam enviadas) delimitados por um par de chaves ({}). Chamamos essas variáveis e funções de membros da classe.

---

Toda classe que deve ser capaz de ser executada deve possuir uma função de nome main que representa o local de início da execução do programa. Esta função deverá receber um vetor de objetos da classe String de tamanho indefinido (cada objeto do vetor representa um parâmetro fornecidos ao programa na linha de comando) e não deverá possuir retorno.

[C:\ExplsJava\Expl\_01\BoasVindas.java]

```
1: public class BoasVindas
2: {
3:     public static void main(String args [])
4:     {
5:         System.out.println ();
6:         System.out.println ("Bem vindos ao estudo de JAVA!");
7:         System.out.println ();
8:     }
9: }
```

## Compilação e Execução

O programa acima, para poder ser executado, deverá primeiro ser compilado através do seguinte comando:

C:\ExplsJava\Expl\_01> javac \*.java

O compilador Java gera, para cada classe presente no arquivo compilado, um arquivo com os ByteCodes resultantes de sua compilação. Tais arquivos recebem sempre o mesmo nome da classe à qual se referem e têm extensão .class.

Assim, após ter sido dado o comando acima, terá sido gerado em disco um arquivo de nome BoasVindas.class. Para executar seu programa, basta dar o seguinte comando:

C:\ExplsJava\Expl\_01> java BoasVindas

Isso produzirá no console a seguinte saída:

Bem vindos ao estudo de JAVA!

## Tipos Básicos

### 1. Inteiros:

---

Os tipos inteiros permitem a declaração de variáveis capazes de armazenar números inteiros. Existem em quatro tamanhos, a saber:

- byte: 8 bits (de -128 a 127);
- short: 16 bits (de -32768 a 32767);
- int: 32 bits (de -2147483648 a 2147483647);
- long 64 bits (de -9223372036854775808 a 9223372036854775807).

## **2. Reais:**

Os tipos reais permitem a declaração de variáveis capazes de armazenar números reais.

Existem em dois tamanhos, a saber:

- float: 32 bits (de  $3,4 \times 10^{-38}$  a  $3,4 \times 10^{+38}$ );
- double: 64 bits (de  $1,7 \times 10^{-308}$  a  $1,7 \times 10^{+308}$ ).

## **3. char:**

O tipo char permite a declaração de variáveis que ocupam 16 bits e que são capazes de armazenar um único caractere. Java não emprega o conjunto ASCII, mas sim o conjunto de caracteres UNICODE, no qual existem 65536 caracteres diferentes.

## **4. boolean:**

O tipo boolean permite a declaração de variáveis booleanas (ou lógicas). Variáveis desse tipo são capazes de armazenar valores que expressam a verdade ou a falsidade.

## **5. void:**

O tipo void especifica um conjunto vazio de valores. Ele é usado como tipo de retorno para funções que não retornam nenhum valor. Não faz sentido declarar variáveis do tipo void.

---

# Classes Wrapper

## A Classe Number

A classe Number é uma classe que oferece uma interface com todos os tipos escalares padrão: int, long, float e double. Possui métodos que retornam o valor potencialmente arredondados do objeto em cada um desses tipos escalares. Veja na documentação da linguagem a interface que a classe especifica para ser compartilhada por todas aquelas classe que dela derivarem.

## A Classe Integer

Derivada da classe Number, a classe Integer possui também outros membros além daqueles herdados de sua classe base. Veja na documentação da linguagem a interface que a classe especifica para comunicação com ela própria e com suas instâncias.

## A Classe Long

Derivada da classe Number, a classe Integer possui também outros membros além daqueles herdados de sua classe base. Veja na documentação da linguagem a interface que a classe especifica para comunicação com ela própria e com suas instâncias.

## A Classe Float

Derivada da classe Number, a classe Integer possui também outros membros além daqueles herdados de sua classe base. Veja na documentação da linguagem a interface que a classe especifica para comunicação com ela própria e com suas instâncias.

## A Classe Double

Derivada da classe Number, a classe Integer possui também outros membros além daqueles herdados de sua classe base. Veja na documentação da linguagem a interface que a classe especifica para comunicação com ela própria e com suas instâncias.

---

## A classe Boolean

Veja na documentação da linguagem a interface que a classe especifica para comunicação com ela própria e com suas instâncias.

## A classe Character

Veja na documentação da linguagem a interface que a classe especifica para comunicação com ela própria e com suas instâncias.

# Constantes

Constantes são instâncias de tipos das linguagens. Temos duas variedades de constantes, a saber, constantes literais e constantes simbólicas.

## Constantes Literais

Constantes literais são aquelas que especificam literalmente uma instância de um tipo da linguagem. A seguir apresentaremos as constantes literais existentes na linguagem Java.

### 1. Constantes Literais do Tipo Inteiro

Uma constante inteira é constituída por uma seqüência de dígitos (eventualmente precedida por um hífen indicando sinal negativo).

Se terminar com letra l (ou L) será considerada do tipo long, caso contrário, será considerada do tipo int.

Se não começar com um dígito 0, será considerada decimal.

Se for precedida por 0x (ou 0X) será considerada hexadecimal. Neste caso, além dos dígitos, também as letras de a (ou A) até f (ou F) podem ser empregadas.

Se for precedida por 0, será considerada octal. Neste caso os dígitos 8 e 9 não podem ser empregados, já que não são dígitos octais válidos.

### 2. Constantes Literais do Tipo Real

---

Uma constante real é constituída por uma constante inteira (indicando a parte inteira da constante real), seguido pelo caractere ponto (indicando o ponto decimal), seguido por outra constante inteira (indicando a parte fracionária da constante real), seguido pelo caractere e (ou E) e por outra constante inteira (indicando a multiplicação por uma potência de 10).

Se terminar com letra f (ou F) será considerada do tipo float, caso contrário, será considerada do tipo double. Também será considerada do tipo double se terminar com a letra d (ou D).

Tanto a parte inteira como a parte fracionária podem ser omitidas (mas não ambas). Tanto o ponto decimal seguida da parte fracionária como o e (ou E) e o expoente podem ser omitidos (mas não ambos). O sufixo de tipo também pode ser omitido.

### 3. Constantes Literais do Tipo char

Uma constantes caractere é constituída por um único caracteres delimitado por apostrofes ('c'). Constantes caractere são do tipo char.

Certos caracteres não visíveis, o apóstrofe ('), as aspas ("") e a barra invertida (\) podem ser representados de acordo com a seguinte tabela de seqüências de caracteres:

'\b'	retrocesso (backspace)
'\t'	tabulação (tab)
'\n'	nova linha (new line)
'\f'	avanço de formulário (form feed)
'\r'	retorno do carro (carriage return)
'\''	apóstrofe (single quote)
'\"''	aspas (double quote)
'\\'	barra invertida (backslash)
'\uHHHH'	caractére unicode (HHHH em hexa)

Uma seqüência como estas, apesar de constituídas por mais de um caractere, representam um único caractere.

#### 4. Constantes do Tipo boolean

Existem somente duas constantes booleanas, a saber: true e false.

#### 5. Constantes do Tipo String

Toda constante literal da classe String é constituída por uma seqüência de zero ou mais caracteres delimitados por aspas ("").

### **Constantes Simbólicas**

Constantes simbólicas são aquelas que associam um nome a uma constante literal, de forma que as referenciamos no programa pelo nome, e não pela menção de seu valor literal.

Somente podem ser definidas diretamente dentro de uma classe, ou, em outras palavras, não podem ser definidas localmente a um bloco de comandos subordinado a um método.

Sendo *Tipo* um tipo, *Const<sub>i</sub>* nomes de identificadores e *Expr<sub>i</sub>* expressões que resultam em valores do tipo *Tipo*, temos que a forma geral de uma declaração de constantes simbólicas é como segue:

```
final Tipo Const1 = Expr1, Const2 = Expr2, ..., Constn = Exprn;
```

## **Cadeias de Caracteres**

### **A Classe String**

Esta classe implementa objetos que representam cadeias de caracteres. Tais objetos são inalteráveis uma vez criados. Veja na documentação da linguagem a interface que a classe especifica para comunicação com ela própria e com suas instâncias.

---

## A Classe **StringBuffer**

Esta classe implementa objetos que representam cadeias de caracteres alteráveis após terem sido criadas. Veja na documentação da linguagem a interface que a classe especifica para comunicação com ela própria e com suas instâncias.

## A Classe **StringTokenizer**

Esta classe implementa métodos capazes de quebrar um *string* em *tokens*. O delimitador pode ser especificado quando da instanciação da classe ou a cada *token*. Tem-se que o delimitador default é o caractere espaço em branco.

Classes que fazem uso desta classe em sua implementação, devem ter logo no início do arquivo onde forem definidas a seguinte diretiva:

```
import java.util.StringTokenizer;
```

ou

```
import java.util.*;
```

Veja na documentação da linguagem a interface que a classe especifica para comunicação com ela própria e com suas instâncias.

## Funções ou Métodos

Convencionalmente entendemos que funções são abstrações de processo, i.e., pelo nome de uma função identificamos e ativamos o processo que ela abstrai, e através de seu retorno e de seus eventuais efeitos colaterais obtemos os resultados de seu processamento.

Num programa orientado a objetos, no entanto, entendemos que as funções representam a descrição das ações que uma classe ou um objeto dispõe a fim de responder mensagens que lhes são dirigidas.

Por isso entendemos que toda função está vinculada à uma classe ou a um objeto, i.e., não faz sentido imaginar uma função “avulsa” como temos em um programa convencional e, no contexto da programação orientada a objetos, são costumeiramente chamadas de métodos.

---

Além disso, entendemos que a chamada de um método representa o envio de uma mensagem à classe ou ao objeto ao qual ela se vincula, e que seu retorno representa a resposta àquela mensagem.

Por isso não faz sentido simplesmente chamar uma função como fazemos numa linguagem convencional. Já que entendemos chamadas de métodos como o envio de uma mensagem, é preciso mencionar o destinatário dessa mensagem, i.e., a classe ou o objeto que deverá recebê-la.

Métodos são definidos mencionando o tipo do retorno do método, seguido pelo identificador do método, seguido por um par de parênteses () contendo, opcionalmente, a lista dos parâmetros formais do método, seguido por um bloco de comandos representando o corpo do método.

Uma lista de parâmetros formais nada mais é do que uma série de definições de parâmetros formais separadas por vírgulas ,. A definição de um parâmetro formal é feita mencionando o nome do tipo do parâmetro e em seguida o identificador do parâmetro formal. É importante ressaltar que, em Java, a única mecanismo de passagem de parâmetro é a passagem por valor.

Não existe em Java o conceito de procedimento, muito embora possamos definir funções que não retornam valor nenhum, o que dá no mesmo. Isto pode ser feito dizendo que o tipo do retorno da função é void.

O comando `return` seguido pela expressão cujo valor se deseja retornar é empregado para fazer o retorno do resultado do método a seu chamante.

## Variáveis ou Atributos

Convencionalmente entendemos que variáveis são células de memória capazes de armazenarem valores para serem futuramente recuperados.

---

Num programa orientado a objetos, no entanto, o principal papel das variáveis é o de representar os atributos de uma classe ou objeto ou, em outras palavras, representar o estado interno de uma classe ou de um objeto.

Variáveis são definidas quando mencionamos o nome de um tipo, e em seguida uma série de identificadores separados por vírgulas (,) e tendo no final um ponto-e-vírgula (;). Cada um dos identificadores será uma variável do tipo que encabeçou a definição.

Sendo *Tipo* um tipo e *Vari<sub>i</sub>* nomes de identificadores, temos que a forma geral de uma declaração de variáveis é como segue:

Tipo Var<sub>1</sub>, Var<sub>2</sub>, ..., Var<sub>n</sub>;

Variáveis podem ser iniciadas no ato de sua definição, i.e., podem ser definidas e receber um valor inicial. Isto pode ser feito acrescentando um sinal de igual (=) e o valor inicial desejado imediatamente após o identificador da variável que desejamos iniciar.

Sendo *Tipo* um tipo, *Var<sub>i</sub>* nomes de identificadores e *Expr<sub>i</sub>* expressões que resultam em valores do tipo *Tipo*, temos que a forma geral de uma declaração de variáveis iniciadas é como segue:

Tipo Var<sub>1</sub> = Expr<sub>1</sub>, Var<sub>2</sub> = Expr<sub>2</sub>, ..., Var<sub>n</sub> = Expr<sub>n</sub>;

## Variáveis locais e globais

Em Java poderemos ainda encontrar variáveis locais servindo a propósitos temporários.

Não existem variáveis globais em Java.

## Acessibilidade de Membros

Atributos e métodos podem ser coletivamente chamados de membros. Podemos estabelecer três níveis de acessibilidade de membros, o que é feito pelo uso de qualificadores específicos. Nesta seção estudaremos dois desses modificadores: o modificador `public` e o modificador `private`.

---

Membros qualificados com `public` são acessíveis a qualquer método, seja ele também membro da classe ou não. Membros qualificados como `private` são acessíveis somente aos métodos escritos na própria classe.

Um programador disciplinado evita qualificar dados variáveis membro com `public`, já que o principal papel destes é descrever o estado interno de classes e objetos (e como o próprio nome diz, o estado interno deve ser interno).

O mesmo não diríamos com respeito aos dados constantes (qualificados com `final`), que podem perfeitamente serem qualificados com `public` sem qualquer prejuízo.

Um programador disciplinado também evita qualificar indiscriminadamente funções membro com `public`. Apenas as funções que descrevem as ações que servem para responder a mensagens externas à classe devem ser públicas.

## Entrada de Dados

Existem algumas alternativas para fazer entrada de dados:

1. A primeira delas, que passaremos, ora, a abordar, exige que se importe, antes de mais nada, da biblioteca de entrada e saída, as classes necessárias. Para tanto, os comandos abaixo devem ser incluídos no início do arquivo no qual acontecerão operações de entrada:

```
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;
```

ou

```
import java.io.*;
```

Mais adiante, neste texto, teremos a oportunidade de compreender melhor as razões da exigência que ora será feita, bem como de conhecer alternativas que permitiriam descumpri-la. No momento, diremos que é preciso que toda função que realiza operações de

---

entrada, assim como toda função que a ativar direta ou indiretamente uma função que realize operações de entrada, tenham acrescentado no final do seu cabeçalho a expressão:

```
throws IOException
```

Deve-se então declarar um objeto da classe BufferedReader que atuará como um prestador de serviços de entrada de dados. Isto pode ser feito como segue:

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

Objetos da classe BufferedReader, como é o caso do objeto Entrada, somente podem ser usados para ler cadeias de caracteres. A solicitação deste serviço pode ser feita como no comando abaixo:

```
String linha = leitor.readLine ();
```

Cadeias de caracteres que contém somente caracteres que conformam constantes literais dos tipos primitivos da linguagem Java, podem ser convertidas nesses mesmos tipos conforme segue:

- **Conversão de String para char:**

```
char C = linha.charAt (0);
```

- **Conversão de String para byte:**

```
byte b = Byte.parseByte (linha);
```

ou, se preferirmos,

```
byte b = new Byte (linha).byteValue ();
```

ou, se preferirmos,

```
byte b = Byte.valueOf (linha).byteValue ();
```

- **Conversão de String para short:**

```
short s = Short.parseShort (linha);
```

ou, se preferirmos,

```
short s = new Short (linha).shortValue ();
```

ou, se preferirmos,

```
short s = Short.valueOf (linha).shortValue ();
```

---

- **Conversão de String para int:**

```
int i = Integer.parseInt (linha);  
ou, se preferirmos,  
int i = new Integer (linha).intValue ();  
ou, se preferirmos,  
int i = Integer.valueOf (linha).intValue ();
```

- **Conversão de String para long:**

```
long l = Long.parseLong (linha);  
ou, se preferirmos,  
long l = new Long (linha).longValue ();  
ou, se preferirmos,  
long l = Long.valueOf (linha).longValue ();
```

- **Conversão de String para float:**

```
float f = Float.parseFloat (linha);  
ou, se preferirmos,  
float f = new Float (linha).floatValue ();  
ou, se preferirmos,  
float f = Float.valueOf (linha).floatValue ();
```

- **Conversão de String para double:**

```
double d = Double.parseDouble (linha);  
ou, se preferirmos,  
double d = new Double (linha).doubleValue ();  
ou, se preferirmos,  
double d = Double.valueOf (linha).doubleValue ();
```

- **Conversão de String para boolean:**

```
boolean b = Boolean.parseBoolean (linha);  
ou, se preferirmos,  
boolean b = new Boolean (linha).booleanValue ();
```

---

ou, se preferirmos,

boolean b = Boolean.valueOf(linha).booleanValue().

### [C:\ExplsJava\Expl\_02\BoasVindas.java]

```
1: import java.io.IOException;
2: import java.io.BufferedReader;
3: import java.io.InputStreamReader;
4:
5: public class BoasVindas
6: {
7:     public static void main (String args []) throws IOException
8:     {
9:         BufferedReader leitor = new BufferedReader (
10:             new InputStreamReader (
11:                 System.in));
12:
13:         System.out.println ();
14:
15:         System.out.print ("Qual o seu nome? ");
16:         String nome = leitor.readLine ();
17:
18:         System.out.println ("Bem vindo(a) ao estudo de JAVA, " +
19:             nome +
20:             "!");
21:
22:         System.out.println ();
23:
24:         System.out.print ("Com que nota voce pretende passar? ");
25:
26:         float nota = Float.parseFloat (leitor.readLine ());
27:         /*
28:         ou
29:         float nota = Float.valueOf (leitor.readLine ()).intValue ();
30:         ou
31:         float nota = new Float (leitor.readLine ()).intValue ();
32:         */
33:
34:         System.out.println ();
35:
36:         System.out.println (nome +
37:             ", desejo sucesso a voce;");
38:
39:         System.out.println ("que voce passe com " +
40:             nota +
41:             " ou mais!");
42:
43:         System.out.println ();
44:     }
45: }
```

Para compilar e executar este programa, daremos os seguintes comandos:

---

```
C:\ExplsJava\Expl_02> javac *.java  
C:\ExplsJava\Expl_02> java BoasVindas
```

Isto poderia produzir no console a seguinte interação:

```
Qual o seu nome? Jose  
Bem vindo ao estudo de JAVA, Jose!
```

```
Com que nota voce pretende passar? 7.0  
Jose, desejo sucesso a voce;  
que voce passe com 7.0 ou mais!
```

2. A segunda delas, que passaremos, ora, a abordar, exige que se importe, antes de mais nada, da biblioteca de entrada e saída, as classes necessárias. Para tanto, os comandos abaixo devem ser incluídos no ínicio do arquivo no qual acontecerão operações de entrada:

## A Classe Math

A classe Math contém muitas funções matemáticas úteis, bem como muitas constantes também úteis. A classe Math não deve ser instanciada e, como trata-se de uma classe final, não pode ser usada como base para derivação de outra classe.

Veja na documentação da linguagem a interface que a classe especifica para comunicação com ela própria.

## Expressões

Uma expressão é uma seqüência de operadores e operandos que especifica um computação e resulta um valor.

A ordem de avaliação de subexpressões é determinada pela precedência e pelo agrupamento dos operadores. As regras matemáticas usuais para associatividade e comutatividade de operadores podem ser aplicadas somente nos casos em que os operadores sejam realmente associativos e comutativos.

---

A ordem de avaliação dos operandos de operadores individuais é indefinida. Em particular, se um valor é modificado duas vezes numa expressão, o resultado da expressão é indefinido, exceto no caso dos operadores envolvidos garantirem alguma ordem.

## **Operadores Aritméticos Convencionais**

Os operadores aritméticos da linguagem Java são, em sua maioria, aqueles que usualmente encontramos nas linguagens de programação imperativas. Todos eles operam sobre números e resultam números. São eles:

1. + (soma);
2. - (subtração);
3. \* (multiplicação);
4. / (divisão);
5. % (resto da divisão inteira); e
6. - (menos unário).

Não existe em Java um operador que realize a divisão inteira, ou, em outras palavras, em Java uma divisão sempre resulta um número real. Para resolver o problema, podemos empregar um conversão de tipo para forçar o resultado da divisão a ser um inteiro. Neste caso, o real resultante terá truncada a sua parte fracionária, se transformando em um inteiro.

O operador % (resto da divisão inteira) opera sobre dois valores inteiros. Seu resultado também será um valor integral. Ele resulta o resto da divisão inteira de seu primeiro operando por seu segundo operando.

O operador - (menos unário) opera sobre um único operando, e resulta o número que se obtém trocando o sinal de seu operando.

---

## ***Operadores de Incremento e Decremento***

Os operadores de incremento e decremento da linguagem Java não são usualmente encontrados em outras linguagens de programação. Todos eles são operadores unários e operam sobre variáveis inteiras e resultam números inteiros. São eles:

1. `++` (prefixo);
2. `--` (prefixo);
3. `++` (pósfixo);
4. `--` (pósfixo).

O operador `++` (prefixo) incrementa seu operando e produz como resultado seu valor (já incrementado).

O operador `--` (prefixo) decremente seu operando e produz como resultado seu valor (já decrementado).

O operador `++` (pósfixo) incrementa seu operando e produz como resultado seu valor original (antes do incremento).

O operador `--` (pósfixo) decremente seu operando e produz como resultado seu valor original (antes do decremento).

Observe, abaixo, uma ilustração do uso destes operadores:

```
...
int a=7, b=13, c;
System.out.println (a+" "+b);           // 7 13

a--; b++;
System.out.println (a+" "+b);           // 6 14

--a; ++b;
System.out.println (a+" "+b);           // 5 15

c = a++ + ++b;
System.out.println (a+" "+b+" "+c); // 6 16 21
...
```

## **Operadores Relacionais**

Os operadores relacionais da linguagem Java são, em sua maioria, aqueles que usualmente encontramos nas linguagens de programação imperativas. Todos eles operam sobre números e resultam valores lógicos. São eles:

1. == (igualdade);
2. != (desigualdade);
3. < (inferioridade);
4. <= (inferioridade ou igualdade);
5. > (superioridade); e
6. >= (superioridade ou igualdade).

## **Operadores Lógicos**

Os operadores lógicos da linguagem Java são, em sua maioria, aqueles que usualmente encontramos nas linguagens de programação imperativas. Todos eles operam sobre valores lógicos e resultam valores lógicos. São eles:

1. && (conjunção ou *and*);
2. || (disjunção ou *or*); e
3. ! (negação ou *not*).

## **Operadores de Bit**

Os operadores de bit da linguagem Java são, em sua maioria, aqueles que usualmente encontramos nas linguagens de programação imperativas. Todos eles operam sobre valores inteiros e resultam valores inteiros. São eles:

1. & (and bit a bit);
  2. | (or bit a bit);
-

3.  $\wedge$  (xor bit a bit);
4.  $\sim$  (not bit a bit);
5.  $<<$  (shift left bit a bit); e
6.  $>>$  (shift right bit a bit).

Observe, abaixo, uma ilustração do uso destes operadores:

```
...
short a,
    b=0x7C3A, // 0111 1100 0011 1010
    c=0x1B3F; // 0001 1011 0011 1111

a = b & c;      // 0001 1000 0011 1010
a = b | c;      // 0111 1111 0011 1111
a = b ^ c;      // 0110 0111 0000 0101
a = ~b;         // 1000 0011 1100 0101
a = b << 3;     // 1110 0001 1101 0000
a = c >> 5;     // 0000 0000 1101 1001
...
...
```

### A Classe BitSet

Esta classe implementa um tipo de dados que representa uma coleção de bits. A coleção crescerá dinamicamente a medida que mais bits forem necessários. Bits específicos são identificados por um inteiro não negativo. O primeiro bit sempre é o de ordem zero.

Classes que fazem uso desta classe em sua implementação, devem ter logo no início do arquivo onde forem definidas a seguinte diretiva:

```
import java.util.BitSet;
```

ou

```
import java.util.*;
```

Veja na documentação da linguagem a interface que a classe especifica para comunicação com ela própria e com suas instâncias.

---

## Conversões de Tipo

Expressões podem ser forçadas a resultar um certo tipo se precedidas pelo tipo desejado entre parênteses.

Sendo Tipo um tipo e Expr uma expressão que resulta um valor que não é do tipo Tipo, temos que:

(Tipo) Expr

resulta a expressão Expressão convertida para o tipo Tipo.

Observe, abaixo, uma ilustração do uso deste operador:

```
...
int a = 65;
char c = (char)a;

System.out.println (c); // A
...
```

## Operadores combinados com Atribuição

Os operadores aritméticos e de bit podem ser combinados com o operador de atribuição para formar um operador de atribuição com operação embutida.

Sendo Var uma variável, Opr um operador aritmético ou de bit e Expr uma expressão, temos que:

Var Opr= Expr;

é equivalente a

Var = Var Opr (Expr);

Observe, abaixo, uma ilustração do uso destes operadores:

```
...
short a, b, c, d, e;

a = a + (3 * b);           // a += 3*b;
b = b / (a - Math.sin (a)); // b /= a-Math.sin(a);
c = c << d;               // c <<= d;
d = d & (a + b);           // d &= a+b;
e = e >> (d + 3);          // e >>= d+3;
...
```

---

## Expressões Condicionais

Expressões condicionais representam uma forma compacta de escrever comandos a escolha de um dentre uma série possivelmente grande de valores. Substituem uma seqüência de ifs.

Sendo Cond uma condição booleana e Expr<sub>i</sub> expressões, temos que:

Cond ? Expr<sub>1</sub> : Expr<sub>2</sub>

resulta Expr<sub>1</sub> no caso de Cond ser satisfeita, e Expr<sub>2</sub>, caso contrário.

Observe, abaixo, uma ilustração do uso deste operador:

```
...
int a, b;
...
System.out.println (a>b?a:b);
...

equivale a

...
int a, b;
...
int maior;

if (a>b)
    maior = a;
else
    maior = b;

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

[C:\ExplsJava\Expl\_03\RaizEquacao.java]

```
1: import java.io.IOException;
2: import java.io.BufferedReader;
3: import java.io.InputStreamReader;
4:
5: public class RaizEquacao
6: {
7:     public static void main (String args []) throws IOException
8:     {
9:         BufferedReader leitor = new BufferedReader
10:            (new InputStreamReader
11:             (System.in));
12:
13:         System.out.println ();
14:         System.out.print ("PROGRAMA PARA CALCULAR RAIZ");
15:         System.out.print (" DE EQUACAO DE 1o GRAU");
```

```
16:         System.out.println ();
17:
18:         System.out.print ("Coeficiente a: ");
19:         double a = Double.parseDouble (leitor.readLine ());
20:
21:         System.out.print ("Coeficiente b: ");
22:         double b = Double.parseDouble (leitor.readLine ());
23:
24:         double raiz = -b/a;
25:
26:         System.out.println ();
27:         System.out.println ("Raiz: " + raiz);
28:         System.out.println ();
29:     }
30: }
```

Para compilar e executar este programa, daremos os seguintes comandos:

```
C:\ExplsJava\Expl_03> javac *.java
C:\ExplsJava\Expl_03> java RaizEquasao
```

Isto poderia produzir no console a seguinte interação:

PROGRAMA PARA CALCULAR RAIZ DE EQUACAO DE 1º GRAU

```
Coeficiente a: 2
Coeficiente b: -10
```

```
Raiz: 5
```

## Comandos

Comandos são a unidade básica de processamento. Em Java não existe um repertório muito vasto de comandos. Muitos comandos que em outras linguagens existem, em Java são encontrados em bibliotecas de classe.

Comandos em Java, salvo raras exceções, são terminados por um ponto-e-vírgula (;).

## O Comando de Atribuição

O comando de atribuição tem a seguinte forma básica: em primeiro lugar vem o identificador da variável receptora da atribuição, em seguida vem o operador de atribuição (=), em seguida vem a expressão a ser atribuída, em seguida vem um ponto-e-vírgula (;).

---

Sendo Var o identificador de uma variável e Expr uma expressão, temos abaixo a forma geral do comando de atribuição:

```
Var = Expr;
```

## Blocos de Comando

Já conhecemos o conceito de bloco de comandos, que nada mais é do que uma série de comandos delimitada por um par de chaves ({}).

Blocos de comando determinam um escopo de declarações e nunca são terminados por ponto-e-vírgula (;).

Blocos de comando são muito úteis para proporcionar a execução de uma série de subcomandos de comandos que aceitam apenas um subcomando.

## O Comando if

O comando if tem a seguinte forma básica: em primeiro lugar vem a palavra-chave if, em seguida vem, entre parênteses, a condição a ser avaliada, em seguida vem o comando a ser executado no caso da referida condição se provar verdadeira.

Sendo Cond uma condição booleana e Cmd um comando, temos abaixo a forma geral do comando if (sem else):

```
if (Cond)
    Cmd;
```

O comando if pode também executar um comando, no caso de sua condição se provar falsa. Para tanto, tudo o que temos que fazer é continuar o comando if que já conhecemos, lhe acrescentando a palavra chave else, e em seguida o comando a ser executado, no caso da referida condição se provar falsa.

Sendo Cond uma condição booleana e Cmd<sub>i</sub> dois comandos, temos abaixo forma geral do comando if com else:

```
if (Cond)
    Cmd1;
```

---

```
        else  
            Cmd2;
```

Observe que, tanto no caso da condição de um comando `if` se provar verdadeira, quanto no caso dela se provar falsa, o comando `if` somente pode executar um comando. Isso nem sempre, ou melhor, quase nunca é satisfatório; é comum desejarmos a execução de mais de um comando.

Por isso, nos lugares onde se espera a especificação de um comando, podemos especificar um bloco de comandos.

## O Comando `switch`

O comando `switch` tem a seguinte forma: em primeiro lugar vem a palavra-chave `switch`, em seguida vem, entre parênteses, uma expressão integral a ser avaliada, em seguida vem, entre chaves (`{ }` ), uma série de casos.

Um caso tem a seguinte forma básica: em primeiro lugar vem a palavra-chave `case`, em seguida vem uma constante integral especificando o caso, em seguida vem o caractere dois pontos (`:`), em seguida vem uma série de comandos a serem executados no caso.

Sendo `Expr` uma expressão, `ConstI` constantes literais do mesmo tipo que `Expr` e `CmdII` comandos, temos abaixo uma das formas gerais do comando `switch`:

```
switch (Expr)  
{  
    case Const1:    Cmd1a;  
                    Cmd1b;  
                    ...  
    case Const2:    Cmd2a;  
                    Cmd2b;  
                    ...  
    case Const3:    Cmd3a;  
                    Cmd3b;  
                    ...  
    ...  
}
```

Se, em alguma circunstância, desejarmos executar uma mesma série de comandos em mais de um caso, tudo o que temos a fazer é especificar em seqüência os casos em questão, deixando para indicar somente no último deles a referida seqüência de comandos.

Sendo Expr uma expressão, Const<sub>I,J</sub> constantes literais do mesmo tipo que Expr e Cmd<sub>I,J</sub> comandos, temos abaixo uma das formas gerais do comando switch:

```
switch (Expr)
{
    case Const1,1:
    case Const1,2:
    ...
        Cmd1a;
        Cmd1b;
        ...
    case Const2,1:
    case Const2,2:
    ...
        Cmd2a;
        Cmd2b;
        ...
    case Const3,1:
    case Const3,2:
    ...
        Cmd3a;
        Cmd3b;
        ...
}
```

Se, em alguma circunstância, desejarmos executar uma série de comandos qualquer que seja o caso, tudo o que temos a fazer é especificar um caso da forma: em primeiro lugar vem a palavra-chave default, em seguida vem o caractere dois pontos (:), em seguida vem uma série de comandos a serem executados qualquer que seja o caso.

Sendo Expr uma expressão, Const<sub>I,J</sub> constantes literais do mesmo tipo que Expr e Cmd<sub>I,J</sub> comandos, temos abaixo uma das formas gerais do comando switch:

```
switch (Expr)
{
    case Const1,1:
    case Const1,2:
    ...
        Cmd1a;
        Cmd1b;
        ...
    case Const2,1:
```

```
case Const2,2:
...
Cmd2a;
Cmd2b;
...
...
default:
CmdDa;
CmdDb;
...
}
```

É importante ficar claro que o comando switch não se encerra após a execução da seqüência de comandos associada a um caso; em vez disso, todas as seqüências de comandos, associadas aos casos subsequentes, serão também executadas.

Se isso não for o desejado, basta terminar cada seqüência (exceto a última), com um comando break.

Sendo Expr uma expressão, Const<sub>I,J</sub> constantes literais do mesmo tipo que Expr e Cmd<sub>I,J</sub> comandos, temos abaixo uma das formas gerais do comando switch com breaks:

```
switch (Expr)
{
    case Const1,1:
    case Const1,2:
    ...
        Cmd1a;
        Cmd1b;
        ...
        break;

    case Const2,1:
    case Const2,2:
    ...
        Cmd2a;
        Cmd2b;
        ...
        break;

    ...
    default:
        CmdDa;
        CmdDb;
        ...
}
```

Observe, abaixo, uma ilustração do uso deste comando:

```
...
int a, b, c;
```

---

```
char operacao;
...
switch (operacao)
{
    case '+': a = b + c;
                break;

    case '-': a = b - c;
                break;

    case '*': a = b * c;
                break;

    case '/': a = b / c;
}
System.out.println (a);
...
```

## O Comando while

O comando `while` tem a seguinte forma básica: em primeiro lugar vem a palavra-chave `while`, em seguida vem, entre parênteses, a condição de iteração, em seguida vem o comando a ser iterado. A iteração se processa enquanto a condição de iteração se provar verdadeira.

Sendo `Cond` uma condição booleana e `Cmd` um comando, temos abaixo a forma geral do comando `while`:

```
while (Cond)
    Cmd;
```

Observe que, conforme especificado acima, o comando `while` itera somente um comando. Isso nem sempre, ou melhor, quase nunca é satisfatório; é comum desejarmos a iteração de um conjunto não unitário de comandos.

Por isso, em lugar de especificar o comando a ser iterado, podemos especificar um bloco de comandos. Assim conseguiremos o efeito desejado.

---

## O Comando do-while

O comando `do-while` tem a seguinte forma básica: em primeiro lugar vem a palavra-chave `do`, em seguida vem o comando a ser iterado, em seguida vem a palavra-chave `while`, em seguida vem, entre parênteses, a condição de iteração. A iteração se processa enquanto a condição de iteração se provar verdadeira.

A diferença que este comando tem com relação ao comando `while` é que este comando sempre executa o comando a ser iterado pelo menos uma vez, já que somente testa a condição de iteração após tê-lo executado. Já o comando `while` testa a condição de iteração antes de executar o comando a ser iterado, e por isso pode parar antes de executá-lo pela primeira vez.

Sendo `Cond` uma condição booleana e `Cmd` um comando, temos abaixo a forma geral do comando `do-while`:

```
do
    Cmd;
    while (Cond);
```

Observe que, conforme especificado acima, o comando `do-while` itera somente um comando. Isso nem sempre, ou melhor, quase nunca é satisfatório; é comum desejarmos a iteração de um conjunto não unitário de comandos.

Por isso, em lugar de especificar o comando a ser iterado, podemos especificar um bloco de comandos. Assim, conseguiremos o efeito desejado.

## O Comando for

A sintaxe clássica do comando `for` tem a seguinte forma básica: em primeiro lugar vem a palavra-chave `for`, em seguida vem, entre parênteses e separadas por pontos-e-vírgulas (`;`), a sessão de iniciação, a condição de iteração, e a sessão de reiniciação. Em seguida vem o comando a ser iterado. A iteração se processa enquanto a condição de iteração se provar verdadeira.

---

No caso de haver mais de um comando na sessão de iniciação ou na sessão de reiniciação, estes devem ser separados por vírgulas (, ).

Sendo  $\text{Cmd}_{I,i}$ ,  $\text{Cmd}_{R,i}$  e  $\text{Cmd}$  comandos, e  $\text{Cond}$  uma condição booleana, temos abaixo a forma geral do comando `for` (os números indicam a ordem de execução das partes do comando `for`):

```
for (CmdI,1, CmdI,2, ... ; Cond; CmdR,1, CmdR,2, ...) Cmd;  
    1           2           3  
    5           4           6  
    8           7           9  
    11          10          12  
    ...         ...  
    n           n-1
```

Observe que, conforme especificado acima, o comando `for` itera somente um comando. Isso nem sempre, ou melhor, quase nunca é satisfatório; é comum desejarmos a iteração de um conjunto não unitário de comandos.

Por isso, no lugar onde se espera a especificação de um comando, podemos especificar um bloco de comandos. Assim, conseguiremos o efeito desejado.

Observe, abaixo, uma ilustração do uso deste comando, que escreve na tela 10 vezes a frase “Java e demais!”:

```
...  
for (int i=1; i<=10; i++)  
    System.out.println ("Java e demais!");  
...
```

equivale a

```
...  
int i;  
  
for (i=1; i<=10; i++)  
    System.out.println ("Java e demais!");  
...
```

equivale a

```
...  
int i=1;
```

---

```
for ( ; i<=10; i++)
    System.out.println ("Java é demais!");
...
```

equivale a

```
...
int i=1;
for ( ; i<=10;)
{
    System.out.println ("Java é demais!");
    i++;
}
...
```

equivale a

```
...
int i=1;
for (;;)
{
    System.out.println ("Java é demais!");
    i++;
    if (i>10) break;
}
...
```

Observe, abaixo, outra ilustração do uso deste comando, que inverte um vetor `vet` de 100 elementos:

```
...
for (int i=0, j=99; i<j; i++, j--)
{
    int temp = vet[i];
    vet[i] = vet[j];
    vet[j] = temp;
}
...
```

Além da sintaxe clássica, há ainda uma outra sintaxe, que, apesar de mais limitada que a clássica, nos casos em que é cabível, representa uma forma elegante de iterar.

Sendo `Tip` um tipo (primitivo ou classe), `Var` uma variável, neste ato, declarada como sendo do tipo `Tip`, `Vet` um vetor (ou uma coleção) de elementos dos tipo `Tip` e `Cmd` o comando a ser repetido (no caso de serem vários, estes devem vir entre chaves `{ }`), segue a forma geral dessa sintaxe alternativa do comando `for`):

---

```
for (Tip Var : Vet) Cmd;
```

A sintaxe do `for` acima especificada tem a semântica de fazer com que a variável `Var` assuma, a cada iteração, um dos valores armazenados em `Vet` e, para cada um desses valores, promover a execução do comando `Cmd`.

Observe, abaixo, uma ilustração do uso deste comando, que escreve na tela todos os valores armazenados no vetor de inteiros `vet`:

```
...
for (int elem : vet)
    System.out.println (elem);
...
```

## O Comando *continue*

O comando `continue` força o reinicio de uma nova iteração nos comandos `while`, `do-while` e `for`. Existem duas variações deste comando: a primeira, simplesmente

```
continue;
```

provoca o reinício imediato de uma nova iteração do comando iterativo, dentro do qual se encontra encaixado mais diretamente; e a segunda,

```
continue Rotulo;
```

provoca o reinício imediato de uma nova iteração do comando iterativo rotulado com `Rotulo` (para rotular um comando, basta preceder-lhe com o identificador do rotulo seguido pelo caractere dois-pontos).

Observe, abaixo, uma ilustração do uso deste comando, que escreve na tela os números de 1 a 7, inclusive, seguidos pelos números de 17 a 20, inclusive:

```
...
int i = 0;
while (i<20)
{
    i++;
    if (i>7 && i<17) continue;
    System.out.println (i);
}
...
```

---

Observe, abaixo, outra ilustração do uso deste comando:

```
...
loop_de_fora: while (cond1)
{
    ...
    while (cond2)
    {
        ...
        if (cond3) continue loop_de_fora;
        ...
    }
    ...
}
```

## O Comando **break**

O comando **break** força a saída imediata dos comandos **while**, **do-while**, **for** e **switch** (veja abaixo os comandos **while**, **do-while** e **for**). Existem duas variações deste comando: a primeira, simplesmente

```
break;
```

provoca a saída imediata do comando **while**, **do-while**, **for** ou **switch**, dentro do qual se encontra encaixado mais diretamente; e a segunda,

```
break Rotulo;
```

provoca a saída imediata do comando **while**, **do-while**, **for** ou **switch** rotulado com **Rotulo** (para rotular um comando, basta preceder-lhe com o identificador do rotulo seguido pelo caractere dois-pontos).

Observe, abaixo, uma ilustração do uso deste comando, que escreve na tela os números de 1 a 7, inclusive:

```
...
int i = 0;
while (i<20)
{
    i++;
    if (i>7 && i<17) break;
    System.out.println (i);
}
...
```

---

Observe, abaixo, outra ilustração do uso deste comando:

```
...
loop_de_fora: while (cond1)
{
    ...
    while (cond2)
    {
        ...
        if (cond3) break loop_de_fora;
        ...
    }
    ...
}
```

[C:\ExplsJava\Expl\_04\Fatorial.java]

```
1: import java.io.IOException;
2: import java.io.BufferedReader;
3: import java.io.InputStreamReader;
4:
5: public class Fatorial
6: {
7:     public static void main (String args []) throws IOException
8:     {
9:         BufferedReader leitor = new BufferedReader
10:            (new InputStreamReader
11:             (System.in)));
12:         int numero;
13:
14:         System.out.println ();
15:         System.out.println ("PROGRAMA PARA CALCULAR FATORIAL");
16:
17:         for (;;)
18:         {
19:             System.out.println ();
20:             System.out.print ("Digite um numero natural: ");
21:             double numeroDigitado = Double.parseDouble
22:                 (leitor.readLine ());
23:
24:             if (numeroDigitado < 0.0)
25:             {
26:                 System.err.print ("Numeros naturais nao ");
27:                 System.err.println ("podem ser negativos!");
28:                 System.err.println ("Tente novamente...");
29:             }
30:             else
31:             {
32:                 numero = (int)numeroDigitado;
33:
34:                 if (numero != numeroDigitado)
35:                 {
36:                     System.err.print ("Numeros naturais nao podem");
37:                     System.err.println ("ter parte fracionaria!");

```

```
38:             System.err.println ("Tente novamente...");  
39:         }  
40:         else  
41:             break;  
42:     }  
43: }  
44:  
45: int fatorial=1;  
46:  
47: while (numero > 1)  
48: {  
49:     fatorial = fatorial*numero;  
50:     numero--;  
51: }  
52:  
53: System.out.println ("Resultado: " + fatorial);  
54: System.out.println ();  
55: }  
56: }
```

Para compilar e executar este programa, daremos os seguintes comandos:

```
C:\ExplsJava\Expl_04> javac *.java  
C:\ExplsJava\Expl_04> java Fatorial
```

Isto poderia produzir no console a seguinte interação:

#### PROGRAMA PARA CALCULAR FATORIAL

```
Digite um numero natural: -2  
Números naturais não podem ser negativos!  
Tente novamente...
```

```
Digite um numero natural: -5.7  
Números naturais não podem ter parte fracionária!  
Tente novamente...
```

```
Digite um numero natural: 5  
Resultado: 120
```

## Membros de Classe e de Instância

Diferenciamos os membros de classe dos membros de instância através de um qualificador. Membros serão de classe se forem precedidos pelo qualificador `static`, e serão de instâncias caso contrário.

Existem 3 formas para acessar membros, a saber:

### **1. Acesso Simples:**

Chamamos de acesso simples aqueles acesso nos quais, para acessar um membro, simplesmente mencionamos o nome do membro. São simples os seguintes acessos:

- Funções de uma classe acessando membros de classe de sua própria classe (neste caso o acesso também pode ser feito mencionando o nome de sua classe, seguido pelo caractere ponto, seguido pelo nome do membro que se deseja acessar);
- Funções de uma instância de classe acessando membros de classe sua própria classe (neste caso o acesso também pode ser feito mencionando o nome de sua classe, seguido pelo caractere ponto, seguido pelo nome do membro que se deseja acessar);
- Funções membro de uma instância de classe acessando membros de instância de classe de sua própria instância de classe (neste caso o acesso também pode ser feito mencionando a palavra chave this, seguida pelo caractere ponto, seguido pelo nome do membro que se deseja acessar).

### **2. Acesso com Nome de Classe:**

Todo acesso não simples a um membro de classe se faz mencionando o nome da classe eu questão, seguido do caractere ponto (.), seguido do nome do membro que se deseja acessar.

### **3. Acesso com Nome de Instância de Classe:**

Todo acesso não simples a um membro de uma instância de classe se faz mencionando o nome do objeto questão, seguido do caractere ponto (.), seguido do nome do membro que se deseja acessar.

## **Construtores de Classe**

O autor de uma classe pode controlar como devem ser iniciados os atributos de classe de uma classe.

---

Apesar da maioria dos atributos poderem ser iniciados no ato de sua declaração, conforme explicado anteriormente, certos atributos necessitam de comandos para a realização de sua iniciação.

Nestes casos, sendo os atributos que necessitam de comandos para sua iniciação atributos de classe, deve-se um método construtor de classe (que se vinculará à classe, ou seja, em cujo cabeçalho o qualificador static estará presente). Este tipo de método não produz jamais retorno (apesar de `void` não ser usado), não possui um nome, não recebe jamais parâmetros e nem mesmo os parênteses que costumeiramente envolvem os parâmetros formais dos métodos é indicada neste tipo de método.

## Exceções

O autor de uma biblioteca pode detectar erros com os quais não consegue lidar. O usuário de uma biblioteca pode querer tratar adequadamente erros que não é capaz de detectar. O conceito de exceção vem para prover meios para resolver este tipo de situação.

A idéia fundamental é que uma função que detecta um problema que não é capaz de tratar lança uma exceção que pode ser pega por uma função que quer tratar o problema mas que não é capaz de detectar.

Uma exceção é um objeto que é uma instância da classe `Throwable` (ou de uma de suas subclasses).

Métodos que podem lançar exceções devem deixar este fato claro através do acréscimo ao final de seu cabeçalho da palavra chave `throws` seguida pelos nomes de todas as exceções possíveis de serem lançadas pelo método separadas por vírgulas (,).

Quando um método chama outro que, por sua vez pode lançar uma exceção, o primeiro deve:

1. Tratar a referida exceção; ou
  2. Avisar através da palavra chave `throws` o possível lançamento da referida exceção.
-

Exceções da classe Error ou RunTimeException (e derivadas) não precisam ser mencionados em um throws porque podem ocorrer em qualquer momento, em qualquer ponto do programa, por qualquer razão.

Sendo, respectivamente Excecao o identificador de uma exceção e p<sub>i</sub> os identificadores dos parâmetros para o contrutor de Excecao , veja abaixo a forma geral do comando que lança uma exceção.

```
throw new Excecao (p1, p2,..., pn);
```

Métodos que fazem uso de métodos que lançam exceções, podem ser implementados de duas maneiras:

**1. NÃO detectando, capturando ou tratando as exceções dos métodos chamados:**

A chamada dos métodos que lançam exceções deve ocorrer da maneira usual, ou seja, da mesma forma que usamos para chamar qualquer outro método. Neste caso, o método chamante deverá deixar o fato de que não detecta, captura ou trata exceções dos métodos chamados através do acréscimo ao final de seu cabeçalho da palavra chave throws seguida pelos nomes de todas as exceções possíveis de serem lançadas pelos métodos chamados separadas por vírgulas (,).

**2. DETECTANDO, CAPTURANDO e TRATANDO as exceções dos métodos chamados:**

Neste caso, nada é acrescentado ao cabeçalho do método chamante.

Já a chamada dos métodos que lançam exceções deve ocorrer da seguinte maneira: primeiro deve vir a palavra chave try; em seguida deve vir, entre chaves ({}), as chamadas dos métodos que podem causar o lançamento de exceções.

O processo de lançar e pegar exceções envolve uma busca retrógrada por um tratador na cadeia de ativações a partir do ponto onde a exceção foi lançada.

A partir do momento que um tratador pega uma exceção já considera-se a exceção tratada, e qualquer outro tratador que possa ser posteriormente encontrado se torna neutralizado.

---

Exceções são pegas pelo comando catch. Um try pode ser sucedido por um número arbitrariamente grande de catches, cada qual com o objetivo de tratar uma das várias exceções que podem ser lançadas dentro do try.

O comando catch tem a seguinte forma geral: primeiro vem a palavra chave catch e, em seguida, entre parênteses (), o nome da exceção que deve ser tratada. Logo após deve vir, entre chaves ({}), o código que trata a referida exceção.

Após todos os catches pode vir um finally. O comando finally tem a seguinte forma geral: primeiro vem a palavra chave finally e, em seguida, vem, entre chaves, o código a ser executado, em qualquer caso, no final do tratamento das exceções (independentemente do fato de em algum tratamento que o antecedesse executar comandos que quebram o fluxo de controle, e.g., return, break ou continue).

Uma classe pode lançar várias exceções, e as funções usuárias podem ou não pegá-las a todas. Exceções não pegadas em um certo nível podem ser pegadas em um nível mais alto.

Tratadores de exceção podem pegar uma exceção mas não conseguir tratá-la completamente.

Neste caso pode ser necessário um novo lançamento de exceção para provocar um tratamento complementar em um nível mais alto.

## Getters e Setters

Já que os atributos de uma classe, por razões de segurança, devem ser sempre privativos, no caso de ser necessário recuperar o valor de um deles, será necessária a escrita de um método com esta específica finalidade. Métodos cujo propósito consista em recuperar o valor de um atributo são chamados de getters e seus nomes, por convenção, devem começar com a palavra get (ou is, caso o atributo a ser recuperado for do tipo boolean), seguido pelo nome do atributo a ser recuperado.

Ainda levando em conta o fato de que os atributos de uma classe, por razões de segurança, como já mencionamos acima, devem ser sempre privativos, no caso de ser necessário

---

ajustar o valor de um deles, será necessária a escrita de um método com esta específica finalidade. Já métodos cujo propósito consista em ajustar o valor de um atributo, naturalmente fazendo as devidas consistências e lançando as devidas exceções, são chamados de setters e seus nomes, por convenção, devem começar com a palavra set, seguido pelo nome do atributo que terá o valor ajustado pelo método.

### [C:\ExplsJava\Expl\_05\Fracao.java]

```
1: public class Fracao
2: {
3:     private static long numerador, denominador;
4:
5:     static
6:     {
7:         Fracao.numerador = 0;
8:         Fracao.denominador = 1;
9:     }
10:
11:    private static void simplifiqueSe ()
12:    {
13:        long menor = Math.min (Math.abs (Fracao.numerador),
14:                               Math.abs (Fracao.denominador));
15:
16:        if (Fracao.numerador %menor == 0 &&
17:            Fracao.denominador%menor == 0)
18:        {
19:            Fracao.numerador = Fracao.numerador /menor;
20:            Fracao.denominador = Fracao.denominador/menor;
21:        }
22:        else
23:            for (int i=2; i<=menor/2; i++)
24:                while (Fracao.numerador %i == 0 &&
25:                      Fracao.denominador%i == 0)
26:                {
27:                    Fracao.numerador = Fracao.numerador /i;
28:                    Fracao.denominador = Fracao.denominador/i;
29:                }
30:    }
31:
32:    public static long getNumerador ()
33:    {
34:        return Fracao.numerador;
35:    }
36:
37:    public static long getDenominador ()
38:    {
39:        return Fracao.denominador;
40:    }
41:
42:    public static void setNumerador (long numerador)
43:    {
```

```
44:         Fracao.numerador = numerador;
45:         Fracao.simplifiqueSe ();
46:     }
47:
48:     public static void setDenominador (long denominador) throws Exception
49:     {
50:         if (denominador==0)
51:             throw new Exception ("Denominador zero");
52:
53:         Fracao.denominador = denominador;
54:         Fracao.simplifiqueSe ();
55:     }
56:
57:     public static void someSeCom (long numero)
58:     {
59:         Fracao.numerador = Fracao.numerador +
60:                             Fracao.denominador * numero;
61:
62:     }
63:
64:     public static void subtraiaDeSi (long numero)
65:     {
66:         Fracao.numerador = Fracao.numerador -
67:                             Fracao.denominador * numero;
68:     }
69:
70:     public static void multipliqueSePor (long numero)
71:     {
72:         Fracao.numerador = Fracao.numerador * numero;
73:     }
74:
75:     public static void dividaSePor (long numero) throws Exception
76:     {
77:         if (numero == 0)
78:             throw new Exception ("Divisao por zero");
79:
80:         Fracao.denominador = Fracao.denominador * numero;
81:
82:         if (Fracao.denominador < 0)
83:         {
84:             Fracao.numerador = -Fracao.numerador;
85:             Fracao.denominador = -Fracao.denominador;
86:         }
87:     }
88: }
```

[C:\ExplsJava\Expl\_05\TesteDeFracao.java]

```
1: public class TesteDeFracao
2: {
3:     public static void main (String args [])
4:     {
5:         try
```



```
62:         }
63:     catch (Exception e)
64:     {
65:         System.err.println (e.getMessage ());
66:     }
67: }
68: }
```

Para compilar e executar este programa, daremos os seguintes comandos:

```
C:\ExplsJava\Expl_05> javac *.java
C:\ExplsJava\Expl_05> java TesteDeFracao
```

Isto produziria no console a seguinte interação:

```
Fracao vale, inicialmente, 0/1
```

```
1/2 + 7 = 15/2
15/2 - 7 = 1/2
1/2 * 7 = 7/2
7/2 / 7 = 7/14
```

## Criação de Instâncias

A declaração de uma variável de uma classe (objeto) não faz com que, automaticamente, seja criada uma instância da classe para ser armazenada no objeto declarado. Isso somente é conseguido com o operador new.

O operador new é um operador que opera sobre uma classe, criando uma instância da mesma. Sendo Classe o identificador de uma classe, temos que a forma geral de uma expressão de criação de uma instância é a seguinte:

```
new Classe ()
```

A expressão de criação de uma instância pode ser usada em qualquer lugar onde um objeto da classe Classe é esperado.

---

## Construtores de Instância

O autor de uma classe pode controlar o que deve acontecer por ocasião da criação e da destruição de instâncias da classe escrevendo construtores que servem, basicamente, para iniciar os membros da instância que está sendo criada.

Construtores devem ter mesmo nome da classe à qual se referem, não produzem retorno (apesar de void não ser usado) e, quando existem, são ativados automaticamente logo após o operador new promover a criação de uma instância; o operador new aloca memória para a instância e o construtor é invocado para realizar as iniciações programadas.

Não há problema algum em construtores receberem parâmetros e, já que são ativados automaticamente, os parâmetros reais são passados entre os parênteses que normalmente são encontrados na sintaxe do operador new.

## Métodos Canônicos

Trata-se de métodos que devem obrigatoriamente serem escritos para qualquer classe (no caso dos 3 primeiros) ou então que devem ser escritos apenas para classes com certas propriedades (no caso dos 3 últimos). São eles:

### ***toString***

Trata-se de um método padrão que se destina a produzir uma versão da instância à qual for aplicado o método em uma instância da classe String. A invocação deste método pode ficar subentendida; o método `toString` é chamado sempre que uma instância for concatenada com uma instância da classe String ou então for usada num local onde se espera uma instância da classe String.

### ***equals***

Trata-se de um método padrão que deve resultar verdadeiro, sempre que a instância à qual for aplicado puder ser considerada igual à instância da classe Object (que pode, efetivamente, ser uma instância de qualquer classe, já que instâncias de qualquer classe

---

podem ser consideradas instâncias da classe Object) fornecida como parâmetro, ou falso, caso contrário.

Naturalmente, para que possam ser consideradas iguais a instância à qual o método for aplicado e a instância fornecida como parâmetro, ambas deverão ser instâncias da mesma classe e isto deverá ser testado através do operador instanceof, antes de fazer uma conversão de tipo da instância da classe Object fornecida como parâmetro para a classe à qual o método for aplicado, de forma a poder testar a igualdade, agora levando em conta os atributos das duas instâncias que, por serem da mesma classe, serão os mesmos.

### **hashCode**

Resulta um código de espalhamento (de hash) que será um número inteiro e que será utilizado em benefício de tabelas de espalhamento (tabelas de hash ou, simplesmente, hash tables). O referido código de espalhamento funcionará como um identificador da instância à qual o método for aplicado.

Este método deverá resultar códigos de espalhamento numericamente idênticos, sempre que for invocado para instâncias que puderem ser consideradas iguais pelo método equals. Não se exige que sejam numericamente os mesmos em duas execuções diferentes da mesma aplicação, embora, em geral, o sejam.

Em outras palavras, se duas instâncias puderem ser consideradas iguais pela aplicação do método equals, então este método aplicado a elas deverá necessariamente produzir o mesmo inteiro como resultado.

Não é necessário que este método resulte inteiros numericamente diferentes quando aplicado a duas instâncias que não são consideradas iguais pelo método equals. No entanto deve-se estar consciente de que a produção de valores inteiros distintos para instâncias que são consideradas distintas pelo método equals, causa impactos positivos ao desempenho de tabelas de espalhamento.

Segue uma receita simples para calcular o código de espalhamento de uma instância:

---

1. Armazene `super.hashCode()` em uma variável do tipo `int` qualquer, digamos chamada `ret`;
2. Para cada atributo significativo `atr` da instância (cada atributo levado em conta pelo método `equals`), levando em conta o tipo do atributo, calcule o código de espalhamento `cē` do atributo, conforme segue:
  - i. Se o atributo for do tipo `byte`, calcule `new Byte (atr).hashCode()`.
  - ii. Se o atributo for do tipo `short`, calcule `new Short (atr).hashCode()`.
  - iii. Se o atributo for do tipo `int`, calcule `new Integer (atr).hashCode()`.
  - iv. Se o atributo for do tipo `long`, calcule `new Long (atr).hashCode()`.
  - v. Se o atributo for do tipo `float`, calcule `new Float (atr).hashCode()`.
  - vi. Se o atributo for do tipo `double`, calcule `new Double (atr).hashCode()`.
  - vii. Se o atributo for do tipo `boolean`, calcule `new Boolean (atr).hashCode()`.
  - viii. Se o atributo for do tipo `char`, calcule `new Character (atr).hashCode()`.
  - ix. Se o `atr` for um objeto de uma certa classe, caso `atr` seja `null`, use 0 (zero), e caso não seja `null`, calcule `atr.hashCode()`.
  - x. Se o `atr` for um vetor com qualquer quantidade de dimensões, trate cada uma de suas posições significativas como se fossem atributos separados, ou seja, calcule um código de espalhamento para cada elemento significativo aplicando, para cada elemento, estas mesmas regras e os combine como descrito no passo b abaixo.

Especificamente no caso de vetores unidimensionais (a) com a quantidade de posições significativas coincidente com a quantidade de posições que

---

fisicamente existem no vetor; ou (b) que armazenam objetos de uma certa classe e cujas posições não significativas contenham null, pode-se calcular `Arrays.hashCode (atr)`.

3. Combine os código de espalhamento `ce` calculados no passo 2 para cada atributo, somando-o ao resultado acumulado multiplicado por um número primo qualquer, por exemplo, 13. Em outras palavras, para cada código de espalhamento `ce` calculado, faça:

```
ret = 13*ret + ce;
```

4. Retorne `ret`.
5. Tendo terminado de escrever o método `hashcode`, faça testes para verificar se ele retorna o mesmo valor para instâncias consideradas iguais pelo método `equals`, realizando correções no método se necessário.

### ***compareTo***

Sempre que uma classe C representar algo para o qual exista uma relação de ordem ou, em outras palavras, sempre que uma classe produzir instâncias que puderem ser comparadas entre si, não apenas para verificar se são iguais ou não (missão do método `equals`), mas também para verificar se uma delas é menor do que a outra ou se é maior, deve-se fazer com que a referida classe implemente `Comparable<C>`, indicando que instâncias da classe C podem ser comparadas outras instâncias, também da classe C.

Para tanto, sendo C o nome da classe em questão, quando a estivermos implementando, ao escrevermos `class C`, basta acrescentarmos `implements Comparable<C>`, além de implementar um método chamado `compareTo` que, compara a instância à qual o método se aplica e a instância da mesma classe fornecida como parâmetro, resultando um número inteiro negativo, caso a primeira seja menor do que a segunda, zero, caso ambas sejam iguais, ou um número inteiro positivo, caso a primeira seja maior que a segunda.

---

## **clone**

Sempre que uma classe contiver algum método que altera um ou mais atributos dos objetos da classe (setters são exemplos típicos, mas não são os únicos), deve-se fazer com que a referida classe implemente `Cloneable`, indicando que instâncias daquela classe podem ser clonadas, além de implementar um método chamado `clone`, sem parâmetros e que retorna um `Object` que representa uma cópia da instância à qual o método for aplicado que funcionará como modelo.

No caso do modelo possuir algum atributo que seja um objeto clonável ou um vetor (com qualquer quantidade de dimensões), clones deles deverão ser atribuídos aos atributos correspondentes da cópia.

Finalmente, caso o modelo possua algum atributo seja um objeto não clonável, ele, a princípio deverá ser simplesmente atribuído ao atributo correspondente da cópia. Mas atenção, isso somente deve ser feito quando o atributo em questão for um objeto de uma classe não clonável que não merece ser clonada; quando essa falta de merecimento não verificar, a situação deve ser denunciada ao responsável pelo desenvolvimento.

## **Construtor de Cópia**

Sempre atrelado ao método `clone`, ou seja, presente, quando o método `clone` estiver presente ou ausente, quando o método `clone` estiver ausente, o construtor de cópia nada mais é do que um construtor que recebe como parâmetro para servir como modelo um objeto da mesma classe das instâncias por ele construídas, que, a propósito, devem ser construídas de forma a se tornarem uma cópia do modelo fornecido.

É comum que o método `clone` abordado acima seja implementado fazendo uso dele parametrizado com `this`.

**[C:\ExplJava\Expl\_06\Fracao.java]**

```
1: public class Fracao implements Comparable<Fracao>, Cloneable
2: {
3:     private long numerador, denominador;
4:
5:     private double valorReal ()
```

```
6:      {
7:          return (double)numerador /
8:                  (double)denominador;
9:      }
10:
11:     private void simplifiqueSe ()
12:     {
13:         long menor = Math.min (Math.abs (this.numerador),
14:                               Math.abs (this.denominador));
15:
16:         if (this.numerador%this.denominador == 0)
17:         {
18:             this.numerador = this.numerador / this.denominador;
19:             this.denominador = 1;
20:         }
21:         else
22:             if (this.numerador %menor == 0 &&
23:                 this.denominador%menor == 0)
24:             {
25:                 this.numerador = this.numerador /menor;
26:                 this.denominador = this.denominador/menor;
27:             }
28:         else
29:             for (long metade=menor/2, i=2; i<=metade; i++)
30:                 while (this.numerador %i == 0 &&
31:                        this.denominador%i == 0)
32:                 {
33:                     this.numerador = this.numerador /i;
34:                     this.denominador = this.denominador/i;
35:                 }
36:     }
37:
38:     public Fracao (long numerador,
39:                    long denominador)
40:                    throws Exception
41:     {
42:         if (denominador == 0)
43:             throw new Exception ("Denominador zero");
44:
45:         if (denominador < 0)
46:         {
47:             this.numerador = -numerador;
48:             this.denominador = -denominador;
49:         }
50:         else
51:         {
52:             this.numerador = numerador;
53:             this.denominador = denominador;
54:         }
55:
56:         this.simplifiqueSe ();
57:     }
58:
59:     public long getNumerador ()
60:     {
61:         return this.numerador;
```

```
62: }
63:
64:     public long getDenominador ()
65:     {
66:         return this.denominador;
67:     }
68:
69:     public void setNumerador (long numerador)
70:     {
71:         this.numerador = numerador;
72:         this.simplifiqueSe ();
73:     }
74:
75:     public void setDenominador (long denominador) throws Exception
76:     {
77:         if (denominador==0)
78:             throw new Exception ("Denominador zero");
79:
80:         this.denominador = denominador;
81:         this.simplifiqueSe ();
82:     }
83:
84:     public Fracao mais (Fracao fracao) throws Exception
85:     {
86:         if (fracao == null)
87:             throw new Exception ("Falta de operando em soma");
88:
89:         long numerador    = this.numerador * fracao.denominador +
90:                           this.denominador * fracao.numerador,
91:
92:         denominador = this.denominador * fracao.denominador;
93:
94:         Fracao resultado = new Fracao (numerador,denominador);
95:
96:         resultado.simplifiqueSe ();
97:         return resultado;
98:     }
99:
100:    public Fracao menos (Fracao fracao) throws Exception
101:    {
102:        if (fracao == null)
103:            throw new Exception ("Falta de operando em soma");
104:
105:        long numerador    = this.numerador * fracao.denominador -
106:                           this.denominador * fracao.numerador,
107:
108:        denominador = this.denominador * fracao.denominador;
109:
110:        Fracao resultado = new Fracao (numerador, denominador);
111:
112:        resultado.simplifiqueSe ();
113:        return resultado;
114:    }
115:
116:    public Fracao vezes (Fracao fracao) throws Exception
117:    {
```

```
118:         if (fracao == null)
119:             throw new Exception ("Falta de operando em multiplicacao");
120:
121:         long numerador    = this.numerador * fracao.numerador,
122:             denominador = this.denominador * fracao.denominador;
123:
124:         Fracao resultado = new Fracao (numerador,denominador);
125:
126:         resultado.simplifiqueSe ();
127:         return resultado;
128:     }
129:
130:     public Fracao divididoPor (Fracao fracao) throws Exception
131:     {
132:         if (fracao == null)
133:             throw new Exception ("Falta de operando em divisao");
134:
135:         if (fracao.numerador == 0)
136:             throw new Exception ("Divisao por zero");
137:
138:         long numerador    = this.numerador * fracao.denominador,
139:             denominador = this.denominador * fracao.numerador;
140:
141:         if (denominador < 0)
142:         {
143:             numerador    = -numerador;
144:             denominador = -denominador;
145:         }
146:
147:         Fracao resultado = new Fracao (numerador,denominador);
148:
149:         resultado.simplifiqueSe ();
150:         return resultado;
151:     }
152:
153:     public String toString ()
154:     {
155:         if (this.numerador == this.denominador)
156:             return "1";
157:
158:         if (this.numerador + this.denominador == 0)
159:             return "-1";
160:
161:         if (this.numerador == 0 || this.denominador == 1)
162:             return "" + this.numerador;
163:
164:         return this.numerador + "/" + this.denominador;
165:     }
166:
167:     public boolean equals (Object obj)
168:     {
169:         if (this == obj)
170:             return true;
171:
172:         if (obj == null)
173:             return false;
```

```
174:
175:         //if (!obj instanceof Fracao)
176:         if (this.getClass() != obj.getClass())
177:             return false; // poderíamos fazer de outra forma
178:             // caso quisessemos compatibilizar
179:             // fracos com outras classes, e.g.,
180:             // com strings; naturalmente, isso
181:             // causaria impacto no hashCode
182:
183:         Fracao fracao = (Fracao)obj;
184:
185:         if (this.valorReal() != fracao.valorReal())
186:             return false;
187:
188:         return true;
189:     }
190:
191:     public int hashCode ()
192:     {
193:         int ret = super.hashCode();
194:
195:         ret = 13*ret +
196:             new Long(numerador).hashCode();
197:
198:         ret = 13*ret +
199:             new Long(denominador).hashCode();
200:
201:         return ret;
202:     }
203:
204:     public int compareTo (Fracao fracao)
205:     {
206:         double vrThis    = this .valorReal(),
207:             vrFracao = fracao.valorReal();
208:
209:         if (vrThis < vrFracao)
210:             return -1;
211:         else
212:             if (vrThis == vrFracao)
213:                 return 0;
214:             else
215:                 return 1;
216:     }
217:
218:     public Fracao (Fracao modelo)
219:             throws Exception
220:     {
221:         if (modelo==null)
222:             throw new Exception ("Modelo não fornecido");
223:
224:         this.numerador    = modelo.numerador;
225:         this.denominador = modelo.denominador;
226:     }
227:
228:     public Object clone ()
229:     {
```

```
230:         Fracao copia=null;
231:
232:         try
233:         {
234:             copia = new Fracao (this);
235:         }
236:         catch (Exception e)
237:         { }
238:
239:         return copia;
240:     }
241: }
```

[C:\ExplsJava\Expl\_06\TesteDeFracao.java]

```
1: public class TesteDeFracao
2: {
3:     public static void main (String[] args)
4:     {
5:         Fracao f1, f2, f3, f4, f5;
6:
7:         try
8:         {
9:             f2 = new Fracao (5,7);
10:
11:             System.out.println ("Numerador de " + f2 + " = " +
12:                                 f2.getNumerador());
13:
14:             System.out.println ("Denominador de " + f2 + " = " +
15:                                 f2.getDenominador());
16:
17:             System.out.println ();
18:
19:             f3 = f2;
20:             f4 = (Fracao)f2.clone();
21:             f5 = (Fracao)f2.clone();
22:
23:             System.out.println ("Alterando numerador e denominador, ");
24:             System.out.println ("respectivamente para 1 e 2, ");
25:             System.out.print (f2 + " torna-se ");
26:
27:             f2.setNumerador (1);
28:             f2.setDenominador (2);
29:
30:             System.out.println (f2);
31:             System.out.println ();
32:
33:             System.out.println ("Observe que a mudanca acima");
34:             System.out.println ("tambem afeta " + f3);
35:             System.out.println ("mas nao " + f4);
36:
37:             System.out.println ();
38: }
```





```
151:         System.out.println ();
152:
153:         System.out.println ("O codigo de espalhamento de " +
154:                             "uma instancia valendo " + f1 +
155:                             " vale " + f1.hashCode ());
156:
157:         System.out.println ("O codigo de espalhamento de " +
158:                             "uma instancia valendo " + f2 +
159:                             " vale " + f2.hashCode ());
160:
161:         System.out.println ("O codigo de espalhamento de " +
162:                             "outra instancia valendo " + f3 +
163:                             " vale " + f3.hashCode ());
164:
165:         System.out.println ("O codigo de espalhamento de " +
166:                             "uma instancia valendo " + f4 +
167:                             " vale " + f4.hashCode ());
168:
169:         System.out.println ("O codigo de espalhamento de " +
170:                             "outra instancia valendo " + f5 +
171:                             " vale " + f5.hashCode ());
172:     }
173:     catch (Exception e)
174:     {
175:         System.err.println (e.getMessage ());
176:     }
177: }
178: }
```

Para compilar e executar este programa, daremos os seguintes comandos:

```
C:\ExplsJava\Expl_06> javac *.java
C:\ExplsJava\Expl_06> java TesteDeFracao
```

Isto produziria no console a seguinte interação:

Numerador de 5/7 = 5  
Denominador de 5/7 = 7

Alterando numerador e denominador,  
respectivamente para 1 e 2,  
5/7 torna-se 1/2

Observe que a mudança acima  
também afeta 1/2  
mas não 5/7

$17/14 = 1/2 + 5/7$   
 $-3/14 = 1/2 - 5/7$   
 $5/14 = 1/2 * 5/7$

$7/10 = 1/2 / 5/7$

$1/2 == 1/2$  (pelo ==)  
 $1/2 == 1/2$  (pelo equals)  
 $1/2 != 5/7$  (pelo ==)  
 $1/2 != 5/7$  (pelo equals)  
 $5/7 != 5/7$  (pelo ==)  
 $5/7 == 5/7$  (pelo equals)

$1/2 == 1/2$  (pelo compareTo)  
 $1/2 == 1/2$  (pelo compareTo)  
 $1/2 < 5/7$  (pelo compareTo)  
 $5/7 > 1/2$  (pelo compareTo)  
 $5/7 == 5/7$  (pelo compareTo)  
 $5/7 == 5/7$  (pelo compareTo)

O código de espalhamento de uma instância valendo  $7/10$  vale 1188

O código de espalhamento de uma instância valendo  $1/2$  vale 994

O código de espalhamento de outra instância valendo  $1/2$  vale 994

O código de espalhamento de uma instância valendo  $5/7$  vale 1123

O código de espalhamento de outra instância valendo  $5/7$  vale 1123

## Organização de Programa

### Modularização

Embora não seja obrigatório partitionar os programas em módulos, recomenda-se fortemente que isso seja feito, especialmente para programas de um porte maior. Isso fará com que o programa se torne mais manutenível, aumenta as possibilidades de reuso, além de minimizar a perda de tempo com compilações, já que módulos são unidades de compilação separadas.

Em Java, módulos são arquivos com uma coleção de classes. Recomenda-se colocar em um mesmo módulo somente classes afins. É importante ressaltar que, se diversas classes forem escritas em um mesmo arquivo fonte, somente poderá ser utilizada por outras classes aquela que tiver o mesmo nome do arquivo fonte, se houver uma classe nessas condições, e retirada a extensão .java, naturalmente. As demais poderão apenas serem utilizadas por suas companheiras com quem compartilha o arquivo fonte.

---

## Pacotes

Como vimos, um programa em Java pode se compor de diversos arquivos. Podem também usar arquivos de dados, imagens, sons, etc. Projetos maiores podem de fato ficar muito desorganizados se todos estes arquivos residirem juntos em um mesmo diretório.

Para resolver este problema, introduzimos o conceito de pacote. Pacotes definem uma hierarquia de espaços onde as classes podem ser armazenadas. Todo pacote pode ter zero ou mais subpacotes, todo subpacote pode ter zero ou mais subsubpacotes, e assim por diante. O compilador Java usa diretórios do sistema de arquivos para armazenar pacotes.

É importante ressaltar o fato de que deve existir uma correspondência biunívoca entre a hierarquia de pacotes e a hierarquia de diretórios que fica abaixo do diretório raiz do diretório onde está armazenado o programa.

É preciso também chamar a atenção para o fato de que as letras minúsculas e maiúsculas que constituem os nomes dos diretórios devem corresponder exatamente às letras que constituem o nome dos pacotes.

Arquivos fonte de classes que residem em um pacote devem começar pelo comando:

```
package NomPac1[NomPac2[...]];
```

onde NomPac1 é o nome do pacote onde reside a classe em questão, NomPac2 é o nome do subpacote do pacote NomPac1 onde ela reside, e assim sucessivamente.

### Classes Públicas

Classes que residem dentro de um mesmo nó da hierarquia de pacotes podem empregar-se mutuamente em suas implementações, ao passo que classes que residem em um determinado nó da hierarquia de pacotes somente podem fazer uso de classes que residem em um outro nó da hierarquia de pacotes se estas últimas forem qualificadas com o qualificador public.

### Membros Default (ou de Pacote)

Conforme sabemos, existem 2 tipos de membros que podem ser definidos em uma classe, a saber: os membros públicos (que são acessíveis para qualquer método que tenha acesso à

---

classe em que foram definidos) e os membros privativos (que são acessíveis somente nos métodos definidos em sua classe).

Ficaremos sabendo agora da existência de um terceiro tipo de membro, os membros default (ou de pacote). Esse tipo de membro é declarado sem nenhuma qualificação, i.e., sem serem precedidos pela palavra public, pela palavra private, ou por qualquer outra palavra especialmente definida para qualifica-los.

Membros default (ou de pacote) são acessíveis para todas os métodos definidos nas classes que integram o pacote onde foi definida sua classe.

Convém ressaltar que, quando não dividimos classes em pacotes, Java coloca todas as classes que definimos em um único pacote.

### [C:\ExplJava\Expl\_07\matematica\Fracao.java]

```
1: package matematica;
2:
3: // se a palavra public abaixo fosse retirada, esta classe
4: // não poderia ser utilizada na main; isto porque classes
5: // não públicas somente podem ser utilizadas por outras
6: // classes dentro da mesma pasta
7:
8: public class Fracao implements Comparable<Fracao>, Cloneable
9: {
10:     private long numerador, denominador;
11:
12:     private double valorReal ()
13:     {
14:         return (double)numerador /
15:             (double)denominador;
16:     }
17:
18:     private void simplifiqueSe ()
19:     {
20:         long menor = Math.min (Math.abs (this.numerador),
21:                               Math.abs (this.denominador));
22:
23:         if (this.numerador%this.denominador == 0)
24:         {
25:             this.numerador = this.numerador / this.denominador;
26:             this.denominador = 1;
27:         }
28:         else
29:             if (this.numerador %menor == 0 &&
30:                 this.denominador%menor == 0)
31:             {
32:                 this.numerador = this.numerador /menor;
```

```
33:             this.denominador = this.denominador/menor;
34:         }
35:     else
36:         for (long metade=menor/2, i=2; i<=metade; i++)
37:             while (this.numerador %i == 0 &&
38:                   this.denominador%i == 0)
39:             {
40:                 this.numerador = this.numerador /i;
41:                 this.denominador = this.denominador/i;
42:             }
43:     }
44:
45:     public Fracao (long numerador,
46:                     long denominador)
47:                     throws Exception
48:     {
49:         if (denominador == 0)
50:             throw new Exception ("Denominador zero");
51:
52:         if (denominador < 0)
53:         {
54:             this.numerador = -numerador;
55:             this.denominador = -denominador;
56:         }
57:         else
58:         {
59:             this.numerador = numerador;
60:             this.denominador = denominador;
61:         }
62:
63:         this.simplifiqueSe ();
64:     }
65:
66:     public long getNumerador ()
67:     {
68:         return this.numerador;
69:     }
70:
71:     public long getDenominador ()
72:     {
73:         return this.denominador;
74:     }
75:
76:     public void setNumerador (long numerador)
77:     {
78:         this.numerador = numerador;
79:         this.simplifiqueSe ();
80:     }
81:
82:     public void setDenominador (long denominador) throws Exception
83:     {
84:         if (denominador==0)
85:             throw new Exception ("Denominador zero");
86:
87:         this.denominador = denominador;
88:         this.simplifiqueSe ();
```

```
89:     }
90:
91:     public Fracao mais (Fracao fracao) throws Exception
92:     {
93:         if (fracao == null)
94:             throw new Exception ("Falta de operando em soma");
95:
96:         long numerador = this.numerador * fracao.denominador +
97:                         this.denominador * fracao.numerador,
98:
99:         denominador = this.denominador * fracao.denominador;
100:
101:        Fracao resultado = new Fracao (numerador,denominador);
102:
103:        resultado.simplifiqueSe ();
104:        return resultado;
105:    }
106:
107:    public Fracao menos (Fracao fracao) throws Exception
108:    {
109:        if (fracao == null)
110:            throw new Exception ("Falta de operando em soma");
111:
112:        long numerador = this.numerador * fracao.denominador -
113:                        this.denominador * fracao.numerador,
114:
115:        denominador = this.denominador * fracao.denominador;
116:
117:        Fracao resultado = new Fracao (numerador, denominador);
118:
119:        resultado.simplifiqueSe ();
120:        return resultado;
121:    }
122:
123:    public Fracao vezes (Fracao fracao) throws Exception
124:    {
125:        if (fracao == null)
126:            throw new Exception ("Falta de operando em multiplicacao");
127:
128:        long numerador = this.numerador * fracao.numerador,
129:                denominador = this.denominador * fracao.denominador;
130:
131:        Fracao resultado = new Fracao (numerador,denominador);
132:
133:        resultado.simplifiqueSe ();
134:        return resultado;
135:    }
136:
137:    public Fracao divididoPor (Fracao fracao) throws Exception
138:    {
139:        if (fracao == null)
140:            throw new Exception ("Falta de operando em divisao");
141:
142:        if (fracao.numerador == 0)
143:            throw new Exception ("Divisao por zero");
144:
```

```
145:         long numerador = this.numerador * fracao.denominador,
146:         denominador = this.denominador * fracao.numerador;
147:
148:         if (denominador < 0)
149:         {
150:             numerador = -numerador;
151:             denominador = -denominador;
152:         }
153:
154:         Fracao resultado = new Fracao (numerador,denominador);
155:
156:         resultado.simplifiqueSe ();
157:         return resultado;
158:     }
159:
160:     public String toString ()
161:     {
162:         if (this.numerador == this.denominador)
163:             return "1";
164:
165:         if (this.numerador + this.denominador == 0)
166:             return "-1";
167:
168:         if (this.numerador == 0 || this.denominador == 1)
169:             return "" + this.numerador;
170:
171:         return this.numerador + "/" + this.denominador;
172:     }
173:
174:     public boolean equals (Object obj)
175:     {
176:         if (this == obj)
177:             return true;
178:
179:         if (obj == null)
180:             return false;
181:
182:         //if (!obj instanceof Fracao)
183:         if (this.getClass() != obj.getClass())
184:             return false; // poderíamos fazer de outra forma
185:                         // caso quisessemos compatibilizar
186:                         // fracos com outras classes, e.g.,
187:                         // com strings; naturalmente, isso
188:                         // causaria impacto no hashCode
189:
190:         Fracao fracao = (Fracao)obj;
191:
192:         if (this.valorReal() != fracao.valorReal())
193:             return false;
194:
195:         return true;
196:     }
197:
198:     public int hashCode ()
199:     {
200:         int ret = super.hashCode();
```

```
201:
202:         ret = 13*ret +
203:                 new Long(numerador).hashCode();
204:
205:         ret = 13*ret +
206:                 new Long(denominador).hashCode();
207:
208:         return ret;
209:     }
210:
211:     public int compareTo (Fracao fracao)
212:     {
213:         double vrThis    = this .valorReal(),
214:             vrFracao = fracao.valorReal();
215:
216:         if (vrThis < vrFracao)
217:             return -1;
218:         else
219:             if (vrThis == vrFracao)
220:                 return 0;
221:             else
222:                 return 1;
223:     }
224:
225:     public Fracao (Fracao modelo)
226:             throws Exception
227:     {
228:         if (modelo==null)
229:             throw new Exception ("Modelo nao fornecido");
230:
231:         this.numerador    = modelo.numerador;
232:         this.denominador = modelo.denominador;
233:     }
234:
235:     public Object clone ()
236:     {
237:         Fracao copia=null;
238:
239:         try
240:         {
241:             copia = new Fracao (this);
242:         }
243:         catch (Exception e)
244:         {}
245:
246:         return copia;
247:     }
248: }
```

[C:\ExplsJava\Expl\_07\TesteDeFracao.java]

```
1: import matematica.*;
2:
3: public class TesteDeFracao
4: {
```

```
5:     public static void main (String[] args)
6:     {
7:         Fracao f1, f2, f3, f4, f5;
8:
9:         try
10:        {
11:            f2 = new Fracao (5,7);
12:
13:            System.out.println ("Numerador de " + f2 + " = " +
14:                                f2.getNumerador());
15:
16:            System.out.println ("Denominador de " + f2 + " = " +
17:                                f2.getDenominador());
18:
19:            System.out.println ();
20:
21:            f3 = f2;
22:            f4 = (Fracao)f2.clone();
23:            f5 = (Fracao)f2.clone();
24:
25:            System.out.println ("Alterando numerador e denominador, ");
26:            System.out.println ("respectivamente para 1 e 2, ");
27:            System.out.print (f2 + " torna-se ");
28:
29:            f2.setNumerador (1);
30:            f2.setDenominador (2);
31:
32:            System.out.println (f2);
33:            System.out.println ();
34:
35:            System.out.println ("Observe que a mudanca acima");
36:            System.out.println ("tambem afeta " + f3);
37:            System.out.println ("mas nao " + f4);
38:
39:            System.out.println ();
40:
41:            f1 = f2.mais(f4);
42:            System.out.println (f1 + " = " + f2 + " + " + f4);
43:
44:            f1 = f2.menos(f4);
45:            System.out.println (f1 + " = " + f2 + " - " + f4);
46:
47:            f1 = f2.vezes(f4);
48:            System.out.println (f1 + " = " + f2 + " * " + f4);
49:
50:            f1 = f2.divididoPor(f4);
51:            System.out.println (f1 + " = " + f2 + " / " + f4);
52:
53:            System.out.println ();
54:
55:            if (f2==f3)
56:                System.out.println (f2 + " == " + f3 + " (pelo ==)");
57:            else
58:                System.out.println (f2 + " != " + f3 + " (pelo ==)");
59:
60:            if (f2.equals(f3))
```

```
61:             System.out.println (f2 + " == " + f3 + " (pelo equals)");
62:         else
63:             System.out.println (f2 + " != " + f3 + " (pelo equals)");
64:
65:         if (f3==f4)
66:             System.out.println (f3 + " == " + f4 + " (pelo ==)");
67:         else
68:             System.out.println (f3 + " != " + f4 + " (pelo ==)");
69:
70:         if (f3.equals(f4))
71:             System.out.println (f3 + " == " + f4 + " (pelo equals)");
72:         else
73:             System.out.println (f3 + " != " + f4 + " (pelo equals)");
74:
75:         if (f4==f5)
76:             System.out.println (f4 + " == " + f5 + " (pelo ==)");
77:         else
78:             System.out.println (f4 + " != " + f5 + " (pelo ==)");
79:
80:         if (f4.equals(f5))
81:             System.out.println (f4 + " == " + f5 + " (pelo equals)");
82:         else
83:             System.out.println (f4 + " != " + f5 + " (pelo equals)");
84:
85:         System.out.println ();
86:
87:         int comp = f2.compareTo(f3);
88:         if (comp < 0)
89:             System.out.println (f2 + " < " + f3 + " (pelo compareTo)");
90:         else
91:             if (comp == 0)
92:                 System.out.println (f2 + " == " + f3 +
93:                                     " (pelo compareTo)");
94:             else
95:                 System.out.println (f2 + " > " + f3 +
96:                                     " (pelo compareTo)");
97:
98:         comp = f3.compareTo(f2);
99:         if (comp < 0)
100:             System.out.println (f3 + " < " + f2 + " (pelo compareTo)");
101:         else
102:             if (comp == 0)
103:                 System.out.println (f3 + " == " + f2 +
104:                                     " (pelo compareTo)");
105:             else
106:                 System.out.println (f3 + " > " + f2 +
107:                                     " (pelo compareTo)");
108:
109:         comp = f3.compareTo(f4);
110:         if (comp < 0)
111:             System.out.println (f3 + " < " + f4 + " (pelo compareTo)");
112:         else
113:             if (comp == 0)
114:                 System.out.println (f3 + " == " + f4 +
115:                                     " (pelo compareTo)");
116:             else
```



```
173:                     " vale " + f5.hashCode());
174:     }
175:     catch (Exception e)
176:     {
177:         System.err.println (e.getMessage ());
178:     }
179: }
180: }
```

Para compilar e executar este programa, daremos os seguintes comandos:

```
C:\ExplsJava\Expl_07> javac *.java
C:\ExplsJava\Expl_07> java TesteDeFracao
```

Isto poderia produzir no console a seguinte interação:

```
Numerador de 5/7 = 5
Denominador de 5/7 = 7
```

```
Alterando numerador e denominador,
respectivamente para 1 e 2,
5/7 torna-se 1/2
```

```
Observe que a mudança acima
também afeta 1/2
mas não 5/7
```

```
17/14 = 1/2 + 5/7
-3/14 = 1/2 - 5/7
5/14 = 1/2 * 5/7
7/10 = 1/2 / 5/7
```

```
1/2 == 1/2 (pelo ==)
1/2 == 1/2 (pelo equals)
1/2 != 5/7 (pelo ==)
1/2 != 5/7 (pelo equals)
5/7 != 5/7 (pelo ==)
5/7 == 5/7 (pelo equals)
```

```
1/2 == 1/2 (pelo compareTo)
1/2 == 1/2 (pelo compareTo)
1/2 < 5/7 (pelo compareTo)
5/7 > 1/2 (pelo compareTo)
5/7 == 5/7 (pelo compareTo)
5/7 == 5/7 (pelo compareTo)
```

O código de espalhamento de uma instância valendo 7/10 vale 1188

---

O código de espalhamento de uma instância valendo 1/2 vale 994  
O código de espalhamento de outra instância valendo 1/2 vale 994  
O código de espalhamento de uma instância valendo 5/7 vale 1123  
O código de espalhamento de outra instância valendo 5/7 vale 1123

## javadoc

A ferramenta javadoc extrai certo tipo de comentários que escrevemos ao programar e gera páginas HTML de documentação. Presta-se a gerar documentação de pacotes inteiros, classes individuais ou ambos.

Os comentários em questão devem ser delimitados com os caracteres `/**` e `*/`, são escritos em HTML e devem preceder pacotes, classes, interfaces, constantes ou métodos.

Classes e interfaces com este tipo de comentários para devem, necessariamente ser gravadas individualmente em arquivos separados e a palavra `public` deverá ser colocada antes da palavra `class` ou `interface` que dá início a sua definição.

Tais comentários são constituídos por:

1. Uma linha contendo unicamente `/**` (marcação de início de comentário para javadoc);
  2. Linhas contendo um parágrafo constituído por um período único e corretamente pontuado que representa uma breve descrição daquilo que se deseja documentar;
  3. Linhas contendo parágrafos constituídos por qualquer número de períodos corretamente pontuados com informações complementares a respeito daquilo que se deseja documentar;
  4. Uma linha em branco;
  5. Linhas contendo parágrafos constituídos por qualquer número de períodos corretamente pontuados com informações a respeito de itens introduzidos por marcações iniciadas pelo caractere arroba (@);
  6. Uma linha contendo unicamente `*/` (marcação de fim de comentário para javadoc).
-

As possíveis marcações iniciadas pelo caractere arroba (@) são as listadas abaixo e devem ocorrer (quando ocorrerem) na ordem em que são abordadas a seguir:

### **1. @param**

Presta-se à documentação de parâmetros; deve se seguir de um nome de parâmetro e de uma descrição do mesmo (caso hajam vários parâmetros, devemos usar várias marcações deste tipo, uma para cada parâmetro, na ordem em que foram declarados);

### **2. @return**

Presta-se à documentação de retorno; deve se seguir da descrição daquilo que é retornado, especificando as situações em que cada possível retorno ocorre;

### **3. @throws**

Presta-se à documentação de exceções lançadas; originalmente @exception, deve se seguir de um nome de uma exceção e de uma descrição da situação em que a mesma é lançada (caso hajam várias exceções possivelmente lançadas, devemos usar várias marcações deste tipo, uma para cada exceção, na ordem em que foram especificadas);

### **4. @author**

Presta-se à especificação do autor daquilo que está sendo documentado; deve se seguir do nome do autor daquilo que está sendo documentado (caso hajam vários autores, devemos usar várias marcações deste tipo, uma para cada autor); autores não são incluídos na documentação gerada;

### **5. @version**

Presta-se à especificação da versão (e de sua data de liberação) daquilo que está sendo documentado; deve se seguir do número da versão e da data de sua liberação; quando usamos o SCCS, deve valer %I%, %G%, o que acaba sendo convertido em número da versão, seguida pela data de sua liberação, quando da extração do SCCS;

### **6. @see**

---

Presta-se à recomendar o exame de itens correlatos que, quando compreendidos, melhorariam o entendimento daquilo que está sendo documentado; deve se seguir do nome daquilo cujo exame está sendo recomendado; quando aquilo cujo exame está sendo recomendado for:

- a. Uma constante, construtor ou método:
  - Da própria classe que está sendo documentada, precederemos o nome daquilo cujo exame está sendo recomendado por um #;
  - De uma classe diferente daquela que está sendo documentada, porém do mesmo pacote, precederemos o nome daquilo cujo exame está sendo recomendado pelo nome da classe e em seguida por um #;
  - De uma classe de outro pacote, precederemos o nome daquilo cujo exame está sendo recomendado pelo nome do pacote, seguido por um ponto, seguido pelo nome da classe e em seguida por um #;
- b. Uma classe:
  - Do mesmo pacote daquilo que está sendo documentado, utilizaremos simplesmente seu nome;
  - De outro pacote, precederemos seu nome pelo nome do pacote seguido por um ponto;
- c. Um pacote:
  - Utilizaremos simplesmente seu nome,

Caso hajam vários itens cujo exame deseja-se recomendar, devemos usar várias marcações deste tipo, uma para cada item;

## 7. @since

Presta-se à especificação da versão a partir da qual aquilo que está sendo documentado foi incluído no produto documentado; deve se seguir do número desta versão;

---

**8. @literal**

Presta-se a fazer com que o texto que deve segui-lo seja mostrado literalmente como é, não interpretando-o (como comandos HTML, por exemplo); deve estar, bem como o texto que o segue, entre chaves;

**9. @code**

Como @literal, exceto pelo fato de que tem o efeito adicional de fazer com que o texto que o segue seja mostrado com fonte própria para mostrar código.

**10. @link**

Deve seguir-se de uma referência a uma constante, construtor, método, classe ou pacote elaborada conforme o fazemos em @see e de um texto, prestando-se a fazer constar na documentação o texto fornecido (formatado como código), fazendo-o funcionar como um link que conduz à documentação daquilo que foi referenciado; deve estar, bem como o texto que o segue, entre chaves;

**11. @linkplain**

Como @link, exceto pelo fato de que tem o texto fornecido não será formatado como código.

**12. @docRoot**

Representa o caminho relativo para a pasta na qual serão armazenados os arquivos e subpastas relativos a uma documentação gerada pelo javadoc; é útil na elaboração de links que precisem fazer referência àquela pasta; deve estar, bem como o texto que o segue, entre chaves;

**13. @value**

Pode ser usada na documentação de uma constante, caso em que representará o valor da constante em questão; pode ser usada em outros contextos seguida uma especificação de constante feita conforme o fazemos em @see, caso em que representará o valor daquela constante; deve estar, bem como o texto que o segue, entre chaves;

---

#### 14. @inheritDoc

Representa documentação do item que está sendo documentado na classe ou interface base (numa situação de herança); somente é válido em descrições de métodos e em @param, @return ou @trows; deve estar, bem como o texto que o segue, entre chaves;

#### 15. @deprecated

Presta-se à fazer constar em documentação a depreciação de um item previamente marcado por preceder o mesmo com um @Deprecated. Deve seguir-se de um texto que explique desde quando ocorreu a depreciação (versão e data), além de explicar a forma ora vigente.

#### 16. @serial, @serialfield e @serialdata

Para toda classe que implementa `Serializable` é gerada uma página especial com informações a respeito de suas constantes e métodos de serialização. Esta informação é de interesse de reimplementadores, não de desenvolvedores que utilizem a classe. Tem-se acesso a esta página clicando no link [\*\*Serializable Form\*\*](#) na seção [\*\*See also\*\*](#) do comentário da classe. Veja como utilizar estes recursos na documentação na documentação do javadoc.

## JARs

JARs, ou Java Archives, são repositórios com formato independente de plataforma, onde jazem diversos arquivos, por vezes diversos diretórios contendo diversos arquivos.

Além de serem úteis para a distribuição de aplicações Java, os JARs podem ser também empregados para construir pacotes de software contendo muitas applets, bem como todos os eventuais arquivos necessários a elas.

Tais pacotes podem ser baixados por um browser em uma única transação HTTP, melhorando significativamente a velocidade de carga de uma página. Além disso, o formato JAR também suporta compressão de dados, o que reduz também o tamanho do arquivo, o que faz cair ainda mais o tempo gasto na transação.

---

Além disso, o autor das applets pode assinar digitalmente cada entrada em um JAR para de forma a autenticar sua origem.

### [C:\ExplsJava\Expl\_08\Fracao.java]

```
1: /**
2:  A classe Fracao representa frações, conforme as conhecemos da matemática.
3:
4: Em outras palavras, a classe Fracao representa o conjunto matemático dos
5: números racionais. Nela encontraremos diversos métodos para operar com frações.
6: @author André Luís dos Reis Gomes de Carvalho
7: @since 2000
8: */
9: public class Fracao implements Comparable<Fracao>, Cloneable
10: {
11:     private long numerador, denominador;
12:
13:     private double valorReal ()
14:     {
15:         return (double)numerador /
16:             (double)denominador;
17:     }
18:
19:     private void simplifiqueSe ()
20:     {
21:         long menor = Math.min (Math.abs (this.numerador),
22:                               Math.abs (this.denominador));
23:
24:         if (this.numerador%this.denominador == 0)
25:         {
26:             this.numerador = this.numerador / this.denominador;
27:             this.denominador = 1;
28:         }
29:         else
30:             if (this.numerador %menor == 0 &&
31:                 this.denominador%menor == 0)
32:             {
33:                 this.numerador = this.numerador /menor;
34:                 this.denominador = this.denominador/menor;
35:             }
36:         else
37:             for (long metade=menor/2, i=2; i<=metade; i++)
38:                 while (this.numerador %i == 0 &&
39:                        this.denominador%i == 0)
40:                 {
41:                     this.numerador = this.numerador /i;
42:                     this.denominador = this.denominador/i;
43:                 }
44:     }
45:
46: /**
47: Constrói uma nova instância da classe Fracao.
48: Para tanto, devem ser fornecidos dois inteiros que serão utilizados
49: respectivamente, como numerador e como denominador da instância recém
```

```
50:     criada.
51:     @param numerador o número inteiro a ser utilizado como numerador
52:     @param denominador o número inteiro a ser utilizado como denominador
53:     @throws Exception se o denominador for igual a zero
54:     */
55:     public Fracao (long numerador,
56:                     long denominador)
57:                     throws Exception
58:     {
59:         if (denominador == 0)
60:             throw new Exception ("Denominador zero");
61:
62:         if (denominador < 0)
63:         {
64:             this.numerador = -numerador;
65:             this.denominador = -denominador;
66:         }
67:         else
68:         {
69:             this.numerador = numerador;
70:             this.denominador = denominador;
71:         }
72:
73:         this.simplifiqueSe ();
74:     }
75:
76:     /**
77:      Obtém o numerador de uma fração.
78:      Resulta o numerador da instância à qual este método for aplicado.
79:      @return o numerador da fração chamante do método
80:     */
81:     public long getNumerador ()
82:     {
83:         return this.numerador;
84:     }
85:
86:     /**
87:      Obtém o denominador de uma fração.
88:      Resulta o denominador da instância à qual este método for aplicado.
89:      @return o denominador da fração chamante do método
90:     */
91:     public long getDenominador ()
92:     {
93:         return this.denominador;
94:     }
95:
96:     /**
97:      Ajusta o numerador de uma fração.
98:      Ajusta o numerador da instância à qual este método for aplicado.
99:      @param numerador o numerador que a fração chamante do método
100:         deve passar a ter
101:     */
102:    public void setNumerador (long numerador)
103:    {
104:        this.numerador = numerador;
105:        this.simplifiqueSe ();
```

```
106:    }
107:
108:    /**
109:     Ajusta o denominador de uma fração.
110:     Ajusta o denominador da instância à qual este método for aplicado.
111:     @param denominador o denominador que a fração chamante do método
112:         deve passar a ter
113:     @throws Exception se o denominador for igual a zero
114:    */
115:    public void setDenominador (long denominador) throws Exception
116:    {
117:        if (denominador==0)
118:            throw new Exception ("Denominador zero");
119:
120:        this.denominador = denominador;
121:        this.simplifiqueSe ();
122:    }
123:
124:    /**
125:     Realiza a operação de soma para duas frações.
126:     Soma a instância à qual este método for aplicado com aquela que lhe
127:     for fornecida como parâmetro.
128:     @param fracao a fração que deve ser somada à chamante do método
129:     @return a fração chamante do método somada à fração fornecida
130:     @throws Exception se for fornecido null como parâmetro
131:    */
132:    public Fracao mais (Fracao fracao) throws Exception
133:    {
134:        if (fracao == null)
135:            throw new Exception ("Falta de operando em soma");
136:
137:        long numerador    = this.numerador    * fracao.denominador +
138:                           this.denominador * fracao.numerador,
139:
140:        denominador = this.denominador * fracao.denominador;
141:
142:        Fracao resultado = new Fracao (numerador,denominador);
143:
144:        resultado.simplifiqueSe ();
145:        return resultado;
146:    }
147:
148:    /**
149:     Realiza a operação de subtração para duas frações.
150:     Subtrai da instância à qual este método for aplicado aquela que lhe
151:     for fornecida como parâmetro.
152:     @param fracao a fração que deve ser subtraída da chamante do método
153:     @return a fração chamante do método menos a fração fornecida
154:     @throws Exception se for fornecido null como parâmetro
155:    */
156:    public Fracao menos (Fracao fracao) throws Exception
157:    {
158:        if (fracao == null)
159:            throw new Exception ("Falta de operando em soma");
160:
161:        long numerador    = this.numerador    * fracao.denominador -
```

```
162:                     this.denominador * fracao.numerador,
163:
164:                     denominador = this.denominador * fracao.denominador;
165:
166:                     Fracao resultado = new Fracao (numerador, denominador);
167:
168:                     resultado.simplifiqueSe ();
169:                     return resultado;
170:                 }
171:
172:             /**
173:              Realiza a operação de multiplicação para duas frações.
174:              Multiplica a instância à qual este método for aplicado por aquela que lhe
175:              for fornecida como parâmetro.
176:              @param fracao a fração que deve ser multiplicada pela chamante do método
177:              @return a fração chamante do método multiplicada pela fração fornecida
178:              @throws Exception se for fornecido null como parâmetro
179:             */
180:             public Fracao vezes (Fracao fracao) throws Exception
181:             {
182:                 if (fracao == null)
183:                     throw new Exception ("Falta de operando em multiplicacao");
184:
185:                 long numerador    = this.numerador    * fracao.numerador,
186:                     denominador = this.denominador * fracao.denominador;
187:
188:                 Fracao resultado = new Fracao (numerador,denominador);
189:
190:                 resultado.simplifiqueSe ();
191:                 return resultado;
192:             }
193:
194:             /**
195:              Realiza a operação de divisão para duas frações.
196:              Divide a instância à qual este método for aplicado por aquela que lhe
197:              for fornecida como parâmetro.
198:              @param fracao a fração pela qual deve ser dividida a chamante do método
199:              @return a fração chamante do método dividida pela fração fornecida
200:              @throws Exception se for fornecido null como parâmetro
201:             */
202:             public Fracao divididoPor (Fracao fracao) throws Exception
203:             {
204:                 if (fracao == null)
205:                     throw new Exception ("Falta de operando em divisao");
206:
207:                 if (fracao.numerador == 0)
208:                     throw new Exception ("Divisao por zero");
209:
210:                 long numerador    = this.numerador    * fracao.denominador,
211:                     denominador = this.denominador * fracao.numerador;
212:
213:                 if (denominador < 0)
214:                 {
215:                     numerador    = -numerador;
216:                     denominador = -denominador;
217:                 }
```

```
218:
219:     Fracao resultado = new Fracao (numerador,denominador);
220:
221:     resultado.simplifiqueSe ();
222:     return resultado;
223: }
224:
225: /**
226:  * Converte uma fração em um String.
227:  * Produz e resulta uma instância da classe String que representa a
228:  * instância à qual este método for aplicado.
229:  * @return o String que representa a fração chamante do método
230:  */
231: public String toString ()
232: {
233:     if (this.numerador == this.denominador)
234:         return "1";
235:
236:     if (this.numerador + this.denominador == 0)
237:         return "-1";
238:
239:     if (this.numerador == 0 || this.denominador == 1)
240:         return "" + this.numerador;
241:
242:     return this.numerador + "/" + this.denominador;
243: }
244:
245: /**
246:  * Verifica a igualdade entre duas frações.
247:  * Verifica se o Object fornecido como parâmetro representa uma
248:  * fração numericamente equivalente àquela representada pela instância
249:  * à qual este método for aplicado, resultando true em caso afirmativo,
250:  * ou false, caso contrário.
251:  * @return true, caso o Object fornecido ao método e a instância chamante do
252:  *         método representarem frações numericamente equivalentes, ou false,
253:  *         caso contrário
254:  */
255: public boolean equals (Object obj)
256: {
257:     if (this == obj)
258:         return true;
259:
260:     if (obj == null)
261:         return false;
262:
263:     //if (!obj instanceof Fracao)
264:     if (this.getClass () != obj.getClass ())
265:         return false; // poderíamos fazer de outra forma
266:                         // caso quisessemos compatibilizar
267:                         // fracos com outras classes, e.g.,
268:                         // com strings; naturalmente, isso
269:                         // causaria impacto no hashCode
270:
271:     Fracao fracao = (Fracao) obj;
272:
273:     if (this.valorReal() != fracao.valorReal())
```

```
274:         return false;
275:
276:         return true;
277:     }
278:
279:     /**
280:      Calcula o código de espalhamento (ou código de hash) de uma fração.
281:      Calcula e resulta o código de espalhamento (ou código de hash, ou ainda o
282:      hashCode) da fração representada pela instância à qual o método for aplicado.
283:      @return o código de espalhamento da fração chamante do método
284:     */
285:    public int hashCode ()
286:    {
287:        int ret = super.hashCode ();
288:
289:        ret = 13*ret +
290:              new Long(numerador).hashCode ();
291:
292:        ret = 13*ret +
293:              new Long(denominador).hashCode ();
294:
295:        return ret;
296:    }
297:
298:    /**
299:     Compara duas frações.
300:     Compara as frações representadas respectivamente pela instância à qual
301:     o método for aplicado e pela instância fornecida como parâmetro, resultando
302:     um número negativo, caso a primeira seja numericamente menor que a segunda,
303:     zero, caso as duas sejam numericamente iguais, ou
304:     um número positivo, caso a primeira seja numericamente maior que a segunda.
305:     @return um número negativo, caso a primeira seja numericamente menor que a
306:             segunda, zero, caso as duas sejam numericamente iguais, ou um
307:             número positivo, caso a primeira seja numericamente maior que a
308:             segunda.
309:
310:     */
311:    public int compareTo (Fracao fracao)
312:    {
313:        double vrThis    = this .valorReal(),
314:              vrFracao = fracao.valorReal();
315:
316:        if (vrThis < vrFracao)
317:            return -1;
318:        else
319:            if (vrThis == vrFracao)
320:                return 0;
321:            else
322:                return 1;
323:    }
324:
325:    /**
326:     Constroi uma cópia da instância da classe Fracao dada.
327:     Para tanto, deve ser fornecida uma instancia da classe Fracao para ser
328:     utilizada como modelo para a construção da nova instância criada.
329:     @param modelo a instância da classe Fracao a ser usada como modelo
```

```
330:     @throws Exception se o modelo for null
331:     */
332:     public Fracao (Fracao modelo)
333:             throws Exception
334:     {
335:         if (modelo==null)
336:             throw new Exception ("Modelo não fornecido");
337:
338:         this.numerador = modelo.numerador;
339:         this.denominador = modelo.denominador;
340:     }
341:
342:     /**
343:      Clona uma fração.
344:      Produz e resulta uma cópia da fração representada pela instância
345:      à qual o método for aplicado.
346:      @return a cópia da fração representada pela instância à qual
347:              o método for aplicado
348:     */
349:     public Object clone ()
350:     {
351:         Fracao copia=null;
352:
353:         try
354:         {
355:             copia = new Fracao (this);
356:         }
357:         catch (Exception e)
358:         {}
359:
360:         return copia;
361:     }
362: }
```

[C:\ExplsJava\Expl\_08\TesteDeFracao.java]

```
1: public class TesteDeFracao
2: {
3:     public static void main (String[] args)
4:     {
5:         Fracao f1, f2, f3, f4, f5;
6:
7:         try
8:         {
9:             f2 = new Fracao (5,7);
10:
11:             System.out.println ("Numerador de " + f2 + " = " +
12:                                 f2.getNumerador());
13:
14:             System.out.println ("Denominador de " + f2 + " = " +
15:                                 f2.getDenominador());
16:
17:             System.out.println ();
18:
19:             f3 = f2;
```

```
20:         f4 = (Fracao)f2.clone();
21:         f5 = (Fracao)f2.clone();
22:
23:         System.out.println ("Alterando numerador e denominador, ");
24:         System.out.println ("respectivamente para 1 e 2, ");
25:         System.out.print     (f2 + " torna-se ");
26:
27:         f2.setNumerador    (1);
28:         f2.setDenominador (2);
29:
30:         System.out.println (f2);
31:         System.out.println ();
32:
33:         System.out.println ("Observe que a mudanca acima");
34:         System.out.println ("também afeta " + f3);
35:         System.out.println ("mas nao " + f4);
36:
37:         System.out.println ();
38:
39:         f1 = f2.mais(f4);
40:         System.out.println (f1 + " = " + f2 + " + " + f4);
41:
42:         f1 = f2.menos(f4);
43:         System.out.println (f1 + " = " + f2 + " - " + f4);
44:
45:         f1 = f2.vezes(f4);
46:         System.out.println (f1 + " = " + f2 + " * " + f4);
47:
48:         f1 = f2.divididoPor(f4);
49:         System.out.println (f1 + " = " + f2 + " / " + f4);
50:
51:         System.out.println ();
52:
53:         System.out.println (f2 + " == " + f2.valorReal());
54:
55:         System.out.println ();
56:
57:         if (f2==f3)
58:             System.out.println (f2 + " == " + f3 + " (pelo ==)");
59:         else
60:             System.out.println (f2 + " != " + f3 + " (pelo ==)");
61:
62:         if (f2.equals(f3))
63:             System.out.println (f2 + " == " + f3 + " (pelo equals)");
64:         else
65:             System.out.println (f2 + " != " + f3 + " (pelo equals)");
66:
67:         if (f3==f4)
68:             System.out.println (f3 + " == " + f4 + " (pelo ==)");
69:         else
70:             System.out.println (f3 + " != " + f4 + " (pelo ==)");
71:
72:         if (f3.equals(f4))
73:             System.out.println (f3 + " == " + f4 + " (pelo equals)");
74:         else
75:             System.out.println (f3 + " != " + f4 + " (pelo equals)");
```



```
132:
133:         comp = f4.compareTo(f5);
134:         if (comp < 0)
135:             System.out.println (f4 + " < " + f5 + " (pelo compareTo)");
136:         else
137:             if (comp == 0)
138:                 System.out.println (f4 + " == " + f5 +
139:                                     " (pelo compareTo)");
140:             else
141:                 System.out.println (f4 + " > " + f5 +
142:                                     " (pelo compareTo)");
143:
144:         comp = f5.compareTo(f4);
145:         if (comp < 0)
146:             System.out.println (f5 + " < " + f4 + " (pelo compareTo)");
147:         else
148:             if (comp == 0)
149:                 System.out.println (f5 + " == " + f4 +
150:                                     " (pelo compareTo)");
151:             else
152:                 System.out.println (f5 + " > " + f4 +
153:                                     " (pelo compareTo)");
154:
155:         System.out.println ();
156:
157:         System.out.println ("O codigo de espalhamento de " +
158:                             "uma instancia valendo " + f1 +
159:                             " vale " + f1.hashCode());
160:
161:         System.out.println ("O codigo de espalhamento de " +
162:                             "uma instancia valendo " + f2 +
163:                             " vale " + f2.hashCode());
164:
165:         System.out.println ("O codigo de espalhamento de " +
166:                             "outra instancia valendo " + f3 +
167:                             " vale " + f3.hashCode());
168:
169:         System.out.println ("O codigo de espalhamento de " +
170:                             "uma instancia valendo " + f4 +
171:                             " vale " + f4.hashCode());
172:
173:         System.out.println ("O codigo de espalhamento de " +
174:                             "outra instancia valendo " + f5 +
175:                             " vale " + f5.hashCode());
176:     }
177:     catch (Exception e)
178:     {
179:         System.err.println (e.getMessage ());
180:     }
181: }
182: }
```

[C:\ExplsJava\Expl\_08\FcaoPrin.MF]

```
1: Main-Class: TesteDeFracao
```

Podemos observar que o exemplo acima apresenta um arquivo de nome FcaoPrin.MF. Este arquivo será empregado no momento de criar o JAR e tem por função indicar qual é a classe principal do JAR, i.e., a classe a partir da qual a execução terá início.

Para compilar este programa, procederemos como temos procedido com todos os programas compilados anteriormente, ou seja, daremos o seguinte comando:

```
C:\ExplsJava\Expl_08> javac TesteDeFracao.java
```

Para criar o JAR, basta dar o seguinte comando:

```
C:\ExplsJava\Expl_08> jar fmc Fracao.jar FcaoPrin.MF Fracao.class TesteDeFracao.class
```

Poderemos observar que foi criado o arquivo Fracao.jar. Para executá-lo, basta dar o seguinte comando:

```
C:\ExplsJava\Expl_08> java -jar Fracao.jar
```

Isso poderia produzir no console a seguinte interação:

```
Numerador de 5/7 = 5  
Denominador de 5/7 = 7
```

```
Alterando numerador e denominador,  
respectivamente para 1 e 2,  
5/7 torna-se 1/2
```

```
Observe que a mudança acima  
também afeta 1/2  
mas não 5/7
```

```
17/14 = 1/2 + 5/7  
-3/14 = 1/2 - 5/7  
5/14 = 1/2 * 5/7  
7/10 = 1/2 / 5/7
```

```
1/2 == 1/2 (pelo ==)  
1/2 == 1/2 (pelo equals)  
1/2 != 5/7 (pelo ==)  
1/2 != 5/7 (pelo equals)  
5/7 != 5/7 (pelo ==)  
5/7 == 5/7 (pelo equals)
```

---

```
1/2 == 1/2 (pelo compareTo)
1/2 == 1/2 (pelo compareTo)
1/2 < 5/7 (pelo compareTo)
5/7 > 1/2 (pelo compareTo)
5/7 == 5/7 (pelo compareTo)
5/7 == 5/7 (pelo compareTo)
```

O código de espalhamento de uma instância valendo 7/10 vale 1188  
O código de espalhamento de uma instância valendo 1/2 vale 994  
O código de espalhamento de outra instância valendo 1/2 vale 994  
O código de espalhamento de uma instância valendo 5/7 vale 1123  
O código de espalhamento de outra instância valendo 5/7 vale 1123

## Sobrecarga

Java permite sobrecarga de nomes de função, i.e., um único nome para diversas funções diferentes.

Cada operação deve ter tipos de parâmetros diferentes. Java seleciona uma operação com base nos tipos dos argumentos fornecidos.

O conceito de sobrecarga, além de possibilitar nomes de função mais significativos, ainda possibilita a escrita de código polimórfico.

## Criação e Destrução

O autor de uma classe pode controlar o que deve acontecer por ocasião da criação e da destruição de instâncias da classe escrevendo funções construtoras e destrutoras.

Construtores servem basicamente para iniciar os membros da instância que está sendo criada. Quanto aos destrutores, como Java tem um mecanismo automático de coleta de lixo, não é preciso, como em outras linguagens, se preocupar com liberação de memória. Por esta razão, é muito raro ser preciso utilizar um destrutor em Java. Mas se a destruição de um objeto tiver implicações outras que a liberação de memória, neste caso empregar um destrutor pode ser interessante.

---

Quando uma instância é criada, aloca-se memória para a instância e em seguida é chamado o construtor da classe (se houver um). Quando uma instância é destruída, o destrutor é chamado (se houver um) e em seguida libera-se a memória alocada para a instância.

Construtores são batizados com o nome da classe a que se referem e destrutores são batizados com o nome de finalize .

Como os construtores e os destrutores são chamados automaticamente, não faz sentido que eles tenham retorno. Não indicamos o tipo retornado por um construtor (nem mesmo void). O tipo de retorno de um destrutores é sempre void.

Destrutores não podem ter parâmetros, o que significa que toda classe terá no máximo um destrutor. Não é possível precisar o momento da execução de um destrutor, já que não se sabe em que momento o mecanismo de coleta de lixo vai liberar a memória associada aos objetos e vetores inutilizados.

Construtores podem ter parâmetros, o que significa que podemos ter vários deles em uma mesma classe (sobrecarga de construtores), cada qual realizando a iniciação do objeto em processo de criação de uma maneira diferente.

No caso de uma classe ter vários construtores, se na implementação de um deles for desejável chamar um outro, poderemos fazê-lo através da palavra chave this (em vez de usar o nome da classe). Este comando sempre deverá ser o PRIMEIRO comando de um construtor, e será construído como segue: a palavra this eventualmente seguida dos argumentos separados por vírgulas (,) e entre parênteses () a serem passados ao construtor a ser chamado.

### [C:\ExplsJava\Expl\_09\Fracao.java]

```
1: import java.util.StringTokenizer;
2:
3: /**
4:  A classe Fracao representa frações, conforme as conhecemos da matemática.
5:
6:  Em outras palavras, a classe Fracao representa o conjunto matemático dos
7:  números racionais. Nela encontraremos diversos métodos para operar com frações.
8:  @author André Luís dos Reis Gomes de Carvalho
9:  @since 2000
10: */
```

---

```
11: public class Fracao implements Comparable<Fracao>, Cloneable
12: {
13:     private long numerador, denominador;
14:
15:     private double valorReal ()
16:     {
17:         return (double)numerador /
18:                (double)denominador;
19:     }
20:
21:     private void assumaValor (long numerador,
22:                               long denominador)
23:                               throws Exception
24:     {
25:         if (denominador == 0)
26:             throw new Exception ("Denominador zero");
27:
28:         if (denominador < 0)
29:         {
30:             this.numerador = -numerador;
31:             this.denominador = -denominador;
32:         }
33:         else
34:         {
35:             this.numerador = numerador;
36:             this.denominador = denominador;
37:         }
38:
39:         this.simplifiqueSe ();
40:     }
41:
42:     private void simplifiqueSe ()
43:     {
44:         long menor = Math.min (Math.abs (this.numerador),
45:                               Math.abs (this.denominador));
46:
47:         if (this.numerador%this.denominador == 0)
48:         {
49:             this.numerador = this.numerador / this.denominador;
50:             this.denominador = 1;
51:         }
52:         else
53:             if (this.numerador %menor == 0 &&
54:                 this.denominador%menor == 0)
55:             {
56:                 this.numerador = this.numerador /menor;
57:                 this.denominador = this.denominador/menor;
58:             }
59:             else
60:                 for (int i=2; i<=menor/2; i++)
61:                     while (this.numerador %i == 0 &&
62:                            this.denominador%i == 0)
63:                     {
64:                         this.numerador = this.numerador /i;
65:                         this.denominador = this.denominador/i;
66:                     }
67:     }
68:
```

```
67:    }
68:
69:    /**
70:     Constroi uma nova instância da classe Fracao.
71:     Para tanto, devem ser fornecidos dois inteiros que serão utilizados
72:     respectivamente, como numerador e como denominador da instância recém
73:     criada.
74:     @param numerador o número inteiro a ser utilizado como numerador
75:     @param denominador o número inteiro a ser utilizado como denominador
76:     @throws Exception se o denominador for igual a zero
77:    */
78:    public Fracao (long numerador, // construtor DE COMPATIBILIZAÇÃO
79:                   long denominador) // para construir uma fracao a
80:                   throws Exception // partir de um par de numeros
81:    {
82:        // inteiros
83:        this.assumaValor (numerador, denominador);
84:
85:    /**
86:     Constroi uma nova instância da classe Fracao.
87:     Para tanto, deve ser fornecido um String com a forma de fração, ou
88:     seja, contendo um valor inteiro e um valor natural separados por
89:     uma barra.
90:     Esses valores serão utilizados, respectivamente, como numerador e
91:     como denominador da instância recém criada.
92:     Na eventualidade da omissao da barra e do valor natural que faria
93:     o papel de denominador, a nova instância será construída com
94:     denominador igual a 1.
95:     @param fracao o String com forma de fração
96:     @throws Exception se o String fornecido for null ou igual ao String
97:                   vazio, ou ainda se contiver caracteres inválidos ou em excesso.
98:    */
99:    public Fracao (String fracao) // construtor DE COMPATIBILIZAÇÃO
100:                   throws Exception // para construir uma fracao a
101:    {
102:        // partir de um string
103:        if (fracao == null)
104:            throw new Exception ("Tentativa de criar fracao a partir do nada");
105:        if (fracao.length () == 0)
106:            throw new Exception ("Tentativa de criar fracao a partir do nada");
107:
108:        StringTokenizer separador = new StringTokenizer (fracao,"/");
109:
110:        long numerador;
111:
112:        try
113:        {
114:            numerador = Long.parseLong (separador.nextToken());
115:        }
116:        catch (NumberFormatException e)
117:        {
118:            throw new Exception ("Caracteres invalidos para fracao");
119:        }
120:
121:        long denominador = 1;
122:
```

```
123:         if (separador.hasMoreTokens ())
124:             try
125:             {
126:                 denominador = Long.parseLong (separador.nextToken ());
127:             }
128:             catch (NumberFormatException e)
129:             {
130:                 throw new Exception ("Caracteres invalidos para fracao");
131:             }
132:
133:         if (separador.hasMoreTokens ())
134:             throw new Exception ("Excesso de caracteres");
135:
136:         this.assumaValor (numerador, denominador);
137:     }
138:
139:     /**
140:      Obtém o numerador de uma fração.
141:      Resulta o numerador da instância à qual este método for aplicado.
142:      @return o numerador da fração chamante do método
143:     */
144:     public long getNumerador ()
145:     {
146:         return this.numerador;
147:     }
148:
149:     /**
150:      Obtém o denominador de uma fração.
151:      Resulta o denominador da instância à qual este método for aplicado.
152:      @return o denominador da fração chamante do método
153:     */
154:     public long getDenominador ()
155:     {
156:         return this.denominador;
157:     }
158:
159:     /**
160:      Ajusta o numerador de uma fração.
161:      Ajusta o numerador da instância à qual este método for aplicado.
162:      @param numerador o numerador que a fração chamante do método
163:          deve passar a ter
164:     */
165:     public void setNumerador (long numerador)
166:     {
167:         this.numerador = numerador;
168:         this.simplifiqueSe ();
169:     }
170:
171:     /**
172:      Ajusta o denominador de uma fração.
173:      Ajusta o denominador da instância à qual este método for aplicado.
174:      @param denominador o denominador que a fração chamante do método
175:          deve passar a ter
176:      @throws Exception se o denominador for igual a zero
177:     */
178:     public void setDenominador (long denominador) throws Exception
```

```
179:  {
180:      if (denominador==0)
181:          throw new Exception ("Denominador zero");
182:
183:      this.denominador = denominador;
184:      this.simplifiqueSe ();
185:  }
186:
187: /**
188:  * Realiza a operação de soma para duas frações.
189:  * Soma a instância à qual este método for aplicado com aquela que lhe
190:  * for fornecida como parâmetro.
191:  * @param fracao a fração que deve ser somada à chamante do método
192:  * @return a fração chamante do método somada à fração fornecida
193:  * @throws Exception se for fornecido null como parâmetro
194:  */
195: public Fracao mais (Fracao fracao) throws Exception
196: {
197:     if (fracao == null)
198:         throw new Exception ("Falta de operando em soma");
199:
200:     long numerador = this.numerador * fracao.denominador +
201:                     this.denominador * fracao.numerador,
202:
203:     denominador = this.denominador * fracao.denominador;
204:
205:     Fracao resultado = new Fracao (numerador,denominador);
206:
207:     resultado.simplifiqueSe ();
208:     return resultado;
209: }
210:
211: /**
212:  * Realiza a operação de soma de uma fração e um número inteiro.
213:  * Soma a instância à qual este método for aplicado com o número
214:  * inteiro que for fornecido como parâmetro.
215:  * @param numero o número inteiro que deve ser somado à fração
216:  * chamante do método.
217:  * @return a fração chamante do método somada ao número inteiro
218:  * fornecido.
219:  */
220: public Fracao mais (long numero)
221: {
222:     long numerador = this.denominador * numero +
223:                     this.numerador,
224:
225:     denominador = this.denominador;
226:
227:     Fracao resultado = null;
228:
229:     try
230:     {
231:         resultado = new Fracao (numerador, denominador);
232:     }
233:     catch (Exception e)
234:     {}
```

```
235:
236:         resultado.simplifiqueSe ();
237:         return resultado;
238:     }
239:
240:    /**
241:     Realiza a operação de subtração para duas frações.
242:     Subtrai da instância à qual este método for aplicado aquela que lhe
243:     for fornecida como parâmetro.
244:     @param fracao a fração que deve ser subtraída da chamante do método
245:     @return a fração chamante do método menos a fração fornecida
246:     @throws Exception se for fornecido null como parâmetro
247:    */
248:    public Fracao menos (Fracao fracao) throws Exception
249:    {
250:        if (fracao == null)
251:            throw new Exception ("Falta de operando em soma");
252:
253:        long numerador = this.numerador * fracao.denominador -
254:                         this.denominador * fracao.numerador,
255:
256:        denominador = this.denominador * fracao.denominador;
257:
258:        Fracao resultado = new Fracao (numerador, denominador);
259:
260:        resultado.simplifiqueSe ();
261:        return resultado;
262:    }
263:
264:    /**
265:     Realiza a operação de subtração entre uma fração e um número inteiro.
266:     Subtrai da instância à qual este método for aplicado o número
267:     inteiro que for fornecido como parâmetro.
268:     @param numero o número inteiro que deve ser subtraído da fração
269:     chamante do método.
270:     @return a diferença entre a fração chamante e o número inteiro
271:     fornecido.
272:    */
273:    public Fracao menos (long numero)
274:    {
275:        long numerador = this.denominador * numero -
276:                         this.numerador,
277:
278:        denominador = this.denominador;
279:
280:        Fracao resultado = null;
281:
282:        try
283:        {
284:            resultado = new Fracao (numerador, denominador);
285:        }
286:        catch (Exception e)
287:        {}
288:
289:        resultado.simplifiqueSe ();
290:        return resultado;
```

```
291:    }
292:
293:    /**
294:     Realiza a operação de multiplicação para duas frações.
295:     Multiplica a instância à qual este método for aplicado por aquela que lhe
296:     for fornecida como parâmetro.
297:     @param fracao a fração que deve ser multiplicada pela chamante do método
298:     @return a fração chamante do método multiplicada pela fração fornecida
299:     @throws Exception se for fornecido null como parâmetro
300:    */
301:    public Fracao vezes (Fracao fracao) throws Exception
302:    {
303:        if (fracao == null)
304:            throw new Exception ("Falta de operando em multiplicacao");
305:
306:        long numerador = this.numerador * fracao.numerador,
307:             denominador = this.denominador * fracao.denominador;
308:
309:        Fracao resultado = new Fracao (numerador,denominador);
310:
311:        resultado.simplifiqueSe ();
312:        return resultado;
313:    }
314:
315:    /**
316:     Realiza a operação de multiplicação de uma fração por um
317:     número inteiro.
318:     Multiplia a instância à qual este método for aplicado pelo número
319:     inteiro que for fornecido como parâmetro.
320:     @param numero o número inteiro pelo qual deve ser multiplicado a
321:     fração chamante do método.
322:     @return a fração chamante do método multiplicada pelo número
323:     inteiro fornecido.
324:    */
325:    public Fracao vezes (long numero)
326:    {
327:        long numerador = this.numerador * numero,
328:             denominador = this.denominador;
329:
330:        Fracao resultado = null;
331:
332:        try
333:        {
334:            resultado = new Fracao (numerador, denominador);
335:        }
336:        catch (Exception e)
337:        {}
338:
339:        resultado.simplifiqueSe ();
340:        return resultado;
341:    }
342:
343:    /**
344:     Realiza a operação de divisão para duas frações.
345:     Divide a instância à qual este método for aplicado por aquela que lhe
346:     for fornecida como parâmetro.
```

```
347:     @param fracao a fração pela qual deve ser dividida a chamante do método
348:     @return a fração chamante do método dividida pela fração fornecida
349:     @throws Exception se for fornecido null como parâmetro ou uma fração
350:         cujo valor é zero.
351:     */
352:     public Fracao divididoPor (Fracao fracao) throws Exception
353:     {
354:         if (fracao == null)
355:             throw new Exception ("Falta de operando em divisao");
356:
357:         if (fracao.numerador == 0)
358:             throw new Exception ("Divisao por zero");
359:
360:         long numerador = this.numerador * fracao.denominador,
361:             denominador = this.denominador * fracao.numerador;
362:
363:         if (denominador < 0)
364:         {
365:             numerador = -numerador;
366:             denominador = -denominador;
367:         }
368:
369:         Fracao resultado = new Fracao (numerador,denominador);
370:
371:         resultado.simplifiqueSe ();
372:         return resultado;
373:     }
374:
375:     /**
376:      Realiza a operação de divisão de uma fração por um número
377:      inteiro.
378:      Divide a instância à qual este método for aplicado pelo
379:      número inteiro que for fornecido como parâmetro.
380:      @param numero o número inteiro pelo qual deve ser dividido
381:      a fração chamante do método.
382:      @return a divisão da fração chamante pelo número inteiro
383:      fornecido.
384:      @throws Exception se for fornecido zero como parâmetro
385:      */
386:     public Fracao divididoPor (long numero) throws Exception
387:     {
388:         if (numero == 0)
389:             throw new Exception ("Divisao por zero");
390:
391:         long numerador = this.numerador,
392:             denominador = this.denominador * numero;
393:
394:         if (denominador < 0)
395:         {
396:             numerador = -numerador;
397:             denominador = -denominador;
398:         }
399:
400:         Fracao resultado = null;
401:
402:         try
```

```
403:         {
404:             resultado = new Fracao (numerador, denominador);
405:         }
406:     catch (Exception e)
407:     {}
408:
409:     resultado.simplifiqueSe ();
410:     return resultado;
411: }
412:
413: /**
414:  * Converte uma fração em um String.
415:  * Produz e resulta uma instância da classe String que representa a
416:  * instância à qual este método for aplicado.
417:  * @return o String que representa a fração chamante do método
418:  */
419: public String toString ()
420: {
421:     if (this.numerador == this.denominador)
422:         return "1";
423:
424:     if (this.numerador + this.denominador == 0)
425:         return "-1";
426:
427:     if (this.numerador == 0 || this.denominador == 1)
428:         return "" + this.numerador;
429:
430:     return this.numerador + "/" + this.denominador;
431: }
432:
433: /**
434:  * Verifica a igualdade entre duas frações.
435:  * Verifica se o Object fornecido como parâmetro representa uma
436:  * fração numericamente equivalente àquela representada pela instância
437:  * à qual este método for aplicado, resultando true em caso afirmativo,
438:  * ou false, caso contrário.
439:  * @return true, caso o Object fornecido ao método e a instância chamante do
440:  * método representarem frações numericamente equivalentes, ou false,
441:  * caso contrário
442:  */
443: public boolean equals (Object obj)
444: {
445:     if (this == obj)
446:         return true;
447:
448:     if (obj == null)
449:         return false;
450:
451:     if (this.getClass() != obj.getClass())
452:         return false; // poderíamos fazer de outra forma
453:                     // caso quisessemos compatibilizar
454:                     // fracos com outras classes, e.g.,
455:                     // com strings.
456:
457:     Fracao fracao = (Fracao) obj;
458:
```

```
459:         if (this.valorReal() != fracao.valorReal())
460:             return false;
461:
462:         return true;
463:     }
464:
465:     /**
466:      Compara duas frações.
467:      Compara as frações representadas respectivamente pela instância à qual
468:      o método for aplicado e pela instância fornecida como parâmetro, resultando
469:      um número negativo, caso a primeira seja numericamente menor que a segunda,
470:      zero, caso as duas sejam numericamente iguais, ou
471:      um número positivo, caso a primeira seja numericamente maior que a segunda.
472:      @return um número negativo, caso a primeira seja numericamente menor que a
473:              segunda, zero, caso as duas sejam numericamente iguais, ou um
474:              número positivo, caso a primeira seja numericamente maior que a
475:              segunda.
476:
477:     */
478:    public int compareTo (Fracao fracao)
479:    {
480:        double vrThis = this .valorReal(),
481:              vrFracao = fracao.valorReal();
482:
483:        if (vrThis < vrFracao)
484:            return -1;
485:        else
486:            if (vrThis == vrFracao)
487:                return 0;
488:            else
489:                return 1;
490:    }
491:
492:    /**
493:       Calcula o código de espalhamento (ou código de hash) de uma fração.
494:       Calcula e resulta o código de espalhamento (ou código de hash, ou ainda o
495:       hashCode) da fração representada pela instância à qual o método for aplicado.
496:       @return o código de espalhamento da fração chamante do método
497:     */
498:    public int hashCode ()
499:    {
500:        int ret = super.hashCode();
501:
502:        ret = 13*ret +
503:              new Long(numerador).hashCode();
504:
505:        ret = 13*ret +
506:              new Long(denominador).hashCode();
507:
508:        return ret;
509:    }
510:
511:    /**
512:       Constroi uma cópia de uma instância da classe Fracao.
513:       Para tanto, deve ser fornecida uma outra instancia da classe Fração
514:       para ser modelo da cópia recém criada.
```

```
515:     @param fracao a instância a ser copiada
516:     */
517:     public Fracao (Fracao modelo)
518:             throws Exception
519:     {
520:         if (modelo==null)
521:             throw new Exception ("Modelo não fornecido");
522:
523:         this.numerador = modelo.numerador;
524:         this.denominador = modelo.denominador;
525:     }
526:
527:     /**
528:      Produz uma cópia fiel de uma fração.
529:      Produz e resulta uma cópia exata da fração à qual o método for aplicado.
530:      @return a cópia da fração chamante do método
531:      */
532:     public Object clone ()
533:     {
534:         Fracao copia=null;
535:
536:         try
537:         {
538:             copia = new Fracao (this);
539:         }
540:         catch (Exception e)
541:         {}
542:
543:         return copia;
544:     }
545: }
```

[C:\ExplJava\Expl\_09\TesteDeFracao.java]

```
1: class TesteDeFracao
2: {
3:     public static void main (String arg [])
4:     {
5:         Fracao f1, f2, f3, f4;
6:
7:         try
8:         {
9:             f2 = new Fracao (1,2);
10:            f3 = new Fracao (3,5);
11:            f4 = new Fracao ("3/5");
12:
13:            System.out.println ();
14:
15:            f1 = f2.mais(f3);
16:            System.out.println (f1 + " = " + f2 + " + " + f3);
17:
18:            f1 = f2.mais(7);
19:            System.out.println (f1 + " = " + f2 + " + " + 7);
20:
21:            f1 = f2.menos(f3);
```

```
22:         System.out.println (f1 + " = " + f2 + " - " + f3);
23:
24:         f1 = f2.menos(7);
25:         System.out.println (f1 + " = " + f2 + " - " + 7);
26:
27:         f1 = f2.vezes(f3);
28:         System.out.println (f1 + " = " + f2 + " * " + f3);
29:
30:         f1 = f2.vezes(7);
31:         System.out.println (f1 + " = " + f2 + " * " + 7);
32:
33:         f1 = f2.divididoPor(f3);
34:         System.out.println (f1 + " = " + f2 + " / " + f3);
35:
36:         f1 = f2.divididoPor(7);
37:         System.out.println (f1 + " = " + f2 + " / " + 7);
38:
39:         System.out.println (f2 + " == " + f2.valorReal());
40:
41:         if (f2.equals(f3))
42:             System.out.println (f2 + " == " + f3 + " (pelo equals)");
43:         else
44:             System.out.println (f2 + " != " + f3 + " (pelo equals)");
45:
46:         if (f3.equals(f4))
47:             System.out.println (f3 + " == " + f4 + " (pelo equals)");
48:         else
49:             System.out.println (f3 + " != " + f4 + " (pelo equals)");
50:
51:         int comp = f2.compareTo(f3);
52:         if (comp < 0)
53:             System.out.println (f2 + " < " + f3 + " (pelo compareTo)");
54:         else
55:             if (comp == 0)
56:                 System.out.println (f2 + " == " + f3 +
57:                                     " (pelo compareTo)");
58:             else
59:                 System.out.println (f2 + " > " + f3 +
60:                                     " (pelo compareTo)");
61:
62:         comp = f3.compareTo(f4);
63:         if (comp < 0)
64:             System.out.println (f3 + " < " + f4 + " (pelo compareTo)");
65:         else
66:             if (comp == 0)
67:                 System.out.println (f3 + " == " + f4 +
68:                                     " (pelo compareTo)");
69:             else
70:                 System.out.println (f3 + " > " + f4 +
71:                                     " (pelo compareTo)");
72:
73:         comp = f3.compareTo(f2);
74:         if (comp < 0)
75:             System.out.println (f3 + " < " + f2 + " (pelo compareTo)");
76:         else
77:             if (comp == 0)
```

```
78:             System.out.println (f3 + " == " + f2 +
79:                                 " (pelo compareTo)");
80:             else
81:                 System.out.println (f3 + " > " + f2 +
82:                                     " (pelo compareTo)");
83:
84:             System.out.println ("O codigo de espalhamento de " +
85:                                 "uma instancia valendo " + f2 +
86:                                     " vale " + f2.hashCode());
87:
88:             System.out.println ("O codigo de espalhamento de " +
89:                                 "uma instancia valendo " + f3 +
90:                                     " vale " + f3.hashCode());
91:
92:             System.out.println ("O codigo de espalhamento de " +
93:                                 "outra instancia valendo " + f4 +
94:                                     " vale " + f4.hashCode());
95:         }
96:     catch (Exception e)
97:     {
98:         System.err.println (e.getMessage ());
99:     }
100: }
101: }
```

Para compilar e executar este programa, daremos os seguintes comandos:

```
C:\ExplsJava\Expl_09> javac TesteDeFracao.java
C:\ExplsJava\Expl_09> java TesteDeFracao
```

Isto poderia produzir no console a seguinte interação:

```
11/10 = 1/2 + 3/5
15/2 = 1/2 + 7
-1/10 = 1/2 - 3/5
13/2 = 1/2 - 7
3/10 = 1/2 * 3/5
7/2 = 1/2 * 7
5/6 = 1/2 / 3/5
1/14 = 1/2 / 7
1/2 == 0.5
1/2 != 3/5 (pelo equals)
3/5 == 3/5 (pelo equals)
1/2 < 3/5 (pelo compareTo)
3/5 == 3/5 (pelo compareTo)
3/5 > 1/2 (pelo compareTo)
O codigo de espalhamento de uma instancia valendo 1/2 vale 1024
O codigo de espalhamento de uma instancia valendo 3/5 vale 1119
O codigo de espalhamento de outra instancia valendo 3/5 vale 1119
```

## Vetores

Um vetor é uma coleção de informações de mesma natureza. Vetores podem ter um número arbitrariamente grande de subscritos. Subscritos identificam de forma única um elemento dentro da coleção. Os subscritos em Java são sempre números naturais, sendo zero o primeiro deles.

A declaração de um vetor tem a seguinte forma: primeiramente vem o tipo dos elementos do vetor, em seguida vem o identificador do vetor, em seguida vem uma série pares de colchetes ([]), um para cada dimensão do vetor. Sendo Tipo um tipo e Vetor o identificador do vetor a ser declarado, temos abaixo a declaração de um vetor de elementos do tipo Tipo:

Tipo Vetor [ ][ ][ ]...[ ]                  ou                  Tipo [ ][ ][ ]...[ ] Vetor

Como acontece com os objetos, os vetores antes de serem usados precisam ser instanciados. Faz-se isso com o operador new. Sendo Vetor o identificador de um vetor do tipo Tipo previamente declarado com k dimensões e Ni números naturais, temos:

Vetor = new Tipo [N<sub>1</sub>][N<sub>2</sub>]...[N<sub>k</sub>];

Para acessar um elemento de um vetor, basta mencionar o seu nome e, entre colchetes o subscrito desejado.

### **Constantes Vetor**

Uma constante vetor consiste dos valores dos seus elementos separados por vírgulas (,) e delimitados por chaves ({}). Podem somente ser usadas para iniciá-lo (não se pode atribuir uma constante vetor a um vetor).

### **Tamanho das Dimensões de um Vetor**

O tamanho de cada uma das dimensões de um vetor pode ser obtido através da propriedade length. Assim, sendo Vetor o identificador de um vetor e i, j e k seus subscritos, temos que:

1. `Vetor.length` → Dá o tamanho da primeira dimensão do vetor;
  2. `Vetor[ i ].length` → Dá o tamanho da segunda dimensão do vetor;
-

3. vetor[ i ][ j ].length → Dá o tamanho da terceira dimensão do vetor;
4. vetor[ i ][ j ][ k ].length → Dá o tamanho da quarta dimensão do vetor;
5. Etc.

[C:\ExplsJava\Expl\_10\Agenda.java]

```
1: import java.util.regex.*;
2:
3: /**
4:  A classe Agenda representa uma simples agenda de telefone implementada tendo
5:  como base dois vetores que armazenam, respectivamente, os nomes e os telefones
6:  dos contatos da agenda.
7:  Instâncias desta classe permitem a realização das operações básicas de uma agenda.
8:  Nela encontramos, por exemplo, métodos para incluir, excluir e listar
9:  contatos.
10: @author André Luís dos Reis Gomes de Carvalho.
11: @since 2000.
12: */
13: public class Agenda implements Cloneable
14: {
15:     private static final String regExNom =
16:         "[A-Z][a-z]*(?: ([A-Z]|[a-z])[a-z]*)*$";
17:
18:     private static final Pattern padraoNom =
19:         Pattern.compile (Agenda.regExNom);
20:
21:     private static final String regExTel =
22:         "(?:\\(([0-9]{2})\\) )?([0-9]{4})-([0-9]{4})$";
23:
24:     private static final Pattern padraoTel =
25:         Pattern.compile (Agenda.regExTel);
26:
27:     /**
28:      Valida o nome de um contato.
29:      Verifica se o nome fornecido como parâmetro é um nome válido,
30:      lançando exceções, caso incorretudes sejam detectadas.
31:      @param nom o nome a ser validado.
32:      @throws Exception se não for fornecido um nome, ou se o nome fornecido
33:                         não parecer ser um nome correto.
34:     */
35:     public static void valideNome (String nom) throws Exception
36:     {
37:         if (nom==null)
38:             throw new Exception ("Nome não fornecido");
39:
40:         if (!Agenda.padraoNom.matcher(nom).matches())
41:             throw new Exception ("Nome inválido");
42:     }
43:
44:     /**
45:      Valida o telefone de um contato.
```

```
46:     Verifica se o número de telefone fornecido é um número
47:     nacional válido, lançando exceções, caso incorretudes sejam
48:     detectadas.
49:     @param tel o telefone a ser validado.
50:     @throws Exception se não for fornecido um telefone, ou se o telefone
51:                     fornecido não parecer ser um telefone correto.
52:     */
53:     public static void valideTelefone (String tel) throws Exception
54:     {
55:         if (tel==null)
56:             throw new Exception ("Telefone nao fornecido");
57:
58:         if (!Agenda.padraoTel.matcher(tel).matches())
59:             throw new Exception ("Telefone invalido");
60:     }
61:
62:     /**
63:      Expressa, em cada instante, a quantidade de contatos
64:      armazenados na agenda.
65:     */
66:     private int qtdContatos=0;
67:
68:     /**
69:      Mantém armazenados os nomes dos contatos armazenados
70:      na agenda.
71:     */
72:     private String[] nome;
73:
74:     /**
75:      Mantém armazenados os telefones dos contatos armazenados
76:      na agenda.
77:     */
78:     private String[] telefone;
79:
80:     /**
81:      Localiza um nome dado na agenda.
82:      Procura um contato na agenda, pelo método da busca sequencial,
83:      resultando um número inteiro negativo, quando o nome
84:      procurado não tiver sido encontrado, ou um numero inteiro
85:      positivo, quando o nome procurado tiver sido encontrado.
86:      @param nom o nome a ser procurado.
87:      @return um inteiro ao qual se deve dar a seguinte interpretação:
88:              <ol>
89:                  <li>
90:                      Um inteiro i negativo é retornado quando o nome
91:                      procurado não foi encontrado, mas, caso ele fosse ser
92:                      inserido, para manter os nomes da agenda em ordem
93:                      alfabetica, o local apropriado para a inserção seria
94:                      a posição -i-1.
95:                  </li>
96:                  <li>
97:                      Um inteiro i positivo é retornado quando o nome
98:                      procurado foi encontrado na posição i-1.
99:                  </li>
100:             </ol>
101:     */
```

```
102:     private int ondeEsta (String nom)
103:     {
104:         int pos;
105:
106:         for (pos=0; pos<this.qtdContatos; pos++)
107:             if (nom.equals(this.nome[pos]))
108:                 return pos+1;
109:
110:         return -(pos+1);
111:     }
112:
113:     /**
114:      Constrói uma nova instância da classe Agenda.
115:      Para tanto, deve ser fornecido um inteiro que será utilizado
116:      como capacidade da instância recém criada.
117:      @param cap o número inteiro a ser utilizado como capacidade.
118:      @throws Exception se a capacidade for negativa ou zero.
119:     */
120:     public Agenda (int cap)
121:         throws Exception
122:     {
123:         if (cap <= 0)
124:             throw new Exception ("Capacidade invalida");
125:
126:         this.nome    = new String [cap];
127:         this.telefone = new String [cap];
128:     }
129:
130:     /**
131:      Inclui um novo contato em uma agenda.
132:      Acrescenta um contato com o nome e telefone fornecidos na instância
133:      à qual este método for aplicado.
134:      @param nom o nome do novo contato.
135:      @param tel o telefone do novo contato.
136:      @throws Exception se não for fornecido um nome, ou se o nome fornecido
137:                      não parecer ser um nome correto, ou se não for fornecido
138:                      um telefone, ou se o telefone fornecido não parecer ser
139:                      um telefone correto, ou se a agenda estiver cheia, ou
140:                      ainda se o nome fornecido já estiver cadastrado.
141:     */
142:     public void registreContato (String nom,
143:                                 String tel)
144:         throws Exception
145:     {
146:         if (this.qtdContatos == this.nome.length)
147:             throw new Exception ("Agenda cheia");
148:
149:         Agenda.validaNome (nom);
150:         Agenda.validaTelefone (tel);
151:
152:         int posicao = this.ondeEsta (nom);
153:
154:         if (posicao > 0)
155:             throw new Exception ("Nome ja registrado");
156:
157:         posicao = (-posicao)-1;
```

```
158:
159:         for (int pos=this.qtdContatos-1; pos>=posicao; pos--)
160:         {
161:             this.nome [pos+1] = this.nome [pos];
162:             this.telefone [pos+1] = this.telefone [pos];
163:         }
164:
165:         this.nome [posicao] = nom;
166:         this.telefone [posicao] = tel;
167:
168:         this.qtdContatos++;
169:     }
170:
171: /**
172: Remove um contato, dado seu nome.
173: Exclui da instância à qual este método for aplicado o contato
174: cujo nome foi fornecido.
175: @param nom o nome do contato a ser descartado.
176: @throws Exception se não for fornecido um nome, ou se o nome fornecido
177:                   não parecer ser um nome correto, ou se a agenda
178:                   estiver vazia, ou ainda se a agenda não contiver um
179:                   contato com o nome fornecido.
180: */
181: public void descarteContato (String nom)
182:                     throws Exception
183: {
184:     if (this.qtdContatos == 0)
185:         throw new Exception ("Agenda vazia");
186:
187:     Agenda.validaNome (nom);
188:
189:     if (!Agenda.padraoNom.matcher(nom).matches())
190:         throw new Exception ("Nome invalido");
191:
192:     int posicao = this.ondeEsta (nom);
193:
194:     if (posicao < 0)
195:         throw new Exception ("Nome inexistente");
196:
197:     posicao--;
198:
199:     int pos;
200:
201:     for (pos = posicao; pos < this.qtdContatos - 1; pos++)
202:     {
203:         this.nome [pos] = this.nome [pos+1];
204:         this.telefone [pos] = this.telefone [pos+1];
205:     }
206:
207:     this.nome [pos] = null;
208:     this.telefone [pos] = null;
209:
210:     this.qtdContatos--;
211: }
212:
213: /**
```

```
214:     Gera uma representação textual de todo conteúdo da agenda.
215:     Produz e resulta um String com todos os nomes e telefones contidos
216:     na agenda.
217:     @return um String contendo todo o conteúdo da agenda.
218:     */
219:     public String toString ()
220:     {
221:         String ret = "";
222:
223:         for (int pos=0; pos<this.qtdContatos; pos++)
224:             ret += this.nome[pos] + " (" + this.telefone[pos] + ") \n";
225:
226:         return ret;
227:     }
228:
229:     /**
230:      Verifica a igualdade entre duas agendas.
231:      Verifica se o Object fornecido como parâmetro representa uma
232:      agenda igual àquela representada pela instância à qual este
233:      método for aplicado, resultando true em caso afirmativo,
234:      ou false, caso contrário.
235:      @param obj o objeto a ser comparado com a instância à qual esse método
236:              for aplicado.
237:      @return true, caso o Object fornecido ao método e a instância chamante do
238:              método representarem agendas iguais, ou false, caso contrário.
239:      */
240:     public boolean equals (Object obj)
241:     {
242:         if (this == obj)
243:             return true;
244:
245:         if (obj == null)
246:             return false;
247:
248:         if (this.getClass() != obj.getClass())
249:             return false;
250:
251:         Agenda agenda = (Agenda) obj;
252:
253:         if (this.qtdContatos != agenda.qtdContatos)
254:             return false;
255:
256:         for (int pos=0; pos<this.qtdContatos; pos++)
257:             if (!this.nome [pos].equals(agenda.nome [pos]) ||
258:                 !this.telefone[pos].equals(agenda.telefone[pos]))
259:                 return false;
260:
261:         return true;
262:     }
263:
264:     /**
265:      Calcula o código de espalhamento (ou código de hash) de uma agenda.
266:      Calcula e resulta o código de espalhamento (ou código de hash, ou ainda o
267:      hashCode) da agenda representada pela instância à qual o método for aplicado.
268:      @return o código de espalhamento da agenda chamante do método.
269:      */
```

```
270:     public int hashCode ()
271:     {
272:         int ret = super.hashCode();
273:
274:         ret = 13*ret + this.qtdContatos;
275:
276:         for (int pos=0; pos<this.qtdContatos; pos++)
277:         {
278:             ret = 13*ret + this.nome [pos].hashCode();
279:             ret = 13*ret + this.telefone[pos].hashCode();
280:         }
281:
282:         return ret;
283:     }
284:
285:     /**
286:      Constroi uma cópia da instância da classe Agenda dada.
287:      Para tanto, deve ser fornecida uma instancia da classe Agenda para ser
288:      utilizada como modelo para a construção da nova instância criada.
289:      @param modelo a instância da classe Agenda a ser usada como modelo.
290:      @throws Exception se o modelo for null.
291:     */
292:     public Agenda (Agenda modelo)
293:             throws Exception
294:     {
295:         if (modelo==null)
296:             throw new Exception ("Modelo nao fornecido");
297:
298:         this.nome      = new String [modelo.nome      .length];
299:         this.telefone = new String [modelo.telefone.length];
300:
301:         this.qtdContatos = modelo.qtdContatos;
302:
303:         for (int pos=0; pos<this.qtdContatos; pos++)
304:         {
305:             this.nome      [pos] = modelo.nome      [pos];
306:             this.telefone [pos] = modelo.telefone [pos];
307:         }
308:     }
309:
310:     /**
311:      Clona uma agenda.
312:      Produz e resulta uma cópia da agenda representada pela instância
313:      à qual o método for aplicado.
314:      @return a cópia da agenda representada pela instância à qual
315:              o método for aplicado.
316:     */
317:     public Object clone ()
318:     {
319:         Agenda copia=null;
320:
321:         try
322:         {
323:             copia = new Agenda (this);
324:         }
325:         catch (Exception e)
```

```
326:     {}
327:
328:     return copia;
329: }
330: }
```

### [C:\ExplsJava\Expl\_10\TesteDeAgenda.java]

```
1: import java.io.IOException;
2: import java.io.BufferedReader;
3: import java.io.InputStreamReader;
4:
5: public class TesteDeAgenda
6: {
7:     public static void main (String[] args)
8:     {
9:         BufferedReader entrada = new BufferedReader
10:            (new InputStreamReader
11:             (System.in)));
12:
13:         Agenda agenda = null;
14:
15:         for (;;)
16:         {
17:             System.out.print ("Capacidade desejada para a Agenda: ");
18:             try
19:             {
20:                 int cap = Integer.parseInt (entrada.readLine ());
21:                 agenda = new Agenda (cap);
22:                 break;
23:             }
24:             catch (IOException e)
25:             {}
26:             catch (NumberFormatException e)
27:             {
28:                 System.err.println ("Nao foi digitado um numero inteiro\n\n");
29:             }
30:             catch (Exception e)
31:             {
32:                 System.err.println (e.getMessage () + "\n\n");
33:             }
34:         }
35:
36:         System.out.println ();
37:
38:         String nome = null, telefone = null;
39:
40:         char opcao = ' ';
41:
42:         do
43:         {
44:             System.out.println ();
45:
46:             System.out.print ("Digite sua Opcão (" +
47:                               "I=Incluir/" +
```

```
48:                     "E=Excluir/" +
49:                     "L=Listar/" +
50:                     "S=Sair)" +
51:                     ": ");
52:
53:             try
54:             {
55:                 String str = entrada.readLine ();
56:
57:                 if (str.length()==1)
58:                     opcao = str.charAt(0);
59:                 else
60:                     opcao = 'A'; // forçando opção inválida
61:             }
62:             catch (IOException e)
63:             {}
64:
65:             switch (opcao)
66:             {
67:                 case 'i':
68:                 case 'I':
69:                     try
70:                     {
71:                         System.out.print ("Nome....: ");
72:                         nome = entrada.readLine ();
73:                     }
74:                     catch (IOException e)
75:                     {}
76:
77:                     try
78:                     {
79:                         System.out.print ("Telefone: ");
80:                         telefone = entrada.readLine ();
81:                     }
82:                     catch (IOException e)
83:                     {}
84:
85:                     try
86:                     {
87:                         agenda.registreContato (nome, telefone);
88:                     }
89:                     catch (Exception e)
90:                     {
91:                         System.err.println (e.getMessage ());
92:                     }
93:
94:                     System.out.println ();
95:                     break;
96:
97:
98:                 case 'e':
99:                 case 'E':
100:                     System.out.print ("Nome: ");
101:                     try
102:                     {
103:                         nome = entrada.readLine ();
```

```
104:        }
105:        catch (IOException e)
106:        {
107:
108:            try
109:            {
110:                agenda.descarteContato (nome);
111:            }
112:            catch (Exception e)
113:            {
114:                System.err.println (e.getMessage ());
115:            }
116:
117:            System.out.println ();
118:            break;
119:
120:
121:            case 'l':
122:            case 'L':
123:                System.out.println (agenda);
124:                break;
125:
126:
127:            case 's':
128:            case 'S':
129:                break;
130:
131:
132:            default :
133:                System.err.println ("Opção invalida");
134:                System.err.println ();
135:            }
136:        }
137:        while ((opcao != 's') && (opcao != 'S'));
138:    }
139: }
```

Para compilar e executar este programa, daremos os seguintes comandos:

```
C:\ExplsJava\Expl_10> javac *.java.java
C:\ExplsJava\Expl_10> java TesteDeAgenda
```

Isto poderia produzir no console a seguinte interação:

```
Capacidade desejada para a Agenda: -10
Capacidade invalida
```

```
Capacidade desejada para a Agenda: 0
Capacidade invalida
```

Capacidade desejada para a Agenda: 10

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): i

Nome....: Jose

Telefone: 3241-4466

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): i

Nome....: Jose

Telefone: 3241-4466

Nome já registrado

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): i

Nome....: Joao

Telefone: 3252-9955

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): l

Jose (3241-4466)

Joao (3252-9955)

Digite sua Opcão (I=Incluir/C=Consultar/E=Excluir/S=Sair): e

Nome: Joao

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): l

Jose (3241-4466)

Digite sua Opcão (I=Incluir/C=Consultar/E=Excluir/S=Sair): a

Opcão invalida

Digite sua Opcão (I=Incluir/C=Consultar/E=Excluir/S=Sair): s

## Herança

Um programa pode muitas vezes conter grupos de classes relacionadas (semelhantes, mas não idênticas). Com o conceito de herança, grupos de classes relacionadas podem ser criados de maneira simples, flexível, eficiente e elegante.

---

Características comuns são agrupadas em uma classe (classe base). Desta classe poderíamos derivar outras classes (classes derivadas), e destas ainda outras classes, e assim por diante. Classes derivadas herdam todos os membros da classe base. Elas podem acrescentar novos membros àqueles herdados, bem como redefinir aqueles membros.

Posto que classes derivadas herdam as características de sua classe base, a descrição de uma classe derivada deve se concentrar nas diferenças que esta apresenta com relação a classe base, adicionando ou modificando características.

As funções acrescentadas por uma classe derivada somente podem ser usadas a partir de objetos da classe derivada. As funções definidas numa classe base podem ser usadas em objetos da classe base ou em objetos de qualquer de suas classes derivadas.

Para redefinir uma função membro de sua classe base, a classe derivada deve implementar uma função membro de mesmo nome e com parâmetros do mesmo tipo. A redefinição não será possível se a função membro da classe base estiver qualificada pelo modificador final.

No caso de uma classe precisar chamar um construtor de sua classe base na redefinição de seu construtor, ela poderá fazê-lo através da palavra chave super. Em vez ser utilizado o nome da classe para chamar o construtor, seria utilizada a palavra super eventualmente seguida dos argumentos a serem passados ao construtor separados por vírgulas (,) e entre parênteses ()).

Classes qualificadas como final não podem ser usadas como base para a derivação de outra classe.

## **Mais sobre Conversões de Tipo**

Conversões de tipo que transformam um objeto de uma dada subclasse em uma de suas superclasses podem acontecer implícita ou explicitamente e são completamente confiáveis.

Conversões de tipo que transformam um objeto de uma dada superclasse em uma de suas subclasses não são completamente confiáveis e somente podem ocorrer explicitamente.

---

Conversões de tipo transformam um objeto de uma dada classe em outra classe que não está em linha direta de ascendência ou descendência em uma mesma hierarquia de classes não são permitidas.

## ***Verificação de Tipo em Java***

Em Java, sempre que um objeto de um certo tipo é esperado, é possível usar um objeto daquele mesmo tipo, ou então, em seu lugar, um objeto de qualquer classe derivada daquele tipo.

Esta característica da linguagem permite a criação de código polimórfico, o que permite a programação adequada tanto funções que precisam trabalhar especificamente com uma certa classe derivada, quanto funções que precisam trabalhar genericamente com qualquer classe da hierarquia.

A função a ser executada quando de uma chamada é determinada com base no tipo do objeto ao qual a chamada foi vinculada, no nome usado na chamada e nos tipos dos parâmetros fornecidos.

Se o tipo do objeto declarado não bate com o tipo do objeto fornecido, então, em tempo de execução, o objeto fornecido será examinado de modo a determinar qual função deve ser executada. Chamamos a isto Binding Dinâmico.

## ***Membros Protegidos***

Conforme sabemos, existem 3 tipos de membros que podem ser definidos em uma classe, a saber: os membros públicos (que são acessíveis para qualquer método que tenha acesso à classe em que foram definidos), os membros privativos (que são acessíveis somente nos métodos definidos em sua classe) e os membros default (ou de pacote, que são acessíveis para todas os métodos definidos nas classes que integram o pacote onde foi definida sua classe).

Ficaremos sabendo agora da existência de um quarto e último tipo de membro, os membros protegidos. Para declarar um membro protegido, basta preceder a declaração do membro com a palavra `protected` (em vez de usar a palavra `public` ou a palavra `private`).

---

Membros protegidos, além de serem, como os membros default (ou de pacote), acessíveis para todas os métodos definidos nas classes que integram o pacote onde foi definida sua classe, são também acessíveis para métodos definidos em uma classe derivada da classe onde foram definidos.

Convém ressaltar que, no caso da classe base e da classe derivada residirem em pacotes diferentes, a classe derivada somente terá acesso aos membros protegidos de sua classe e de instâncias de sua própria classe, não tendo acesso aos membros protegidos de sua classe base ou de instâncias de sua classe base.

[C:\ExplsJava\Expl\_11\agenda\Agenda.java]

```
1: package agenda;
2: import java.util.regex.*;
3:
4: /**
5:  A classe Agenda representa uma simples agenda de telefone implementada tendo
6:  como base dois vetores que armazenam, respectivamente, os nomes e os telefones
7:  dos contatos da agenda.
8:  Instâncias desta classe permitem a realização das operações básicas de uma agenda.
9:  Nela encontramos, por exemplo, métodos para incluir, excluir e listar
10: contatos.
11: @author André Luís dos Reis Gomes de Carvalho.
12: @since 2000.
13: */
14: public class Agenda implements Cloneable
15: {
16:     private static final String regExNom =
17:         "[A-Z][a-z]*(?: ([A-Z]|[a-z])[a-z]*)*$";
18:
19:     private static final Pattern padraoNom =
20:         Pattern.compile (Agenda.regExNom);
21:
22:     private static final String regExTel =
23:         "(?:\\(([0-9]{2})\\) )?9?[0-9]{4}-[0-9]{4}$";
24:
25:     private static final Pattern padraoTel =
26:         Pattern.compile (Agenda.regExTel);
27:
28:     /**
29:      Valida o nome de um contato.
30:      Verifica se o nome fornecido como parâmetro é um nome válido,
31:      lançando exceções, caso incorretudes sejam detectadas.
32:      @param nom o nome a ser validado.
33:      @throws Exception se não for fornecido um nome, ou se o nome fornecido
34:                         não parecer ser um nome correto.
35:     */
36:     protected static void valideNome (String nom) throws Exception
37:     {
```

```
38:         if (nom==null)
39:             throw new Exception ("Nome nao fornecido");
40:
41:         if (!Agenda.padraoNom.matcher(nom).matches())
42:             throw new Exception ("Nome invalido");
43:     }
44:
45:     /**
46:      Valida o telefone de um contato.
47:      Verifica se o número de telefone fornecido é um número
48:      nacional válido, lançando exceções, caso incorretudes sejam
49:      detectadas.
50:      @param tel o telefone a ser validado.
51:      @throws Exception se não for fornecido um telefone, ou se o telefone
52:                      fornecido não parecer ser um telefone correto.
53:     */
54:     protected static void valideTelefone (String tel) throws Exception
55:     {
56:         if (tel==null)
57:             throw new Exception ("Telefone nao fornecido");
58:
59:         if (!Agenda.padraoTel.matcher(tel).matches())
60:             throw new Exception ("Telefone invalido");
61:     }
62:
63:     /**
64:      Expressa, em cada instante, a quantidade de contatos
65:      armazenados na agenda.
66:     */
67:     protected int qtdContatos=0;
68:
69:     /**
70:      Mantém armazenados os nomes dos contatos armazenados
71:      na agenda.
72:     */
73:     protected String[] nome;
74:
75:     /**
76:      Mantém armazenados os telefones dos contatos armazenados
77:      na agenda.
78:     */
79:     protected String[] telefone;
80:
81:     /**
82:      Localiza um nome dado na agenda.
83:      Procura um contato na agenda, pelo método da busca sequencial,
84:      resultando um número inteiro negativo, quando o nome
85:      procurado não tiver sido encontrado, ou um numero inteiro
86:      positivo, quando o nome procurado tiver sido encontrado.
87:      @param nom o nome a ser procurado.
88:      @return um inteiro ao qual se deve dar a seguinte interpretação:
89:              <ol>
90:                  <li>
91:                      Um inteiro i negativo é retornado quando o nome
92:                      procurado não foi encontrado, mas, caso ele fosse ser
93:                      inserido, para manter os nomes da agenda em ordem
```

```
94:             alfabética, o local apropriado para a inserção seria
95:             a posição -i-1.
96:         </li>
97:         <li>
98:             Um inteiro i positivo é retornado quando o nome
99:             procurado foi encontrado na posição i-1.
100:            </li>
101:        </ol>
102:    */
103:    protected int ondeEsta (String nom)
104:    {
105:        int pos;
106:
107:        for (pos=0; pos<this.qtdContatos; pos++)
108:            if (nom.equals(this.nome[pos]))
109:                return pos+1;
110:
111:        return -(pos+1);
112:    }
113:
114:    /**
115:     Constroi uma nova instância da classe Agenda.
116:     Para tanto, deve ser fornecido um inteiro que será utilizado
117:     como capacidade da instância recém criada.
118:     @param cap o número inteiro a ser utilizado como capacidade.
119:     @throws Exception se a capacidade for negativa ou zero.
120:    */
121:    public Agenda (int cap)
122:        throws Exception
123:    {
124:        if (cap <= 0)
125:            throw new Exception ("Capacidade invalida");
126:
127:        this.nome = new String [cap];
128:        this.telefone = new String [cap];
129:    }
130:
131:    /**
132:     Inclui um novo contato em uma agenda.
133:     Acrescenta um contato com o nome e telefone fornecidos na instância
134:     à qual este método for aplicado.
135:     @param nom o nome do novo contato.
136:     @param tel o telefone do novo contato.
137:     @throws Exception se não for fornecido um nome, ou se o nome fornecido
138:                     não parecer ser um nome correto, ou se não for fornecido
139:                     um telefone, ou se o telefone fornecido não parecer ser
140:                     um telefone correto, ou se a agenda estiver cheia, ou
141:                     ainda se o nome fornecido já estiver cadastrado.
142:    */
143:    public void registreContato (String nom,
144:                                String tel)
145:        throws Exception
146:    {
147:        if (this.qtdContatos == this.nome.length)
148:            throw new Exception ("Agenda cheia");
149:
```

```
150:         Agenda.validaNome      (nom);
151:         Agenda.validaTelefone (tel);
152:
153:         int posicao = this.ondeEsta (nom);
154:
155:         if (posicao > 0)
156:             throw new Exception ("Nome ja registrado");
157:
158:         posicao = (-posicao)-1;
159:
160:         for (int pos=this.qtdContatos-1; pos>=posicao; pos--)
161:         {
162:             this.nome      [pos+1] = this.nome      [pos];
163:             this.telefone [pos+1] = this.telefone [pos];
164:         }
165:
166:         this.nome      [posicao] = nom;
167:         this.telefone [posicao] = tel;
168:
169:         this.qtdContatos++;
170:     }
171:
172:     /**
173:      Remove um contato, dado seu nome.
174:      Exclui da instância à qual este método for aplicado o contato
175:      cujo nome foi fornecido.
176:      @param nom o nome do contato a ser descartado.
177:      @throws Exception se não for fornecido um nome, ou se o nome fornecido
178:                         não parecer ser um nome correto, ou se a agenda
179:                         estiver vazia, ou ainda se a agenda não contiver um
180:                         contato com o nome fornecido.
181:     */
182:     public void descarteContato (String nom)
183:                               throws Exception
184:     {
185:         if (this.qtdContatos == 0)
186:             throw new Exception ("Agenda vazia");
187:
188:         Agenda.validaNome (nom);
189:
190:         if (!Agenda.padraoNom.matcher(nom).matches())
191:             throw new Exception ("Nome invalido");
192:
193:         int posicao = this.ondeEsta (nom);
194:
195:         if (posicao < 0)
196:             throw new Exception ("Nome inexistente");
197:
198:         posicao--;
199:
200:         int pos;
201:
202:         for (pos = posicao; pos < this.qtdContatos - 1; pos++)
203:         {
204:             this.nome      [pos] = this.nome      [pos+1];
205:             this.telefone [pos] = this.telefone [pos+1];
```

```
206:        }
207:
208:        this.nome [pos] = null;
209:        this.telefone [pos] = null;
210:
211:        this.qtdContatos--;
212:    }
213:
214:    /**
215:     * Obtem o nome do contato com número de ordem dado.
216:     * Resulta o nome do contato cujo número de ordem foi fornecido.
217:     * @return o nome do contato desejado
218:     * @throws Exception quando a agenda estiver vazia ou então quando
219:             o número de ordem fornecido estiver fora dos
220:             limites
221:    */
222:    public String getNome (int pos) throws Exception
223:    {
224:        if (this.qtdContatos == 0)
225:            throw new Exception ("Agenda vazia");
226:
227:        if (pos < 0 || pos >= this.qtdContatos)
228:            throw new Exception ("Nome fora dos limites");
229:
230:        return this.nome [pos];
231:    }
232:
233:    /**
234:     * Obtem o telefone do contato com número de ordem dado.
235:     * Resulta o telefone do contato cujo número de ordem foi fornecido.
236:     * @return o telefone do contato desejado
237:     * @throws Exception quando a agenda estiver vazia ou então quando
238:             o número de ordem fornecido estiver fora dos
239:             limites
240:    */
241:    public String getTelefone (int pos) throws Exception
242:    {
243:        if (this.qtdContatos == 0)
244:            throw new Exception ("Agenda vazia");
245:
246:        if (pos < 0 || pos >= this.qtdContatos)
247:            throw new Exception ("Telefone fora dos limites");
248:
249:        return this.telefone [pos];
250:    }
251:
252:    /**
253:     * Obtem a capacidade de uma agenda.
254:     * Resulta a capacidade da instância à qual este método for aplicado.
255:     * @return a capacidade da agenda chamante do método.
256:    */
257:    public int getCapacidade ()
258:    {
259:        return this.nome.length;
260:    }
261:
```

```
262:    /**
263:     Obtém a quantidade de contatos cadastrados em uma agenda.
264:     Resulta a quantidade de contatos armazenados na instância à qual este
265:     método for aplicado.
266:     @return a quantidade de contatos da agenda chamante do método.
267:    */
268:    public int getQtdContatos ()
269:    {
270:        return this.qtdContatos;
271:    }
272:
273:    /**
274:     Verifica a presença de um contato em uma agenda.
275:     Resulta true, caso um contato com o nome fornecido estiver
276:     cadastrado na instância à qual este método for aplicado, ou false,
277:     caso contrário.
278:     @return a indicação da presença na agenda do contato procurado.
279:    */
280:    public boolean haRegistroDoContato (String nom)
281:    {
282:        return this.ondeEsta(nom)>0;
283:    }
284:
285:    /**
286:     Gera uma representação textual de todo conteúdo da agenda.
287:     Produz e resulta um String com todos os nomes e telefones contidos
288:     na agenda.
289:     @return um String contendo todo o conteúdo da agenda.
290:    */
291:    public String toString ()
292:    {
293:        String ret = "";
294:
295:        for (int pos=0; pos<this.qtdContatos; pos++)
296:            ret += this.nome[pos] + " (" + this.telefone[pos] + ")\n";
297:
298:        return ret;
299:    }
300:
301:    /**
302:     Verifica a igualdade entre duas agendas.
303:     Verifica se o Object fornecido como parâmetro representa uma
304:     agenda igual àquela representada pela instância à qual este
305:     método for aplicado, resultando true em caso afirmativo,
306:     ou false, caso contrário.
307:     @param obj o objeto a ser comparado com a instância à qual esse método
308:             for aplicado.
309:     @return true, caso o Object fornecido ao método e a instância chamante do
310:             método representarem agendas iguais, ou false, caso contrário.
311:    */
312:    public boolean equals (Object obj)
313:    {
314:        if (this == obj)
315:            return true;
316:
317:        if (obj == null)
```

```
318:         return false;
319:
320:         if (this.getClass() != obj.getClass())
321:             return false;
322:
323:         Agenda agenda = (Agenda) obj;
324:
325:         if (this.qtdContatos != agenda.qtdContatos)
326:             return false;
327:
328:         for (int pos=0; pos<this.qtdContatos; pos++)
329:             if (!this.nome [pos].equals(agenda.nome [pos]) ||
330:                 !this.telefone[pos].equals(agenda.telefone[pos]))
331:                 return false;
332:
333:         return true;
334:     }
335:
336:     /**
337:      Calcula o código de espalhamento (ou código de hash) de uma agenda.
338:      Calcula e resulta o código de espalhamento (ou código de hash, ou ainda o
339:      hashCode) da agenda representada pela instância à qual o método for aplicado.
340:      @return o código de espalhamento da agenda chamante do método.
341:     */
342:     public int hashCode ()
343:     {
344:         int ret = super.hashCode();
345:
346:         ret = 13*ret + this.qtdContatos;
347:
348:         for (int pos=0; pos<this.qtdContatos; pos++)
349:         {
350:             ret = 13*ret + this.nome [pos].hashCode();
351:             ret = 13*ret + this.telefone[pos].hashCode();
352:         }
353:
354:         return ret;
355:     }
356:
357:     /**
358:      Constroi uma cópia da instância da classe Agenda dada.
359:      Para tanto, deve ser fornecida uma instancia da classe Agenda para ser
360:      utilizada como modelo para a construção da nova instância criada.
361:      @param modelo a instância da classe Agenda a ser usada como modelo.
362:      @throws Exception se o modelo for null.
363:     */
364:     public Agenda (Agenda modelo)
365:             throws Exception
366:     {
367:         if (modelo==null)
368:             throw new Exception ("Modelo nao fornecido");
369:
370:         this.nome      = new String [modelo.nome      .length];
371:         this.telefone = new String [modelo.telefone.length];
372:
373:         this.qtdContatos = modelo.qtdContatos;
```

```

374:
375:         for (int pos=0; pos<this.qtdContatos; pos++)
376:         {
377:             this.nome [pos] = modelo.nome [pos];
378:             this.telefone[pos] = modelo.telefone[pos];
379:         }
380:     }
381:
382: /**
383: Clona uma agenda.
384: Produz e resulta uma cópia da agenda representada pela instância
385: à qual o método for aplicado.
386: @return a cópia da agenda representada pela instância à qual
387:         o método for aplicado.
388: */
389: public Object clone ()
390: {
391:     Agenda copia=null;
392:
393:     Try
394:     {
395:         copia = new Agenda (this);
396:     }
397:     catch (Exception e)
398:     {}
399:
400:     return copia;
401: }
402: }
```

[C:\ExplsJava\Expl\_11\agenda\consultavel\AgendaConsultavel.java]

```

1: package agenda.consultavel;
2: import agenda.*;
3:
4: /**
5: A classe AgendaConsultavel representa uma simples agenda de telefone
6: possível de ser consultada.
7:
8: Instâncias desta classe permitem a realização das operações básicas de uma agenda.
9: Nela encontramos, não só, métodos para incluir, excluir e listar
10: contatos, mas também métodos que permitem consultá-la.
11: @author André Luís dos Reis Gomes de Carvalho.
12: @since 2000.
13: */
14: public class AgendaConsultavel extends Agenda
15: {
16:     /**
17:     Constroi uma nova instância da classe AgendaConsultavel.
18:     Para tanto, deve ser fornecido um inteiro que será utilizado
19:     como capacidade da instância recém criada.
20:     @param cap o número inteiro a ser utilizado como capacidade.
21:     @throws Exception se a capacidade for negativa ou zero.
22:     */
23:     public AgendaConsultavel (int cap) throws Exception
```

```
24: {
25:     super (cap); // invoca o construtor da classe Agenda
26: }
27:
28: /**
29: Obtém o telefone de um dado nome.
30: Resulta o telefone do contato cujo nome foi fornecido.
31: @return o telefone do contato desejado.
32: @throws Exception se não for fornecido um nome, ou se o nome fornecido
33:                 não parecer ser um nome correto, ou se a agenda
34:                 estiver vazia, ou ainda se a agenda não contiver um
35:                 contato com o nome fornecido.
36: */
37: public String getTelefone (String nom)
38:                     throws Exception
39: {
40:
41:
42:     if (this.qtdContatos == 0)
43:         throw new Exception ("Agenda vazia");
44:
45:     int posicao = this.ondeEsta (nom);
46:
47:     if (posicao < 0)
48:         throw new Exception ("Nome inexistente");
49:
50:     posicao--;
51:
52:     return this.telefone [posicao];
53: }
54:
55: /**
56: Constroi uma cópia da instância da classe AgendaConsultavel dada.
57: Para tanto, deve ser fornecida uma instancia da classe AgendaConsultavel
58: para ser utilizada como modelo para a construção da nova instância
59: criada.
60: @param modelo a instância da classe AgendaConsultavel a ser usada como
61:             modelo.
62: @throws Exception se o modelo for null.
63: */
64: public AgendaConsultavel (AgendaConsultavel modelo) throws Exception
65: {
66:     super (modelo);
67: }
68: }
```

[C:\ExplsJava\Expl\_11\TesteDeAgenda.java]

```
1: import java.io.IOException;
2: import java.io.BufferedReader;
3: import java.io.InputStreamReader;
4: import agenda.consultavel.*;
5:
6: public class TesteDeAgenda
7: {
```

```
8:     public static void main (String[] args)
9:     {
10:         BufferedReader entrada = new BufferedReader
11:             (new InputStreamReader
12:                 (System.in));
13:
14:         AgendaConsultavel agenda = null;
15:
16:         for (;)
17:         {
18:             System.out.print ("Capacidade desejada para a Agenda: ");
19:             try
20:             {
21:                 int cap = Integer.parseInt (entrada.readLine ());
22:                 agenda = new AgendaConsultavel (cap);
23:                 break;
24:             }
25:             catch (IOException e)
26:             {}
27:             catch (NumberFormatException e)
28:             {
29:                 System.err.println ("Nao foi digitado um numero inteiro\n\n");
30:             }
31:             catch (Exception e)
32:             {
33:                 System.err.println (e.getMessage () +"\n\n");
34:             }
35:         }
36:
37:         System.out.println ();
38:
39:         String nome = null, telefone = null;
40:
41:         char opcao = ' ';
42:
43:         do
44:         {
45:             System.out.println ();
46:
47:             System.out.print ("Digite sua Opcão (" +
48:                             "I=Incluir/" +
49:                             "E=Excluir/" +
50:                             "C=Consultar/" +
51:                             "L=Listar/" +
52:                             "S=Sair)" +
53:                             ": ");
54:
55:             try
56:             {
57:                 String str = entrada.readLine ();
58:
59:                 if (str.length ()==1)
60:                     opcao = str.charAt (0);
61:                 else
62:                     opcao = 'A'; // forçando opção inválida
63:             }
```

```
64:         catch (IOException e)
65:         {}
66:
67:         switch (opcao)
68:         {
69:             case 'i':
70:             case 'I':
71:                 try
72:                 {
73:                     System.out.print ("Nome....: ");
74:                     nome = entrada.readLine ();
75:                 }
76:                 catch (IOException e)
77:                 {}
78:
79:                 try
80:                 {
81:                     System.out.print ("Telefone: ");
82:                     telefone = entrada.readLine ();
83:                 }
84:                 catch (IOException e)
85:                 {}
86:
87:                 try
88:                 {
89:                     agenda.registreContato (nome, telefone);
90:                 }
91:                 catch (Exception e)
92:                 {}
93:                     System.err.println (e.getMessage ());
94:                 }
95:
96:                 System.out.println ();
97:                 break;
98:
99:
100:            case 'e':
101:            case 'E':
102:                System.out.print ("Nome: ");
103:                try
104:                {
105:                    nome = entrada.readLine ();
106:                }
107:                catch (IOException e)
108:                {}
109:
110:                try
111:                {
112:                    agenda.descarteContato (nome);
113:                }
114:                catch (Exception e)
115:                {}
116:                    System.err.println (e.getMessage ());
117:                }
118:
119:                System.out.println ();
```

```
120:                     break;
121:
122:
123:             case 'c':
124:             case 'C':
125:                 try
126:                 {
127:                     System.out.print ("Nome....: ");
128:                     nome = entrada.readLine ();
129:                 }
130:                 catch (IOException e)
131:                 {}
132:
133:                 try
134:                 {
135:                     telefone = agenda.getTelefone (nome);
136:                 }
137:                 catch (Exception e)
138:                 {
139:                     System.err.println (e.getMessage ());
140:                     System.err.println ();
141:                     break;
142:                 }
143:
144:                     System.out.println ("Telefone: " + telefone);
145:                     System.out.println ();
146:                     break;
147:
148:
149:             case 'l':
150:             case 'L':
151:                 System.out.println (agenda);
152:                 break;
153:
154:
155:
156:             case 's':
157:             case 'S':
158:                 break;
159:
160:
161:             default :
162:                 System.err.println ("Opção invalida");
163:                 System.err.println ();
164:             }
165:         }
166:         while ((opcao != 's') && (opcao != 'S'));
167:     }
168: }
```

Para compilar e executar este programa, daremos os seguintes comandos:

```
C:\ExplsJava\Expl_11> javac *.java
C:\ExplsJava\Expl_11> java TesteDeAgenda
```

Isto poderia produzir no console a seguinte interação:

Capacidade desejada para a Agenda: -10  
Capacidade invalida

Capacidade desejada para a Agenda: 0  
Capacidade invalida

Capacidade desejada para a Agenda: 10

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): i  
Nome....: Jose  
Telefone: 3241.4466

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): i  
Nome....: Jose  
Nome já registrado

Digite sua Opcão (I=Incluir/C=Consultar/L=Listar/E=Excluir/S=Sair): c  
Nome....: Jose  
Telefone: 3241.4466

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): i  
Nome....: Joao  
Telefone: 3252.9955

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): l  
Jose (3241.4466)  
Joao (3252.9955)

Digite sua Opcão (I=Incluir/C=Consultar/E=Excluir/S=Sair): e  
Nome: Joao

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): l  
Jose (3241.4466)

---

Digite sua Opcão (I=Incluir/C=Consultar/E=Excluir/S=Sair): a  
Opcão invalida

Digite sua Opcão (I=Incluir/C=Consultar/E=Excluir/S=Sair): s

## A Classe Vector

Esta classe implementa um repositório de instâncias de uma dada classe. Tais instâncias podem ser identificadas posicionalmente, sendo zero a posição da primeira delas.

Tal repositório cresce dinamicamente a medida que mais instâncias vão sendo armazenadas nele. Para otimizar o processo de crescimento, o repositório cresce sempre de várias unidades de armazenamento, e não apenas de uma. Ajustar a capacidade do repositório antes de inserir nele uma grande quantidade de objetos é uma boa ideia para minimizar o tempo gasto com realocações incrementais.

Classes que fazem uso desta classe em sua implementação, devem ter logo no início do arquivo onde forem definidas a seguinte diretiva:

import java.util.Vector;

ou

import java.util.\*;

Veja na documentação da linguagem a interface que a classe especifica para comunicação com ela própria e com suas instâncias.

## Interfaces

Interfaces são, em certo sentido, muito parecidas com classes. Podemos pensar que interfaces sejam tão simplesmente classes totalmente abstratas, ou seja, classes cujos métodos, em sua totalidade, são abstratos e, em Java, podem ou não ser qualificados com a palavra chave abstract, que poderá ficar subentendida.

---

Em vez de serem introduzidas pela palavra chave class, interfaces são introduzidas pela palavra chave interface.

Os dados de uma interface, se existirem, deverão ser todos qualificados com os modificadores static final; em outras palavras, os dados de uma interface, se existirem, deverão ser constantes de classe.

Assim como a herança entre classes, a herança entre interfaces é expressa pela palavra extends. No entanto, quando uma classe herda de uma interface, a palavra chave a ser usada para expressar a herança é a palavra implements.

## Classes Membro

Já estamos acostumados a ver, dentro da definição de classes, a definição de dados e funções membros de instância ou de classe; mas não são apenas os dados e as funções que podem ser membros, classes também podem. E não apenas dentro de classes, mas de interfaces também.

Classes membro nunca são qualificadas com a palavra static, mas, mesmo assim, sempre são consideradas membros de classe.

Classes membro podem ser qualificadas com as palavras public, private ou protected de forma a indicar um nível de acessibilidade diferente do default, exatamente como qualquer membro. O mais usual, no entanto, é que as qualifiquemos como private ou protected.

*[C:\ExplsJava\Expl\_12\Agenda.java]*

```
1: import java.util.regex.*;
2:
3: /**
4: A interface Agenda especifica o comportamento padrão de uma agenda telefonica.
5: Instâncias de classes que implementam esta interface devem conter métodos
6: para incluir e excluir.
7: Apesar de não especificar, explicitamente, os métodos canônicos, é óbvio que
8: esses métodos devem ser implementados nas classes que implementam esta
9: interface.
10: @author André Luís dos Reis Gomes de Carvalho.
11: @since 2000.
12: */
13: public interface Agenda extends Cloneable
```

---

```
14: {
15:     class Contato
16:     {
17:         /**
18:          Expressão regular que define a forma de um nome válido.
19:         */
20:         public static final String regExNom =
21:             "^[A-Z][a-z]*(?: (?:[A-Z]|[a-z])[a-z]*)*$";
22:
23:         /**
24:          Padrão que define como é um nome válido.
25:         */
26:         public static final Pattern padraoNom = Pattern.compile (regExNom);
27:
28:         /**
29:          Expressão regular que define a forma de um telefone nacional.
30:         */
31:         public static final String regExTel =
32:             "^(?:\\([0-9]{2}\\) )?9?[0-9]{4}-[0-9]{4}$";
33:
34:         /**
35:          Padrão que define como é um telefone nacional válido.
36:         */
37:         public static final Pattern padraoTel = Pattern.compile (regExTel);
38:
39:         /**
40:          Valida o nome de um contato.
41:          Verifica se o nome fornecido como parâmetro é um nome válido,
42:          lançando exceções, caso incorretudes sejam detectadas.
43:          @param nom o nome a ser validado.
44:          @throws Exception se não for fornecido um nome, ou se o nome fornecido
45:                          não parecer ser um nome correto.
46:         */
47:         public static void valideNome (String nom) throws Exception
48:         {
49:             if (nom==null)
50:                 throw new Exception ("Nome não fornecido");
51:
52:             if (!Agenda.Contato.padraoNom.matcher(nom).matches())
53:                 throw new Exception ("Nome invalido");
54:         }
55:
56:         /**
57:          Valida o telefone de um contato.
58:          Verifica se o número de telefone fornecido é um número
59:          nacional válido, lançando exceções, caso incorretudes sejam
60:          detectadas.
61:          @param tel o telefone a ser validado.
62:          @throws Exception se não for fornecido um telefone, ou se o telefone
63:                          não parecer ser um telefone correto.
64:         */
65:         public static void valideTelefone (String tel) throws Exception
66:         {
67:             if (tel==null)
68:                 throw new Exception ("Telefone não fornecido");
69:
```

```
70:         if (!Agenda.Contato.padraoTel.matcher(tel).matches())
71:             throw new Exception ("Telefone invalido");
72:     }
73:
74:     /**
75:      Atributo que representa o nome de um contato.
76:      */
77:     protected String nome;
78:
79:     /**
80:      Atributo que representa o telefone de um contato.
81:      */
82:     protected String telefone;
83:
84:     /**
85:      Constroi uma nova instância da classe interna e protegida
86:      Contato.
87:      Para tanto, devem ser fornecidos dois Strings que serão
88:      utilizados, respectivamente, como nome e telefone da
89:      instância recém criada.
90:      @param nome o String a ser utilizado como nome.
91:      @param telefone o String a ser utilizado como telefone.
92:      @throws Exception se não for fornecido um nome, ou se o nome fornecido
93:                      não parecer ser um nome correto, ou se não for fornecido
94:                      um telefone, ou se o telefone fornecido não parecer ser
95:                      um telefone correto.
96:      */
97:     public Contato (String nom, String tel) throws Exception
98:     {
99:         this.setNome (nom);
100:        this.setTelefone (tel);
101:    }
102:
103:    /**
104:       Obtem o nome de um contato.
105:       Permite que seja recuperado o nome do contato chamante
106:       do método.
107:       @return o nome do contato.
108:       */
109:     public String getNome ()
110:     {
111:         return this.nome;
112:     }
113:
114:    /**
115:       Obtem o telefone de um contato.
116:       Permite que seja recuperado o telefone do contato chamante
117:       do método.
118:       @return o telefone do contato.
119:       */
120:     public String getTelefone ()
121:     {
122:         return this.telefone;
123:     }
124:    /**
125:       */
```

```
126:     Ajusta o nome de um contato.
127:     Permite a modificação do nome do contato chamante
128:     do método.
129:     @param nom o novo nome do contato.
130:     @throws Exception se não for fornecido um nome, ou se o nome fornecido
131:                     não parecer ser um nome correto.
132:     */
133:     public void setNome (String nom) throws Exception
134:     {
135:         Agenda.Contato.validateNome (nom);
136:         this.nome = nom;
137:     }
138:
139:     /**
140:     Ajusta o telefone de um contato.
141:     Permite a modificação do telefone do contato chamante
142:     do método.
143:     @param tel o novo telefone do contato.
144:     @throws Exception se não for fornecido um telefone, ou se o telefone
145:                     fornecido não parecer ser um telefone correto.
146:     */
147:     public void setTelefone (String tel) throws Exception
148:     {
149:         Agenda.Contato.validateTelefone (tel);
150:         this.telefone = tel;
151:     }
152:
153:     /**
154:     Converte uma agenda em um String.
155:     Produz e resulta uma instância da classe String que
156:     representa a instância à qual este método for aplicado.
157:     @return o String que representa a agenda chamante do
158:             método.
159:     */
160:     public String toString ()
161:     {
162:         return this.nome + " (" +
163:                 this.telefone + ")";
164:     }
165:
166:     /**
167:     Verifica a igualdade entre duas agendas.
168:     Verifica se o Object fornecido como parâmetro representa
169:     uma agenda com conteúdo idêntico ao da instância à qual
170:     este método for aplicado, resultando true em caso afirmativo,
171:     ou false, caso contrário.
172:     @return true, caso o Object fornecido ao método e a instância
173:             chamante do método representarem frações numericamente
174:             equivalentes, ou false, caso contrário.
175:     */
176:     public boolean equals (Object obj)
177:     {
178:         if (obj==null)
179:             return false;
180:
181:         if (this.getClass()!=obj.getClass())
```

```
182:         return false;
183:
184:         Contato contato = (Contato)obj;
185:
186:         if (!this.nome.equals(contato.nome))
187:             return false;
188:
189:         if (!this.telefone.equals(contato.telefone))
190:             return false;
191:
192:         return true;
193:     }
194:
195:     /**
196:      Calcula o código de espalhamento (ou código de hash) de uma
197:      agenda.
198:      Calcula e resulta o código de espalhamento (ou código de
199:      hash, ou ainda o hashCode) da agenda representada pela
200:      instância à qual o método for aplicado.
201:      @return o código de espalhamento da agenda chamante do método.
202:     */
203:     public int hashCode ()
204:     {
205:         int ret = super.hashCode();
206:
207:         ret = 13*ret + this.nome.hashCode();
208:         ret = 13*ret + this.telefone.hashCode();
209:
210:         return ret;
211:     }
212:
213:     /**
214:      Constroi uma nova instância da classe Contato.
215:      Para tanto, deve ser fornecida uma agenda que servirá como
216:      modelo.
217:      @param modelo a instância que servirá como modelo.
218:      @throws Exception se o modelo for null.
219:     */
220:     public Contato (Contato modelo) throws Exception
221:     {
222:         if (modelo==null)
223:             throw new Exception ("Contato modelo nao fornecido");
224:
225:         this.nome      = modelo.nome;
226:         this.telefone = modelo.telefone;
227:     }
228:
229:
230:     /**
231:      Produz uma cópia fiel de uma agenda.
232:      Produz e resulta uma cópia exata da agenda à qual o método for cado.
233:      @return a cópia da agenda chamante do método.
234:     */
235:     public Object clone ()
236:     {
237:         Contato ret = null;
```

```

238:
239:         try
240:         {
241:             ret = new Contato (this);
242:         }
243:         catch (Exception e)
244:         {}
245:
246:         return ret;
247:     }
248: }
249:
250: /**
251:  * Estabelece que agendas são capazes de incorporar um novo contato.
252:  * Estabelece que, dados um nome e um telefone, agendas às quais este
253:  * método for aplicado devem passar a armazenar um novo contato com
254:  * os dados fornecidos.
255:  * @param nom o nome do novo contato.
256:  * @param tel o telefone do novo contato.
257:  * @throws Exception se não for fornecido um nome, ou se o nome fornecido
258:  *                   não parecer ser um nome correto, ou se não for fornecido
259:  *                   um telefone, ou se o telefone fornecido não parecer ser
260:  *                   um telefone correto, ou se a agenda estiver cheia, ou
261:  *                   ainda se o nome fornecido já estiver cadastrado.
262:  */
263: void registreContato (String nom, String tel) throws Exception;
264:
265: /**
266:  * Estabelece que agendas são capazes de descartar um contato.
267:  * Estabelece que, dado um nome, agendas às quais este método for aplicado
268:  * devem deixar de armazenar o contato com o nome fornecido.
269:  * @param nom o nome do contato a ser descartado.
270:  * @throws Exception se não for fornecido um nome, ou se o nome fornecido
271:  *                   não parecer ser um nome correto, ou se a agenda
272:  *                   estiver vazia, ou ainda se a agenda não contiver um
273:  *                   contato com o nome fornecido.
274:  */
275: void descarteContato (String nom) throws Exception;
276: }

```

### [C:\ExplsJava\Expl\_12\Agenda1.java]

```

1: /**
2:  * A classe Agenda1 representa uma simples agenda de telefone implementada tendo
3:  * como base um vetor que armazena instâncias da classe Contato, que é uma classe
4:  * interna da interface Agenda, implementada por esta classe.
5:  * Instâncias desta classe permitem a realização das operações básicas de uma agenda.
6:  * Nela encontramos, por exemplo, métodos para incluir, excluir e listar
7:  * contatos.
8:  * @author André Luís dos Reis Gomes de Carvalho.
9:  * @since 2000.
10: */
11: public class Agenda1 implements Agenda
12: {
13:     /**

```

```
14:     Atributo que representa, em qualquer instante, a quantidade de contatos
15:     armazenados em uma agenda.
16:     */
17:     protected int qtdContatos=0;
18:
19:     /**
20:     Atributo que representa o conjunto dos contatos armazenados em uma
21:     agenda.
22:     */
23:     protected Agenda.Contato[] contato;
24:
25:     /**
26:     Localiza um nome dado na agenda.
27:     Procura um contato na agenda, pelo método da busca binária,
28:     resultando um número inteiro negativo, quando o nome
29:     procurado não tiver sido encontrado, ou um numero inteiro
30:     positivo, quando o nome procurado tiver sido encontrado.
31:     @param nom o nome a ser procurado.
32:     @return um inteiro ao qual se deve dar a seguinte interpretação:
33:             <ol>
34:                 <li>
35:                     Um inteiro i negativo é retornado quando o nome
36:                     procurado não foi encontrado, mas, caso ele fosse ser
37:                     inserido, para manter os nomes da agenda em ordem
38:                     alfabetica, o local apropriado para a inserção seria
39:                     a posição -i-1.
40:                 </li>
41:                 <li>
42:                     Um inteiro i positivo é retornado quando o nome
43:                     procurado foi encontrado na posição i-1.
44:                 </li>
45:             </ol>
46:     */
47:     protected int ondeEsta (String nom)
48:     {
49:         int inicio      = 0,
50:             fim        = this.qtdContatos - 1,
51:             meio       = 0,
52:             comparacao;
53:
54:         while (inicio <= fim)
55:         {
56:             meio      = (inicio + fim) / 2;
57:             comparacao = nom.compareTo (this.contato[meio].getNome());
58:
59:             if (comparacao == 0)
60:                 return meio+1;
61:             else
62:                 if (comparacao < 0)
63:                     fim = meio-1;
64:                 else
65:                     inicio = meio+1;
66:         }
67:
68:         return -(inicio+1);
69:     }
```

```
70:
71:     /**
72:      Constroi uma nova instância da classe Agenda1.
73:      Para tanto, deve ser fornecido um inteiro que será utilizado
74:      como capacidade da instância recém criada.
75:      @param cap o número inteiro a ser utilizado como capacidade.
76:      @throws Exception se a capacidade for negativa ou zero.
77:     */
78:    public Agenda1 (int cap) throws Exception
79:    {
80:        if (cap <= 0)
81:            throw new Exception ("Capacidade invalida");
82:
83:        this.contato = new Agenda.Contato [cap];
84:    }
85:
86:    /**
87:     Inclui um novo contato em uma agenda.
88:     Acrescenta um contato com o nome e telefone fornecidos na instância
89:     à qual este método for aplicado.
90:     @param nom o nome do novo contato.
91:     @param tel o telefone do novo contato.
92:     @throws Exception se não for fornecido um nome, ou se o nome fornecido
93:                     não parecer ser um nome correto, ou se não for fornecido
94:                     um telefone, ou se o telefone fornecido não parecer ser
95:                     um telefone correto, ou se a agenda estiver cheia, ou
96:                     ainda se o nome fornecido já estiver cadastrado.
97:    */
98:    public void registreContato (String nom, String tel) throws Exception
99:    {
100:        if (this.qtdContatos == this.contato.length)
101:            throw new Exception ("Agenda cheia");
102:
103:        Agenda.Contato.validaNome (nom);
104:        Agenda.Contato.validaTelefone (tel);
105:
106:        int posicao = this.ondeEsta (nom);
107:
108:        if (posicao > 0)
109:            throw new Exception ("Nome ja registrado");
110:
111:        posicao = (-posicao)-1;
112:
113:        for (int pos=this.qtdContatos-1; pos>=posicao; pos--)
114:            this.contato [pos+1] = this.contato [pos];
115:
116:        this.contato [posicao] = new Agenda.Contato (nom, tel);
117:
118:        this.qtdContatos++;
119:    }
120:
121:    /**
122:     Remove um contato, dado seu nome.
123:     Exclui da instância à qual este método for aplicado o contato
124:     cujo nome foi fornecido.
125:     @param nom o nome do contato a ser descartado.
```

```
126:     @throws Exception se não for fornecido um nome, ou se o nome fornecido
127:                         não parecer ser um nome correto, ou se a agenda
128:                         estiver vazia, ou ainda se a agenda não contiver um
129:                         contato com o nome fornecido.
130:     */
131:     public void descarteContato (String nom) throws Exception
132:     {
133:         if (this.qtdContatos == 0)
134:             throw new Exception ("Agenda vazia");
135:
136:         Agenda.Contato.validaNome (nom);
137:
138:         int posicao = this.ondeEsta (nom);
139:
140:         if (posicao < 0)
141:             throw new Exception ("Nome inexistente");
142:
143:         posicao--;
144:
145:         int pos;
146:
147:         for (pos = posicao; pos < this.qtdContatos - 1; pos++)
148:             this.contato [pos] = this.contato [pos+1];
149:
150:         this.contato [pos] = null;
151:
152:         this.qtdContatos--;
153:     }
154:
155:     /**
156:      Gera uma representação textual de todo conteúdo da agenda.
157:      Produz e resulta um String com todos os nomes e telefones contidos
158:      na agenda.
159:      @return um String contendo todo o conteúdo da agenda.
160:     */
161:     public String toString ()
162:     {
163:         String ret = "";
164:
165:         for (int pos=0; pos<this.qtdContatos; pos++)
166:             ret += this.contato [pos] + "\n";
167:
168:         return ret;
169:     }
170:
171:     /**
172:      Verifica a igualdade entre duas agendas.
173:      Verifica se o Object fornecido como parâmetro representa uma
174:      agenda igual àquela representada pela instância à qual este
175:      método for aplicado, resultando true em caso afirmativo,
176:      ou false, caso contrário.
177:      @return true, caso o Object fornecido ao método e a instância chamante do
178:              método representarem agendas iguais, ou false, caso contrário.
179:     */
180:     public boolean equals (Object obj)
181:     {
```

```
182:         if (this == obj)
183:             return true;
184:
185:         if (obj == null)
186:             return false;
187:
188:         if (this.getClass() != obj.getClass())
189:             return false;
190:
191:         Agendal agenda = (Agendal) obj;
192:
193:         if (this.qtdContatos != agenda.qtdContatos)
194:             return false;
195:
196:         for (int pos=0; pos<this.qtdContatos; pos++)
197:             if (!this.contato[pos].equals(agenda.contato[pos]))
198:                 return false;
199:
200:         return true;
201:     }
202:
203: /**
204:  * Calcula o código de espalhamento (ou código de hash) de uma agenda.
205:  * Calcula e resulta o código de espalhamento (ou código de hash, ou ainda o
206:  * hashCode) da agenda representada pela instância à qual o método for aplicado.
207:  * @return o código de espalhamento da agenda chamante do método.
208: */
209: public int hashCode ()
210: {
211:     int ret = super.hashCode();
212:
213:     ret = 13*ret + this.qtdContatos;
214:
215:     for (int pos=0; pos<this.qtdContatos; pos++)
216:         ret = 13*ret + this.contato[pos].hashCode();
217:
218:     return ret;
219: }
220:
221: /**
222:  * Constroi uma cópia da instância da classe Agendal dada.
223:  * Para tanto, deve ser fornecida uma instancia da classe Agendal para ser
224:  * utilizada como modelo para a construção da nova instância criada.
225:  * @param modelo a instância da classe Agendal a ser usada como modelo.
226:  * @throws Exception se o modelo for null.
227: */
228: public Agendal (Agendal modelo)
229:             throws Exception
230: {
231:     if (modelo==null)
232:         throw new Exception ("Modelo nao fornecido");
233:
234:     this.contato = new Agenda.Contato [modelo.contato.length];
235:
236:     this.qtdContatos = modelo.qtdContatos;
237:
```

```
238:         for (int pos=0; pos<this.qtdContatos; pos++)
239:             this.contato[pos] = (Agenda.Contato)modelo.contato[pos].clone();
240:     }
241:
242:     /**
243:      Clona uma agenda.
244:      Produz e resulta uma cópia da agenda representada pela instância
245:      à qual o método for aplicado.
246:      @return a cópia da agenda representada pela instância à qual
247:              o método for aplicado.
248:     */
249:     public Object clone ()
250:     {
251:         Agendal copia=null;
252:
253:         try
254:         {
255:             copia = new Agendal (this);
256:         }
257:         catch (Exception e)
258:         {}
259:
260:         return copia;
261:     }
262: }
```

### [C:\ExplsJava\Expl\_12\Agenda2.java]

```
1: import java.util.Vector;
2:
3: /**
4:  A classe Agenda2 representa uma simples agenda de telefone implementada tendo
5:  como base um Vector que armazena instâncias da classe Contato, que é uma classe
6:  interna da interface Agenda, implementada por esta classe.
7:  Instâncias desta classe permitem a realização das operações básicas de uma agenda.
8:  Nela encontramos, por exemplo, métodos para incluir, excluir e listar
9:  contatos.
10: @author André Luís dos Reis Gomes de Carvalho.
11: @since 2000.
12: */
13: class Agenda2 implements Agenda
14: {
15:     /**
16:      Atributo que armazena os contatos destinados à agenda.
17:      */
18:     protected Vector<Agenda.Contato> contatos;
19:
20:     /**
21:      Localiza um nome dado na agenda.
22:      Procura um contato na agenda, pelo método da busca binária,
23:      resultando um número inteiro negativo, quando o nome
24:      procurado não tiver sido encontrado, ou um numero inteiro
25:      positivo, quando o nome procurado tiver sido encontrado.
26:      @param nom o nome a ser procurado
27:      @return um inteiro ao qual se deve dar a seguinte interpretação:
```

```
28:          <ol>
29:              <li>
30:                  Um inteiro i negativo é retornado quando o nome
31:                  procurado não foi encontrado, mas, caso ele fosse ser
32:                  inserido, para manter os nomes da agenda em ordem
33:                  alfabética, o local apropriado para a inserção seria
34:                  a posição -i-1.
35:              </li>
36:          <li>
37:              Um inteiro i positivo é retornado quando o nome
38:              procurado foi encontrado na posição i-1.
39:          </li>
40:      </ol>
41:  */
42: protected int ondeEsta (String nom)
43: {
44:     int inicio    = 0,
45:         fim       = this.contatos.size() - 1,
46:         meio      = 0,
47:         comparacao = -1;
48:
49:     while (inicio <= fim)
50:     {
51:         meio      = (inicio + fim) / 2;
52:         comparacao = nom.compareTo (this.contatos.elementAt(meio).getNome ());
53:
54:         if (comparacao == 0)
55:             return meio+1;
56:         else
57:             if (comparacao < 0)
58:                 fim = meio-1;
59:             else
60:                 inicio = meio+1;
61:     }
62:
63:     return -(inicio+1);
64: }
65:
66: /**
67: Constrói uma cópia da instância da classe Agenda dada.
68: Para tanto, deve ser fornecida uma instância da classe Agenda para ser
69: utilizada como modelo para a construção da nova instância criada.
70: @param modelo a instância da classe Agenda2 a ser usada como modelo.
71: @throws Exception se o modelo for null.
72: */
73:
74: public Agenda2 ()
75: {
76:     contatos = new Vector<Agenda.Contato> ();
77: }
78:
79: /**
80: Inclui um novo contato em uma agenda.
81: Acrescenta um contato com o nome e telefone fornecidos na instância
82: à qual este método for aplicado.
83: @param nom o nome do novo contato.
```

```
84:     @param tel o telefone do novo contato.
85:     @throws Exception se não for fornecido um nome, ou se o nome fornecido
86:                         não parecer ser um nome correto, ou se a agenda
87:                         estiver vazia, ou ainda se a agenda não contiver um
88:                         contato com o nome fornecido.
89:     */
90:     public void registreContato (String nom, String tel) throws Exception
91:     {
92:         Agenda.Contato.validaNome (nom);
93:         Agenda.Contato.validaTelefone (tel);
94:
95:         int posicao = this.ondaEsta (nom);
96:
97:         if (posicao > 0)
98:             throw new Exception ("Nome ja registrado");
99:
100:        posicao = (-posicao)-1;
101:
102:        this.contatos.insertElementAt (new Agenda.Contato(nom,tel), posicao);
103:    }
104:
105:    /**
106:     Remove um contato, dado seu nome.
107:     Exclui da instância à qual este método for aplicado o contato
108:     cujo nome foi fornecido.
109:     @param nom o nome do contato a ser descartado.
110:     @throws Exception se não for fornecido um nome, ou se o nome fornecido
111:                         não parecer ser um nome correto, ou se a agenda
112:                         estiver vazia, ou ainda se a agenda não contiver um
113:                         contato com o nome fornecido.
114:     */
115:     public void descarteContato (String nom) throws Exception
116:     {
117:         if (this.contatos.size() == 0)
118:             throw new Exception ("Agenda vazia");
119:
120:         Agenda.Contato.validaNome (nom);
121:
122:         int posicao = this.ondaEsta (nom);
123:
124:         if (posicao < 0)
125:             throw new Exception ("Nome inexistente");
126:
127:         posicao--;
128:
129:         this.contatos.removeElementAt (posicao);
130:     }
131:
132:    /**
133:     Gera uma representação textual de todo conteúdo da agenda.
134:     Produz e resulta um String com todos os nomes e telefones contidos
135:     na agenda.
136:     @return um String contendo todo o conteúdo da agenda.
137:     */
138:     public String toString ()
139:     {
```

```
140:         String ret = "";
141:
142:         for (Agenda.Contato c : this.contatos)
143:             ret += c.getNome() + "(" + c.getTelefone() + ") \n";
144:
145:         return ret;
146:     }
147:
148:     /**
149:      Verifica a igualdade entre duas agendas.
150:      Verifica se o Object fornecido como parâmetro representa uma
151:      agenda igual àquela representada pela instância à qual este
152:      método for aplicado, resultando true em caso afirmativo,
153:      ou false, caso contrário.
154:      @return true, caso o Object fornecido ao método e a instância chamante do
155:             método representarem agendas iguais, ou false, caso contrário.
156:     */
157:     public boolean equals (Object obj)
158:     {
159:         if (this == obj)
160:             return true;
161:
162:         if (obj == null)
163:             return false;
164:
165:         if (this.getClass() != obj.getClass())
166:             return false;
167:
168:         Agenda2 agenda = (Agenda2) obj;
169:
170:         return this.contatos.equals(agenda.contatos);
171:     }
172:
173:     /**
174:      Calcula o código de espalhamento (ou código de hash) de uma agenda.
175:      Calcula e resulta o código de espalhamento (ou código de hash, ou ainda o
176:      hashCode) da agenda representada pela instância à qual o método for aplicado.
177:      @return o código de espalhamento da agenda chamante do método.
178:     */
179:     public int hashCode ()
180:     {
181:         return this.contatos.hashCode();
182:     }
183:
184:     /**
185:      Constroi uma cópia da instância da classe Agenda dada.
186:      Para tanto, deve ser fornecida uma instancia da classe Agenda para ser
187:      utilizada como modelo para a construção da nova instância criada.
188:      @param modelo a instância da classe Agenda2 a ser usada como modelo.
189:      @throws Exception se o modelo for null.
190:     */
191:     public Agenda2 (Agenda2 modelo)
192:             throws Exception
193:     {
194:         if (modelo==null)
195:             throw new Exception ("Modelo nao fornecido");
```

```
196:
197:         /*
198:          Não fosse o fato do método clone da classe Vector fazer shall copy
199:          (em vez de deep copy), poderíamos substituir tudo que segue este
200:          comentário, simplesmente, por:
201:
202:          this.contatos = (Vector<Agenda.Contato>)modelo.contatos.clone();
203:          */
204:
205:          this.contatos = new Vector<Agenda.Contato> (modelo.contatos.capacity());
206:
207:          for (Agenda.Contato c : modelo.contatos)
208:              this.contatos.add ((Agenda.Contato)c.clone());
209:      }
210:
211:      /**
212:       Clona uma agenda.
213:       Produz e resulta uma cópia da agenda representada pela instância
214:       à qual o método for aplicado.
215:       @return a cópia da agenda representada pela instância à qual
216:              o método for aplicado.
217:      */
218:      public Object clone ()
219:      {
220:          Agenda2 copia=null;
221:
222:          Try
223:          {
224:              copia = new Agenda2 (this);
225:          }
226:          catch (Exception e)
227:          {}
228:
229:          return copia;
230:      }
231:  }
```

[C:\ExplsJava\Expl\_12\Agenda3.java]

```
1: import java.util.ArrayList;
2:
3: /**
4:  A classe Agenda3 representa uma simples agenda de telefone implemementada tendo
5:  como base um ArrayList que armazena instâncias da classe Contato, que é uma classe
6:  interna da interface Agenda, implementada por esta classe.
7:  Instâncias desta classe permitem a relização das operações básicas de uma agenda.
8:  Nela encontramos, por exemplo, métodos para incluir, excluir e listar
9:  contatos.
10: @author André Luis dos Reis Gomes de Carvalho.
11: @since 2000.
12: */
13: class Agenda3 implements Agenda
14: {
15:     /**
16:      Atributo que armazena os contatos destinados à agenda.
```

```
17:    */
18:    protected ArrayList<Agenda.Contato> contatos;
19:
20:    /**
21:     Localiza um nome dado na agenda.
22:     Procura um contato na agenda, pelo método da busca binária,
23:     resultando um número inteiro negativo, quando o nome
24:     procurado não tiver sido encontrado, ou um numero inteiro
25:     positivo, quando o nome procurado tiver sido encontrado.
26:     @param nom o nome a ser procurado.
27:     @return um inteiro ao qual se deve dar a seguinte interpretação:
28:             <ol>
29:                 <li>
30:                     Um inteiro i negativo é retornado quando o nome
31:                     procurado não foi encontrado, mas, caso ele fosse ser
32:                     inserido, para manter os nomes da agenda em ordem
33:                     alfabetica, o local apropriado para a inserção seria
34:                     a posição -i-1.
35:                 </li>
36:                 <li>
37:                     Um inteiro i positivo é retornado quando o nome
38:                     procurado foi encontrado na posição i-1.
39:                 </li>
40:             </ol>
41:    */
42:    protected int ondeEsta (String nom)
43:    {
44:        int inicio      = 0,
45:            fim         = this.contatos.size() - 1,
46:            meio        = 0,
47:            comparacao  = -1;
48:
49:        while (inicio <= fim)
50:        {
51:            meio      = (inicio + fim) / 2;
52:            comparacao = nom.compareTo (this.contatos.get(meio).getNome());
53:
54:            if (comparacao == 0)
55:                return meio+1;
56:            else
57:                if (comparacao < 0)
58:                    fim = meio-1;
59:                else
60:                    inicio = meio+1;
61:        }
62:
63:        return -(inicio+1);
64:    }
65:
66:    /**
67:     Constroi uma cópia da instância da classe Agenda dada.
68:     Para tanto, deve ser fornecida uma instancia da classe Agenda para ser
69:     utilizada como modelo para a construção da nova instância criada.
70:     @param modelo a instância da classe Agenda4 a ser usada como modelo.
71:     @throws Exception se o modelo for null.
72:    */
```

```
73:
74:     public Agenda3 ()
75:     {
76:         contatos = new ArrayList<Agenda.Contato> ();
77:     }
78:
79:     /**
80:      Inclui um novo contato em uma agenda.
81:      Acrescenta um contato com o nome e telefone fornecidos na instância
82:      à qual este método for aplicado.
83:      @param nom o nome do novo contato.
84:      @param tel o telefone do novo contato.
85:      @throws Exception se não for fornecido um nome, ou se o nome fornecido
86:                      não parecer ser um nome correto, ou se a agenda
87:                      estiver vazia, ou ainda se a agenda não contiver um
88:                      contato com o nome fornecido.
89:     */
90:     public void registreContato (String nom, String tel) throws Exception
91:     {
92:         Agenda.Contato.validaNome (nom);
93:         Agenda.Contato.validaTelefone (tel);
94:
95:         int posicao = this.ondeEsta (nom);
96:
97:         if (posicao > 0)
98:             throw new Exception ("Nome ja registrado");
99:
100:        posicao = (-posicao)-1;
101:
102:        this.contatos.add (posicao, new Agenda.Contato(nom,tel));
103:    }
104:
105:    /**
106:       Remove um contato, dado seu nome.
107:       Exclui da instância à qual este método for aplicado o contato
108:       cujo nome foi fornecido.
109:       @param nom o nome do contato a ser descartado.
110:       @throws Exception se não for fornecido um nome, ou se o nome fornecido
111:                      não parecer ser um nome correto, ou se a agenda
112:                      estiver vazia, ou ainda se a agenda não contiver um
113:                      contato com o nome fornecido.
114:    */
115:    public void descarteContato (String nom) throws Exception
116:    {
117:        if (this.contatos.size() == 0)
118:            throw new Exception ("Agenda vazia");
119:
120:        Agenda.Contato.validaNome (nom);
121:
122:        int posicao = this.ondeEsta (nom);
123:
124:        if (posicao < 0)
125:            throw new Exception ("Nome inexistente");
126:
127:        posicao--;
128:
```

```
129:         this.contatos.remove (posicao);
130:     }
131:
132:     /**
133:      Gera uma representação textual de todo conteúdo da agenda.
134:      Produz e resulta um String com todos os nomes e telefones contidos
135:
136:      na agenda.
137:      @return um String contendo todo o conteúdo da agenda.
138:     */
139:    public String toString ()
140:    {
141:        String ret = "";
142:
143:        for (Agenda.Contato c : this.contatos)
144:            ret += c.getNome() + "(" + c.getTelefone() + ")\n";
145:
146:        return ret;
147:    }
148:
149:    /**
150:       Verifica a igualdade entre duas agendas.
151:       Verifica se o Object fornecido como parâmetro representa uma
152:       agenda igual àquela representada pela instância à qual este
153:       método for aplicado, resultando true em caso afirmativo,
154:       ou false, caso contrário.
155:       @return true, caso o Object fornecido ao método e a instância chamante do
156:              método representarem agendas iguais, ou false, caso contrário.
157:     */
158:    public boolean equals (Object obj)
159:    {
160:        if (this == obj)
161:            return true;
162:
163:        if (obj == null)
164:            return false;
165:
166:        if (this.getClass() != obj.getClass())
167:            return false;
168:
169:        Agenda3 agenda = (Agenda3) obj;
170:
171:        return this.contatos.equals(agenda.contatos);
172:    }
173:
174:    /**
175:       Calcula o código de espalhamento (ou código de hash) de uma agenda.
176:       Calcula e resulta o código de espalhamento (ou código de hash, ou ainda o
177:       hashCode) da agenda representada pela instância à qual o método for aplicado.
178:       @return o código de espalhamento da agenda chamante do método.
179:     */
180:    public int hashCode ()
181:    {
182:        return this.contatos.hashCode();
183:    }
184:
```

```
185:  /**
186:   Constrói uma cópia da instância da classe Agenda dada.
187:   Para tanto, deve ser fornecida uma instância da classe Agenda para ser
188:   utilizada como modelo para a construção da nova instância criada.
189:   @param modelo a instância da classe Agenda4 a ser usada como modelo.
190:   @throws Exception se o modelo for null.
191: */
192: public Agenda3 (Agenda3 modelo)
193:           throws Exception
194: {
195:     if (modelo==null)
196:         throw new Exception ("Modelo não fornecido");
197:
198:     /*
199:      Não fosse o fato do método clone da classe Vector fazer shall copy
200:      (em vez de deep copy), poderíamos substituir tudo que segue este
201:      comentário, simplesmente, por:
202:
203:      this.contatos = (Vector<Agenda.Contato>)modelo.contatos.clone();
204:    */
205:
206:     this.contatos = new ArrayList<Agenda.Contato> (modelo.contatos.size());
207:
208:     for (Agenda.Contato c : modelo.contatos)
209:         this.contatos.add ((Agenda.Contato)c.clone());
210: }
211:
212: /**
213:  Clona uma agenda.
214:  Produz e resulta uma cópia da agenda representada pela instância
215:  à qual o método for aplicado.
216:  @return a cópia da agenda representada pela instância à qual
217:          o método for aplicado.
218: */
219: public Object clone ()
220: {
221:     Agenda3 copia=null;
222:
223:     try
224:     {
225:         copia = new Agenda3 (this);
226:     }
227:     catch (Exception e)
228:     {}
229:
230:     return copia;
231: }
232: }
```

[C:\ExplsJava\Expl\_12\TesteDeAgenda.java]

```
1: import java.io.IOException;
2: import java.io.BufferedReader;
3: import java.io.InputStreamReader;
4:
```

```
5: public class TesteDeAgenda
6: {
7:     public static void main (String[] args)
8:     {
9:         Agenda agenda = null;
10:
11:        BufferedReader entrada = new BufferedReader
12:                (new InputStreamReader
13:                     (System.in));
14:        int opcao;
15:
16:        for (;;)
17:        {
18:            System.out.print ("Digite sua opcao (" +
19:                               "1=Vetor/" +
20:                               "2=Vector)" +
21:                               "3=ArrayList)" +
22:                               ": ");
23:
24:            try
25:            {
26:                opcao = Integer.parseInt (entrada.readLine ());
27:
28:                if (opcao>=1 && opcao<=3)
29:                    break;
30:            }
31:            catch (IOException e)
32:            {}
33:            catch (NumberFormatException e)
34:            {
35:                System.err.println ("Opcao invalida\n\n");
36:            }
37:        }
38:
39:        System.err.println ();
40:
41:        switch (opcao)
42:        {
43:            case 1:
44:                for (;;)
45:                {
46:                    System.out.println ();
47:
48:                    System.out.print ("Capacidade desejada para a Agenda: ");
49:                    try
50:                    {
51:                        int cap = Integer.parseInt (entrada.readLine ());
52:                        agenda = new Agendal (cap);
53:                        break;
54:                    }
55:                    catch (IOException e)
56:                    {}
57:                    catch (NumberFormatException e)
58:                    {
59:                        System.err.println ("Nao foi digitado um numero inteiro");
60:                        System.err.println ();
```

```
61:          }
62:          catch (Exception e)
63:          {
64:              System.err.println (e.getMessage ());
65:              System.err.println ();
66:          }
67:      }
68:      break;
69:
70:
71:      case 2:
72:          agenda = new Agenda2 ();
73:          break;
74:
75:
76:      case 3:
77:          agenda = new Agenda3 ();
78:      }
79:
80:      System.out.println ();
81:
82:      String nome = null, telefone = null;
83:
84:      do
85:      {
86:          System.out.println ();
87:
88:          System.out.print ("Digite sua Opção (" +
89:                             "I=Incluir/" +
90:                             "E=Excluir/" +
91:                             "L=Listar/" +
92:                             "S=Sair)" +
93:                             ": ");
94:
95:          try
96:          {
97:              String str = entrada.readLine ();
98:
99:              if (str.length() == 1)
100:                  opcao = str.charAt(0);
101:              else
102:                  opcao = 'A'; // forçando opção inválida
103:          }
104:          catch (IOException e)
105:          {}
106:
107:          switch (opcao)
108:          {
109:              case 'i':
110:              case 'I':
111:                  try
112:                  {
113:                      System.out.print ("Nome....: ");
114:                      nome = entrada.readLine ();
115:                  }
116:                  catch (IOException e)
```

```
117:          {}
118:
119:          try
120:          {
121:              System.out.print ("Telefone: ");
122:              telefone = entrada.readLine ();
123:          }
124:          catch (IOException e)
125:          {}
126:
127:          try
128:          {
129:              agenda.registreContato (nome, telefone);
130:          }
131:          catch (Exception e)
132:          {
133:              System.err.println (e.getMessage ());
134:          }
135:
136:          System.out.println ();
137:          break;
138:
139:
140:          case 'e':
141:          case 'E':
142:              System.out.print ("Nome....: ");
143:              try
144:              {
145:                  nome = entrada.readLine ();
146:              }
147:              catch (IOException e)
148:              {}
149:
150:          try
151:          {
152:              agenda.descarteContato (nome);
153:          }
154:          catch (Exception e)
155:          {
156:              System.err.println (e.getMessage ());
157:          }
158:
159:          System.out.println ();
160:          break;
161:
162:
163:          case 'l':
164:          case 'L':
165:              System.out.println (agenda);
166:              break;
167:
168:
169:          case 's':
170:          case 'S':
171:              break;
172:
```

```
173:  
174:         default :  
175:             System.err.println ("Opcao invalida");  
176:             System.err.println ();  
177:         }  
178:     }  
179:     while ((opcao != 's') && (opcao != 'S'));  
180:   }  
181: }
```

Para compilar e executar este programa, daremos os seguintes comandos:

```
C:\ExplsJava\Expl_12> javac TesteDeAgenda.java  
C:\ExplsJava\Expl_12> java TesteDeAgenda
```

Isto poderia produzir no console a seguinte interação:

```
Digite sua opcao (1=Vetor/2=Vector/3=ArrayList): 7  
Opcao invalida!
```

```
Digite sua opcao (1=Vetor/2=Vector/3=ArrayList): 2
```

```
Capacidade desejada para a Agenda: -10  
Capacidade invalida
```

```
Capacidade desejada para a Agenda: 0  
Capacidade invalida
```

```
Capacidade desejada para a Agenda: 10
```

```
Digite sua Opcao (I=Incluir/E=Excluir/L=Listar/S=Sair): i  
Nome....: Jose  
Telefone: 3241-4466
```

```
Digite sua Opcao (I=Incluir/E=Excluir/L=Listar/S=Sair): i  
Nome....: Jose  
Telefone: 3241-4466  
Nome já registrado
```

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): i  
Nome....: Joao  
Telefone: 3252-9955

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): l  
Joao (3252-9955)  
Jose (3241-4466)

Digite sua Opcão (I=Incluir/C=Consultar/E=Excluir/S=Sair): e  
Nome: Joao

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): l  
Jose (3241-4466)

Digite sua Opcão (I=Incluir/C=Consultar/E=Excluir/S=Sair): a  
Opcão invalida

Digite sua Opcão (I=Incluir/C=Consultar/E=Excluir/S=Sair): s

## ***Herança Múltipla***

A linguagem Java não implementa o recurso de herança múltipla de forma tão abrangente quanto fazem outras linguagens, e.g., C++. Veja abaixo as restrições que Java impõe à herança múltipla:

1. Interfaces não podem derivar de classes;
  2. Interfaces podem derivar de múltiplas interfaces (sendo  $IB_i$  e  $ID$  nomes de interfaces, veja abaixo a forma geral do cabeçalho de uma interface que deriva de outras interfaces: interface  $ID$  extends  $IB_1, IB_2, \dots, IB_n$ );
  3. Classes podem derivar de apenas uma outra classe, mas podem derivar de múltiplas interfaces (sendo  $CB$  e  $CD$  nomes de classes, e  $IB_i$  nomes de interfaces, veja abaixo a forma geral do cabeçalho de uma classe que deriva de uma outra classe e de diversas interfaces: class  $CD$  extends  $CB$  implements  $IB_1, IB_2, \dots, IB_n$ ).
-

## A Classe `java.lang.Object`

Esta classe é a raiz da hierarquia de classes da linguagem Java, i.e., toda classe a tem como superclasse. Todos as classes (sejam elas pré-definidas na linguagem ou pelo programador definidas), incluindo os vetores, herdaram as características desta classe.

São particularmente úteis para servirem de base para a implementação genérica de classes container, i.e., classes armazéns.

Isto porque, se fizermos tais classes serem capazes de armazenar instâncias desta classe, torna-las-emos capazes de armazenar instâncias de qualquer classe, já que todas as classes são derivadas desta classe.

Naturalmente, nesta situação, as instâncias armazenadas, em certo sentido, perdem sua identidade e passam coletivamente a serem consideradas instâncias de Object.

Assim, ao serem extraídas, para que as instâncias possam recuperar sua identidade e, por conseguinte, ser usadas como instâncias das classes usadas em sua criação, as mesmas deverão passar por um processo de conversão de tipo, que, neste caso, funcionará mais como um processo de reposição de tipo do que, propriamente, de conversão de tipo.

Veja na documentação da linguagem a interface que a classe especifica para comunicação com ela própria e com suas instâncias.

## Classes Abstratas

Conforme sabemos, a herança convencional representa a possibilidade de incorporar e extender as características (comportamento e implementação) de uma classe já existente.

Também sabemos da existência de interfaces, que definem comportamentos, sem implementá-los, que podem ser herdados por outras interfaces, bem como implementados por classes.

---

Classes abstratas nos oferecem um meio termo, i.e., a possibilidade de herdar alguns comportamentos implementados, outros por implementar. Os comportamentos não implementados são representados por métodos abstratos.

### [C:\ExplsJava\Expl\_13\agenda\Agenda.java]

```
1: package agenda;
2:
3: import java.util.regex.*;
4:
5: /**
6:  A classe Agenda representa uma simples agenda de telefone implementada tendo
7:  como base dois vetores que armazenam, respectivamente, os nomes e os telefones
8:  dos contatos da agenda.
9:  Instâncias desta classe permitem a realização das operações básicas de uma agenda.
10: Nela encontramos, por exemplo, métodos para incluir, excluir e listar
11: contatos.
12: @author André Luís dos Reis Gomes de Carvalho.
13: @since 2000.
14: */
15: public abstract class Agenda implements Cloneable
16: {
17:     /**
18:      Expressão regular que define a forma de um nome válido.
19:      */
20:     protected static final String regExNom =
21:         "[A-Z][a-z]*(?: ([A-Z]|[a-z])[a-z]*)*$";
22:
23:     /**
24:      Padrão que define como é um nome válido.
25:      */
26:     protected static final Pattern padraoNom = Pattern.compile (regExNom);
27:
28:     /**
29:      Expressão regular que define a forma de um telefone nacional.
30:      */
31:
32:     /**
33:      Padrão que define como é um telefone nacional válido.
34:      */
35:     protected static final String regExTel =
36:         "(?:\\([0-9]{2}\\) )?9?[0-9]{4}-[0-9]{4}$";
37:
38:     protected static final Pattern padraoTel = Pattern.compile (regExTel);
39:
40:     /**
41:      Valida o nome de um contato.
42:      Verifica se o nome fornecido como parâmetro é um nome válido,
43:      lançando exceções, caso incorretudes sejam detectadas.
44:      @param nom o nome a ser validado.
45:      @throws Exception se não for fornecido um nome, ou se o nome fornecido
46:                      não parecer ser um nome correto.
```

```
47:    */
48:    public static void valideNome (String nom) throws Exception
49:    {
50:        if (nom==null)
51:            throw new Exception ("Nome nao fornecido");
52:
53:        if (!Agenda.padraoNom.matcher(nom).matches())
54:            throw new Exception ("Nome invalido");
55:    }
56:
57:    /**
58:     Valida o telefone de um contato.
59:     Verifica se o número de telefone fornecido é um número
60:     nacional válido, lançando exceções, caso incorretudes sejam
61:     detectadas.
62:     @param tel o telefone a ser validado.
63:     @throws Exception se não for fornecido um telefone, ou se o telefone
64:                         fornecido não parecer ser um telefone correto.
65:    */
66:    public static void valideTelefone (String tel) throws Exception
67:    {
68:        if (tel==null)
69:            throw new Exception ("Telefone nao fornecido");
70:
71:        if (!Agenda.padraoTel.matcher(tel).matches())
72:            throw new Exception ("Telefone invalido");
73:    }
74:
75:    /**
76:     Expressa, em cada instante, a quantidade de contatos
77:     armazenados na agenda.
78:    */
79:    protected int qtdContatos=0;
80:
81:    /**
82:     Mantém armazenados os nomes dos contatos armazenados
83:     na agenda.
84:    */
85:    protected String[] nome;
86:
87:    /**
88:     Mantém armazenados os telefones dos contatos armazenados
89:     na agenda.
90:    */
91:    protected String[] telefone;
92:
93:    /**
94:     Localiza um nome dado na agenda.
95:     Procura um contato na agenda, resultando um número inteiro
96:     negativo, quando o nome procurado não tiver sido encontrado,
97:     ou um numero inteiro positivo, quando o nome procurado tiver
98:     sido encontrado.
99:     @param nom o nome a ser procurado.
100:    @return um inteiro ao qual se deve dar a seguinte interpretação:
101:          <ol>
102:              <li>
```

```
103:          Um inteiro i negativo é retornado quando o nome
104:          procurado não foi encontrado, mas, caso ele fosse ser
105:          inserido, para manter os nomes da agenda em ordem
106:          alfabetica, o local apropriado para a inserção seria
107:          a posição -i-1.
108:      </li>
109:      <li>
110:          Um inteiro i positivo é retornado quando o nome
111:          procurado foi encontrado na posição i-1.
112:      </li>
113:  </ol>
114:  */
115: protected abstract int ondeEsta (String nom);
116:
117: /**
118: Constrói uma nova instância da classe Agenda.
119: Para tanto, deve ser fornecido um inteiro que será utilizado
120: como capacidade da instância recém criada.
121: @param cap o número inteiro a ser utilizado como capacidade.
122: @throws Exception se a capacidade for negativa ou zero.
123: */
124: public Agenda (int cap)
125:             throws Exception
126: {
127:     if (cap <= 0)
128:         throw new Exception ("Capacidade invalida");
129:
130:     this.nome    = new String [cap];
131:     this.telefone = new String [cap];
132: }
133:
134: /**
135: Inclui um novo contato em uma agenda.
136: Acrescenta um contato com o nome e telefone fornecidos na instância
137: à qual este método for aplicado.
138: @param nom o nome do novo contato.
139: @param tel o telefone do novo contato.
140: @throws Exception se não for fornecido um nome, ou se o nome fornecido
141:                 não parecer ser um nome correto, ou se não for fornecido
142:                 um telefone, ou se o telefone fornecido não parecer ser
143:                 um telefone correto, ou se a agenda estiver cheia, ou
144:                 ainda se o nome fornecido já estiver cadastrado.
145: */
146: public abstract void registreContato (String nom,
147:                                         String tel)
148:             throws Exception;
149:
150: /**
151: Remove um contato, dado seu nome.
152: Exclui da instância à qual este método for aplicado o contato
153: cujo nome foi fornecido.
154: @param nom o nome do contato a ser descartado.
155: @throws Exception se não for fornecido um nome, ou se o nome fornecido
156:                 não parecer ser um nome correto, ou se a agenda
157:                 estiver vazia, ou ainda se a agenda não contiver um
158:                 contato com o nome fornecido.
```

```
159:     */
160:     public void descarteContato (String nom)
161:                         throws Exception
162:     {
163:         if (this.qtdContatos == 0)
164:             throw new Exception ("Agenda vazia");
165:
166:         if (nom==null)
167:             throw new Exception ("Nome nao fornecido");
168:
169:         if (!Agenda.padraoNom.matcher(nom).matches())
170:             throw new Exception ("Nome invalido");
171:
172:         int posicao = this.ondeEsta (nom);
173:
174:         if (posicao < 0)
175:             throw new Exception ("Nome inexistente");
176:
177:         posicao--;
178:
179:         int pos;
180:
181:         for (pos = posicao; pos < this.qtdContatos - 1; pos++)
182:         {
183:             this.nome [pos] = this.nome [pos+1];
184:             this.telefone [pos] = this.telefone [pos+1];
185:         }
186:
187:         this.nome [pos] = null;
188:         this.telefone [pos] = null;
189:
190:         this.qtdContatos--;
191:     }
192:
193:     /**
194:      Gera uma representação textual de todo conteúdo da agenda.
195:      Produz e resulta um String com todos os nomes e telefones contidos
196:      na agenda.
197:      @return um String contendo todo o conteúdo da agenda.
198:      */
199:     public String toString ()
200:     {
201:         String ret = "";
202:
203:         for (int pos=0; pos<this.qtdContatos; pos++)
204:             ret += this.nome[pos] + " (" + this.telefone[pos] + ")\n";
205:
206:         return ret;
207:     }
208:
209:     /**
210:      Verifica a igualdade entre duas agendas.
211:      Verifica se o Object fornecido como parâmetro representa uma
212:      agenda igual àquela representada pela instância à qual este
213:      método for aplicado, resultando true em caso afirmativo,
214:      ou false, caso contrário.
```

```
215:     @param obj o objeto a ser comparado com a instância à qual esse método
216:             for aplicado.
217:     @return true, caso o Object fornecido ao método e a instância chamante do
218:             método representarem agendas iguais, ou false, caso contrário.
219:     */
220:     public boolean equals (Object obj)
221:     {
222:         if (this == obj)
223:             return true;
224:
225:         if (obj == null)
226:             return false;
227:
228:         if (this.getClass() != obj.getClass())
229:             return false;
230:
231:         Agenda agenda = (Agenda) obj;
232:
233:         if (this.qtdContatos != agenda.qtdContatos)
234:             return false;
235:
236:         for (int pos=0; pos<this.qtdContatos; pos++)
237:             if (!this.nome [pos].equals(agenda.nome [pos]) ||
238:                 !this.telefone[pos].equals(agenda.telefone[pos]))
239:                 return false;
240:
241:         return true;
242:     }
243:
244:     /**
245:      Calcula o código de espalhamento (ou código de hash) de uma agenda.
246:      Calcula e resulta o código de espalhamento (ou código de hash, ou ainda o
247:      hashCode) da agenda representada pela instância à qual o método for aplicado.
248:      @return o código de espalhamento da agenda chamante do método.
249:     */
250:     public int hashCode ()
251:     {
252:         int ret = super.hashCode();
253:
254:         ret = 13*ret + this.qtdContatos;
255:
256:         for (int pos=0; pos<this.qtdContatos; pos++)
257:         {
258:             ret = 13*ret + this.nome [pos].hashCode();
259:             ret = 13*ret + this.telefone[pos].hashCode();
260:         }
261:
262:         return ret;
263:     }
264:
265:     /**
266:      Constroi uma cópia da instância da classe Agenda dada.
267:      Para tanto, deve ser fornecida uma instancia da classe Agenda para ser
268:      utilizada como modelo para a construção da nova instância criada.
269:      @param modelo a instância da classe Agenda a ser usada como modelo.
270:      @throws Exception se o modelo for null.
```

```

271:      */
272:      public Agenda (Agenda modelo)
273:          throws Exception
274:      {
275:          if (modelo==null)
276:              throw new Exception ("Modelo nao fornecido");
277:
278:          this.nome      = new String [modelo.nome      .length];
279:          this.telefone = new String [modelo.telefone.length];
280:
281:          this.qtdContatos = modelo.qtdContatos;
282:
283:          for (int pos=0; pos<this.qtdContatos; pos++)
284:          {
285:              this.nome      [pos] = modelo.nome      [pos];
286:              this.telefone [pos] = modelo.telefone [pos];
287:          }
288:      }
289:  }

```

[C:\ExplsJava\Expl\_13\agenda\desordenada\AgendaDesordenada.java]

```

1: package agenda.desordenada;
2: import agenda.*;
3:
4: /**
5:  A classe AgendaDesordenada representa uma simples agenda de telefone implementada
6:  tendo como base dois vetores que armazenam, respectivamente, os nomes e os telefones
7:  dos contatos da agenda.
8:  Instâncias desta classe permitem a realização das operações básicas de uma agenda.
9:  Nela encontramos, por exemplo, métodos para incluir, excluir e listar
10: contatos.
11: @author André Luís dos Reis Gomes de Carvalho.
12: @since 2000.
13: */
14: public class AgendaDesordenada extends Agenda
15: {
16:     /**
17:      Localiza um nome dado na agenda.
18:      Procura um contato na agenda, pelo método da busca sequencial,
19:      resultando um número inteiro negativo, quando o nome
20:      procurado não tiver sido encontrado, ou um numero inteiro
21:      positivo, quando o nome procurado tiver sido encontrado.
22:      @param nom o nome a ser procurado.
23:      @return um inteiro ao qual se deve dar a seguinte interpretação:
24:             <ol>
25:                 <li>
26:                     Um inteiro i negativo é retornado quando o nome
27:                     procurado não foi encontrado, mas, caso ele fosse ser
28:                     inserido, para manter os nomes da agenda em ordem
29:                     alfabética, o local apropriado para a inserção seria
30:                     a posição -i-1.
31:                 </li>
32:                 <li>
33:                     Um inteiro i positivo é retornado quando o nome

```

```
34:             procurado foi encontrado na posição i-1.
35:         </li>
36:     </ol>
37:     */
38:     protected int ondeEsta (String nom)
39:     {
40:
41:
42:         for (pos=0; pos<this.qtdContatos; pos++)
43:             if (nom.equals(this.nome[pos]))
44:                 return pos+1;
45:
46:         return -(pos+1);
47:     }
48:
49:     /**
50:      Constroi uma nova instância da classe AgendaDesordenada.
51:      Para tanto, deve ser fornecido um inteiro que será utilizado
52:      como capacidade da instância recém criada.
53:      @param cap o número inteiro a ser utilizado como capacidade.
54:      @throws Exception se a capacidade for negativa ou zero.
55:     */
56:     public AgendaDesordenada (int cap)
57:             throws Exception
58:     {
59:         super (cap);
60:     }
61:
62:     /**
63:      Inclui um novo contato em uma agenda.
64:      Acrescenta um contato com o nome e telefone fornecidos na instância
65:      à qual este método for aplicado.
66:      @param nom o nome do novo contato.
67:      @param tel o telefone do novo contato.
68:      @throws Exception se não for fornecido um nome, ou se o nome fornecido
69:                      não parecer ser um nome correto, ou se não for fornecido
70:                      um telefone, ou se o telefone fornecido não parecer ser
71:                      um telefone correto, ou se a agenda estiver cheia, ou
72:                      ainda se o nome fornecido já estiver cadastrado.
73:     */
74:     public void registreContato (String nom,
75:                                 String tel)
76:             throws Exception
77:     {
78:         if (this.qtdContatos == this.nome.length)
79:             throw new Exception ("Agenda cheia");
80:
81:         Agenda.validaNome (nom);
82:         Agenda.validaTelefone (tel);
83:
84:         int posicao = this.ondeEsta (nom);
85:
86:         if (posicao > 0)
87:             throw new Exception ("Nome ja registrado");
88:
89:         this.nome [this.qtdContatos] = nom;
```

```
90:         this.telefone [this.qtdContatos] = tel;
91:
92:         this.qtdContatos++;
93:     }
94:
95:     /**
96:      Constroi uma cópia da instância da classe AgendaDesordenada dada.
97:      Para tanto, deve ser fornecida uma instancia da classe
98:      AgendaDesordenada para ser utilizada como modelo para a
99:      construção da nova instância criada.
100:     @param modelo a instância da classe AgendaDesordenada a ser usada
101:           como modelo.
102:     @throws Exception se o modelo for null.
103:    */
104:    public AgendaDesordenada (AgendaDesordenada modelo)
105:        throws Exception
106:    {
107:        super (modelo);
108:    }
109:
110:   /**
111:      Clona uma agenda.
112:      Produz e resulta uma cópia da agenda representada pela instância
113:      à qual o método for aplicado.
114:      @return a cópia da agenda representada pela instância à qual
115:             o método for aplicado.
116:    */
117:    public Object clone ()
118:    {
119:        AgendaDesordenada copia=null;
120:
121:        try
122:        {
123:            copia = new AgendaDesordenada (this);
124:        }
125:        catch (Exception e)
126:        {}
127:
128:        return copia;
129:    }
130: }
```

[C:\ExplsJava\Expl\_13\agenda\ordenada\AgendaOrdenada.java]

```
1: package agenda.ordenada;
2: import agenda.*;
3: import java.util.regex.*;
4:
5: /**
6:  A classe AgendaOrdenada representa uma simples agenda de telefone implementada
7:  tendo como base dois vetores que armazenam, respectivamente, os nomes e os telefones
8:  dos contatos da agenda.
9:  Instâncias desta classe permitem a realização das operações básicas de uma agenda.
10: Nela encontramos, por exemplo, métodos para incluir, excluir e listar
11: contatos.
```

```
12: @author André Luís dos Reis Gomes de Carvalho.
13: @since 2000.
14: */
15: public class AgendaOrdenada extends Agenda
16: {
17:     /**
18:      Localiza um nome dado na agenda.
19:      Procura um contato na agenda, pelo método da busca binária,
20:      resultando um número inteiro negativo, quando o nome
21:      procurado não tiver sido encontrado, ou um numero inteiro
22:      positivo, quando o nome procurado tiver sido encontrado.
23:      @param nom o nome a ser procurado.
24:      @return um inteiro ao qual se deve dar a seguinte interpretação:
25:              <ol>
26:                  <li>
27:                      Um inteiro i negativo é retornado quando o nome
28:                      procurado não foi encontrado, mas, caso ele fosse ser
29:                      inserido, para manter os nomes da agenda em ordem
30:                      alfabética, o local apropriado para a inserção seria
31:                      a posição -i-1.
32:                  </li>
33:                  <li>
34:                      Um inteiro i positivo é retornado quando o nome
35:                      procurado foi encontrado na posição i-1.
36:                  </li>
37:              </ol>
38:      */
39:     protected int ondeEsta (String nom)
40:     {
41:         int inicio = 0,
42:             fim    = this.qtdContatos - 1,
43:             meio,
44:             comparacao;
45:
46:         while (inicio <= fim)
47:         {
48:             meio      = (inicio + fim) / 2;
49:             comparacao = nom.compareTo (this.nome [meio]);
50:
51:             if (comparacao == 0)
52:                 return meio+1;
53:             else
54:                 if (comparacao < 0)
55:                     fim = meio-1;
56:                 else
57:                     inicio = meio+1;
58:         }
59:
60:         return -(inicio+1);
61:     }
62:
63:     /**
64:      Constroi uma nova instância da classe AgendaDesordenada.
65:      Para tanto, deve ser fornecido um inteiro que será utilizado
66:      como capacidade da instância recém criada.
67:      @param cap o número inteiro a ser utilizado como capacidade.
```

```
68:     @throws Exception se a capacidade for negativa ou zero.
69:     */
70:     public AgendaOrdenada (int cap)
71:             throws Exception
72:     {
73:         super (cap);
74:     }
75:
76:     /**
77:      Inclui um novo contato em uma agenda.
78:      Acrescenta um contato com o nome e telefone fornecidos na instância
79:      à qual este método for aplicado.
80:      @param nom o nome do novo contato.
81:      @param tel o telefone do novo contato.
82:      @throws Exception se não for fornecido um nome, ou se o nome fornecido
83:                      não parecer ser um nome correto, ou se não for fornecido
84:                      um telefone, ou se o telefone fornecido não parecer ser
85:                      um telefone correto, ou se a agenda estiver cheia, ou
86:                      ainda se o nome fornecido já estiver cadastrado.
87:     */
88:     public void registreContato (String nom,
89:                                 String tel)
90:             throws Exception
91:     {
92:         if (this.qtdContatos == this.nome.length)
93:             throw new Exception ("Agenda cheia");
94:
95:         Agenda.validaNome (nom);
96:         Agenda.validaTelefone (tel);
97:
98:         int posicao = this.ondeEsta (nom);
99:
100:        if (posicao > 0)
101:            throw new Exception ("Nome ja registrado");
102:
103:        posicao = (-posicao)-1;
104:
105:        for (int pos=this.qtdContatos-1; pos>=posicao; pos--)
106:        {
107:            this.nome [pos+1] = this.nome [pos];
108:            this.telefone [pos+1] = this.telefone [pos];
109:        }
110:
111:        this.nome [posicao] = nom;
112:        this.telefone [posicao] = tel;
113:
114:        this.qtdContatos++;
115:    }
116:
117:    /**
118:       Constroi uma cópia da instância da classe AgendaOrdenada dada.
119:       Para tanto, deve ser fornecida uma instancia da classe AgendaOrdenada
120:       para ser utilizada como modelo para a construção da nova instância
121:       criada.
122:       @param modelo a instância da classe AgendaOrdenada a ser usada como
123:       modelo.
```

```
124:     @throws Exception se o modelo for null.  
125:     */  
126:     public AgendaOrdenada (AgendaOrdenada modelo)  
127:             throws Exception  
128:     {  
129:         super (modelo);  
130:     }  
131:  
132:     /**  
133:      Clona uma agenda.  
134:      Produz e resulta uma cópia da agenda representada pela instância  
135:      à qual o método for aplicado.  
136:      @return a cópia da agenda representada pela instância à qual  
137:              o método for aplicado.  
138:      */  
139:     public Object clone ()  
140:     {  
141:         AgendaOrdenada copia=null;  
142:  
143:         try  
144:         {  
145:             copia = new AgendaOrdenada (this);  
146:         }  
147:         catch (Exception e)  
148:         {}  
149:  
150:         return copia;  
151:     }  
152: }
```

[C:\ExplsJava\Expl\_13\TesteDeAgenda.java]

```
1: import java.io.IOException;  
2: import java.io.BufferedReader;  
3: import java.io.InputStreamReader;  
4:  
5: import agenda.*;  
6: import agenda.ordenada.*;  
7: import agenda.desordenada.*;  
8:  
9: public class TesteDeAgenda  
10: {  
11:     public static void main (String[] args) throws IOException  
12:     {  
13:         Agenda agenda = null;  
14:  
15:         BufferedReader entrada = new BufferedReader  
16:             (new InputStreamReader  
17:                 (System.in));  
18:         int opcao=0;  
19:  
20:         for (;;) {  
21:             System.out.println ();
```

```
24:         System.out.print ("Digite sua opcao (" +
25:                         "1=Desordenada/" +
26:                         "2=Ordenada)" +
27:                         ": ");
28:
29:         try
30:         {
31:             opcao = Integer.parseInt (entrada.readLine ());
32:         }
33:         catch (IOException e)
34:         {}
35:         catch (NumberFormatException e)
36:         {
37:             System.err.println ("Nao foi digitado um numero inteiro");
38:             System.err.println ();
39:         }
40:
41:         if (opcao==1 || opcao==2)
42:             break;
43:
44:         System.err.println ("Opcao invalida!");
45:     }
46:
47:     for (;;)
48:     {
49:         System.out.println ();
50:
51:         System.out.print ("Capacidade desejada para a Agenda: ");
52:         try
53:         {
54:             int cap  = Integer.parseInt (entrada.readLine ());
55:
56:             if (opcao==1)
57:                 agenda = new AgendaDesordenada (cap);
58:             else
59:                 agenda = new AgendaOrdenada    (cap);
60:
61:             break;
62:         }
63:         catch (IOException e)
64:         {}
65:         catch (NumberFormatException e)
66:         {
67:             System.err.println ("Nao foi digitado um numero inteiro");
68:             System.err.println ();
69:         }
70:         catch (Exception e)
71:         {
72:             System.err.println (e.getMessage ());
73:             System.err.println ();
74:         }
75:     }
76:
77:     String nome = null, telefone = null;
78:
79:     do
```

```
80:         {
81:             System.out.println ();
82:
83:             System.out.print ("Digite sua Opção (" +
84:                               "I=Incluir/" +
85:                               "E=Excluir/" +
86:                               "L=Listar/" +
87:                               "S=Sair)" +
88:                               ": ");
89:
90:             try
91:             {
92:                 String str = entrada.readLine ();
93:
94:                 if (str.length() == 1)
95:                     opcao = str.charAt(0);
96:                 else
97:                     opcao = 'A'; // forçando opção inválida
98:             }
99:             catch (IOException e)
100:            {}
101:
102:             switch (opcao)
103:             {
104:                 case 'i':
105:                 case 'I':
106:                     try
107:                     {
108:                         System.out.print ("Nome....: ");
109:                         nome = entrada.readLine ();
110:                     }
111:                     catch (IOException e)
112:                     {}
113:
114:                     try
115:                     {
116:                         System.out.print ("Telefone: ");
117:                         telefone = entrada.readLine ();
118:                     }
119:                     catch (IOException e)
120:                     {}
121:
122:                     try
123:                     {
124:                         agenda.registreContato (nome, telefone);
125:                     }
126:                     catch (Exception e)
127:                     {
128:                         System.err.println (e.getMessage ());
129:                     }
130:
131:                     System.out.println ();
132:                     break;
133:
134:                 case 'e':
```

```
136:             case 'E':
137:                 System.out.print ("Nome....: ");
138:                 try
139:                 {
140:                     nome = entrada.readLine ();
141:                 }
142:                 catch (IOException e)
143:                 { }
144:
145:                 try
146:                 {
147:                     agenda.descarteContato (nome);
148:                 }
149:                 catch (Exception e)
150:                 {
151:                     System.err.println (e.getMessage ());
152:                 }
153:
154:                 System.out.println ();
155:                 break;
156:
157:
158:             case 'l':
159:             case 'L':
160:                 System.out.println (agenda);
161:                 break;
162:
163:
164:             case 's':
165:             case 'S':
166:                 break;
167:
168:
169:             default :
170:                 System.err.println ("Opcao invalida");
171:                 System.err.println ();
172:             }
173:         }
174:         while ((opcao != 's') && (opcao != 'S'));
175:     }
176: }
```

Para compilar e executar este programa, daremos os seguintes comandos:

```
C:\ExplsJava\Expl_13> javac *.java
C:\ExplsJava\Expl_13> java TesteDeAgenda
```

Isto poderia produzir no console a seguinte interação:

```
Digite sua opcao (1=Desordenada/2=Ordenada): 7
Opcao invalida!
```

Digite sua opcao (1=2vetores/2=1vetor/3=Vector/4=ArrayList): 2

Capacidade desejada para a Agenda: -10

Capacidade invalida

Capacidade desejada para a Agenda: 0

Capacidade invalida

Capacidade desejada para a Agenda: 10

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): i

Nome....: Jose

Telefone: 3241.4466

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): i

Nome....: Jose

Nome já registrado

Digite sua Opcão (I=Incluir/C=Consultar/L=Listar/E=Excluir/S=Sair): c

Nome....: Jose

Telefone: 3241.4466

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): i

Nome....: Joao

Telefone: 3252.9955

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): l

Joao (3252.9955)

Jose (3241.4466)

Digite sua Opcão (I=Incluir/C=Consultar/E=Excluir/S=Sair): e

Nome: Joao

Digite sua Opcão (I=Incluir/E=Excluir/L=Listar/S=Sair): l

Jose (3241.4466)

---

Digite sua Opcão (I=Incluir/C=Consultar/E=Excluir/S=Sair): a  
Opcão invalida

Digite sua Opcão (I=Incluir/C=Consultar/E=Excluir/S=Sair): s

## Classes Genéricas (templates)

Trata-se de um conceito avançado de programação que permite a definição de classes parametrizadas com tipos que podem ser usados para declarar variáveis, parâmetros ou retornos de método.

São especialmente úteis para a implementação de classes que atuam como depósito de informações, tais como as estruturas de dados em geral.

Veja o exemplo abaixo:

[C:\ExplsJava\Expl\_14\Deposito.java]

```
1: import java.lang.reflect.*;
2:
3: /**
4:  A classe Deposito representa um repositório genérico de itens.
5:  Instâncias desta classe permitem a realização das operações básicas de uma
6:  Nela encontramos, por exemplo, métodos para guardar itens, jogar fora itens
7:  verificar a presença de itens no repositório e listar itens.
8:  @author André Luís dos Reis Gomes de Carvalho.
9:  @since 2016.
10: */
11: public class Deposito <X> implements Cloneable // x é forçosamente uma classe
12: {
13:     /**
14:      Mantém armazenados os itens armazenados no deposito.
15:     */
16:     private Object[] elem;
17:
18:     /**
19:      Expressa, em cada instante, a posição na qual se encontra amazenado o último
20:      do deposito.
21:     */
22:     private int ultimo;
23:
24:     /**
25:      Constroi uma nova instância da classe genérica Deposito.
26:      Para tanto, deve ser fornecido um inteiro que será utilizado
```

```
27:     como capacidade da instância recém criada.
28:     @param tam número inteiro a ser utilizado como capacidade.
29:     @throws Exception se a capacidade for negativa ou zero.
30:     */
31:     public Deposito (int tam) throws Exception
32:     {
33:         if (tam<=0)
34:             throw new Exception ("Tamanho invalido");
35:
36:         elem = new Object [tam];
37:         ultimo = -1;
38:     }
39:
40:     /**
41:      Armazena um novo ítem em um depósito.
42:      Acrescenta um novo ítem na instância à qual este método for aplicado.
43:      @param i o ítem a ser guardado.
44:      @throws Exception se o deposito estiver cheio.
45:      */
46:     public void guarde (X i) throws Exception
47:     {
48:         if (ultimo==elem.length-1)
49:             throw new Exception ("Deposito cheio");
50:
51:         ultimo++;
52:
53:         if (i instanceof Cloneable)
54:         {
55:             Class classe = i.getClass();
56:             Class<?>[] parmsFormais = null; // = null é desnecessário
57:             Method metodo = classe.getMethod ("clone", parmsFormais);
58:             Object[] parmsReais = null;
59:             elem[ultimo] = (X)metodo.invoke(i,parmsReais);
60:
61:             //elem[ultimo] = (X)i.clone();
62:         }
63:         else
64:             elem[ultimo] = i;
65:     }
66:
67:     /**
68:      Verifica se um certo ítem se encontra armazenado.
69:      Resulta um valor lógico, verdadeiro ou falso, conforme o ítem fornecido
70:      possa ou não ser encontrado no depósito.
71:      @return a possibilidade de encontrar o ítem dado.
72:      @throws Exception se não for fornecido o item a ser procurado.
73:      */
74:     public boolean tem (X i) throws Exception
75:     {
76:         if (i==null)
77:             throw new Exception ("Ítem a pesquisar não fornecido");
78:
79:         for (int pos=0; pos<=ultimo; pos++)
80:             if (elem[pos].equals(i))
81:                 return true;
82:
```

```
83:         return false;
84:     }
85:
86:     /**
87:      Recupera um ítem armazenado em uma posição fornecida.
88:      Recupera um ítem, dado o número de ordem dele no depósito.
89:      @return o ítem recuperado.
90:      @throws Exception o número de ordem fornecido não se referir a uma posição
91:                      no depósito.
92:     */
93:     public X getElem (int pos) throws Exception
94:     {
95:         if (pos<0 || pos>this.ultimo)
96:             throw new Exception ("Posicao invalida");
97:
98:         if (this.elem[pos] instanceof Cloneable)
99:         {
100:             Class classe          = this.elem[pos].getClass();
101:             Class<?>[] parmsFormais = null; // = null é desnecessário
102:             Method metodo          = classe.getMethod ("clone", parmsFormais);
103:             Object[] parmsReais    = null;
104:             return (X)metodo.invoke(this.elem[pos],parmsReais);
105:
106:             //return (X)this.elem[pos].clone();
107:         }
108:         else
109:             return (X)this.elem[pos];
110:     }
111:
112:     /**
113:      Remove um ítem do depósito.
114:      Exclui da instância à qual este método for aplicado o ítem fornecido.
115:      @param nom o ítem a ser jogado fora.
116:      @throws Exception se não for fornecido um ítem a ser descartato.
117:     */
118:     public void jogueFora (X i) throws Exception
119:     {
120:         int pos;
121:
122:         if (ultimo== -1)
123:             throw new Exception ("Deposito vazio");
124:
125:         for (pos=0; pos<=ultimo; pos++)
126:             if (elem[pos].equals(i))
127:                 break;
128:
129:         if (pos==ultimo+1)
130:             throw new Exception ("Valor inexistente");
131:
132:         for ( ; pos<ultimo; pos++)
133:             elem[pos] = elem[pos+1];
134:
135:         ultimo--;
136:     }
137:
138:     /**
```

```
139:     Verifica a igualdade entre dois depósitos.
140:     Verifica se o Object fornecido como parâmetro representa um
141:     depósito igual àquele representada pela instância à qual este
142:     método foi aplicado, resultando true em caso afirmativo,
143:     ou false, caso contrário.
144:     @param obj o objeto a ser comparado com a instância à qual esse método
145:             foi aplicado.
146:     @return true, caso o Object fornecido ao método e a instância chamante do
147:             método representarem depósitos iguais, ou false, caso contrário.
148:     */
149:     public boolean equals (Object obj)
150:     {
151:         if (this==obj)
152:             return true;
153:
154:         if (obj==null)
155:             return false;
156:
157:         //if (!(obj instanceof Deposito<X>))
158:         if (this.getClass() != obj.getClass())
159:             return false;
160:
161:         Deposito dep = (Deposito)obj;
162:
163:         if (this.ultimo!=dep.ultimo)
164:             return false;
165:
166:         for (int i=0; i<=this.ultimo; i++)
167:             if (!this.elem[i].equals(dep.elem[i]))
168:                 return false;
169:
170:         return true;
171:     }
172:
173:     /**
174:      Gera uma representação textual de todo conteúdo do depósito.
175:      Produz e resulta um String com todos os itens contidos no deposito.
176:      @return um String contendo todo o conteúdo do depósito.
177:      */
178:     public String toString ()
179:     {
180:         String ret="";
181:
182:         for (int i=0; i<this.ultimo; i++)
183:             ret += this.elem[i]+",";
184:
185:         if (this.ultimo>-1)
186:             ret += this.elem[this.ultimo];
187:
188:         ret += "}";
189:
190:         return ret;
191:     }
192:
193:     /**
194:      Calcula o código de espalhamento (ou código de hash) de um depósito.
```

```
195:     Calcula e resulta o código de espalhamento (ou código de hash, ou ainda o
196:     hashCode) do depósito representado pela instância à qual o método for aplicado.
197:     @return o código de espalhamento do depósito chamante do método.
198:     */
199:    public int hashCode ()
200:    {
201:        final int O_PRIMO_ESCOLHIDO = 13;
202:
203:        int ret = super.hashCode ();
204:
205:        ret = ret*O_PRIMO_ESCOLHIDO + new Integer(this.ultimo).hashCode(); // como
206:
207:        for (int i=0; i<=this.ultimo; i++)
208:            ret = ret*O_PRIMO_ESCOLHIDO + this.elem[i].hashCode();
209:
210:        return ret;
211:    }
212:
213:    /**
214:     Constroi uma cópia da instância da classe Deposito dada.
215:     Para tanto, deve ser fornecida uma instancia da classe Depósito para ser
216:     utilizada como modelo para a construção da nova instância criada.
217:     @param modelo a instância da classe Depósito a ser usada como modelo.
218:     @throws Exception se o modelo for null.
219:     */
220:    public Deposito (Deposito<X> modelo) throws Exception
221:    {
222:        if (modelo==null)
223:            throw new Exception ("Modelo não fornecido");
224:
225:        this.elem = new Object [modelo.elem.length];
226:
227:        for (int i=0; i<=modelo.ultimo; i++)
228:            if (modelo.elem[i] instanceof Cloneable)
229:            {
230:                Class classe          = modelo.elem[i].getClass();
231:                Class<?>[] parmsFormais = null; // = null é desnecessário
232:                Method metodo          = classe.getMethod ("clone", parmsFormais);
233:                Object[] parmsReais    = null;
234:                this.elem[i]           =
235:
236:                //this.elem[i] = (x)modelo.elem[i].clone();
237:            }
238:            else
239:                this.elem[i] = modelo.elem[i];
240:
241:        this.ultimo = modelo.ultimo;
242:    }
243:
244:    /**
245:     Clona um depósito.
246:     Produz e resulta uma cópia do depósito representado pela instância
247:     à qual o método for aplicado.
248:     @return a cópia do depósito representado pela instância à qual
249:             o método for aplicado.
250:     */
```

```
251:     public Object clone ()
252:     {
253:         Deposito<X> ret=null;
254:
255:         try
256:         {
257:             ret = new Deposito<X> (this);
258:         }
259:         catch (Exception erro)
260:         {} // tenho certeza que nao vai acontecer
261:
262:         return ret;
263:     }
264: }
```

[C:\ExplsJava\Expl\_14\TesteDeDeposito.java]

```
1: import java.io.IOException;
2: import java.io.BufferedReader;
3: import java.io.InputStreamReader;
4:
5: public class TesteDeDeposito
6: {
7:     public static void main (String[] args) throws IOException
8:     {
9:         Deposito<String> deposito = null;
10:
11:        BufferedReader entrada = new BufferedReader
12:                      (new InputStreamReader
13:                      (System.in));
14:
15:        for (;;)
16:        {
17:            System.out.print ("Capacidade desejada para o Deposito: ");
18:            try
19:            {
20:                int cap = Integer.parseInt (entrada.readLine ());
21:                deposito = new Deposito<String> (cap);
22:                System.out.println ();
23:                break;
24:            }
25:            catch (IOException e)
26:            {}
27:            catch (NumberFormatException e)
28:            {
29:                System.err.println ("Nao foi digitado um numero inteiro\n");
30:                System.err.println ();
31:            }
32:            catch (Exception e)
33:            {
34:                System.err.println (e.getMessage () +"\n");
35:                System.err.println ();
36:            }
37:        }
38:    }
```

```
39:         char    opcao=' ';
40:         String item = null;
41:
42:         do
43:         {
44:             System.out.println ();
45:
46:             System.out.print ("Digite sua Opção (" +
47:                               "G=Guardar/" +
48:                               "I=obter Item/" +
49:                               "T=ver se Tem/" +
50:                               "J=Jogar fora/" +
51:                               "L=Listar/" +
52:                               "S=Sair)" +
53:                               ": ");
54:
55:             try
56:             {
57:                 String str = entrada.readLine ();
58:
59:                 if (str.length()==1)
60:                     opcao = str.charAt(0);
61:                 else
62:                     opcao = 'A'; // forçando opção inválida
63:             }
64:             catch (IOException e)
65:             {}
66:
67:             switch (opcao)
68:             {
69:                 case 'g':
70:                 case 'G':
71:                     try
72:                     {
73:                         System.out.print ("Item: ");
74:                         item = entrada.readLine ();
75:                     }
76:                     catch (IOException e)
77:                     {}
78:
79:                     try
80:                     {
81:                         deposito.guarda(item);
82:                     }
83:                     catch (Exception e)
84:                     {
85:                         System.err.println (e.getMessage ());
86:                     }
87:
88:                     System.out.println ();
89:                     break;
90:
91:                 case 't':
92:                 case 'T':
93:                     System.out.print ("Item: ");
```

```
95:             try
96:             {
97:                 item = entrada.readLine ();
98:
99:                 if (deposito.tem(item))
100:                     System.out.println ("Tem");
101:                 else
102:                     System.out.println ("Nao tem");
103:             }
104:             catch (Exception e)
105:             { }
106:
107:             System.out.println ();
108:             break;
109:
110:
111:             case 'i':
112:             case 'I':
113:                 int pos;
114:
115:                 for (;;)
116:                 {
117:                     System.out.print ("Numero de ordem: ");
118:                     try
119:                     {
120:                         pos = Integer.parseInt (entrada.readLine ());
121:
122:                         if (pos<0)
123:                             System.err.println ("Nao foi digitato um numero");
124:                         else
125:                             break;
126:                     }
127:                     catch (IOException e)
128:                     { }
129:                     catch (NumberFormatException e)
130:                     {
131:                         System.err.println ("Nao foi digitado um numero");
132:                     }
133:                 }
134:
135:                 try
136:                 {
137:                     item = deposito.getElem(pos);
138:                     System.out.println (item);
139:                 }
140:                 catch (Exception e)
141:                 {
142:                     System.err.println (e.getMessage ());
143:                 }
144:
145:                 System.out.println ();
146:                 break;
147:
148:
149:             case 'j':
150:             case 'J':
```

```
151:             System.out.print ("Item: ");
152:             try
153:             {
154:                 item = entrada.readLine ();
155:             }
156:             catch (IOException e)
157:             {}
158:
159:             try
160:             {
161:                 deposito.jogueFora(item);
162:             }
163:             catch (Exception e)
164:             {}
165:                 System.err.println (e.getMessage ());
166:             }
167:
168:             System.out.println ();
169:             break;
170:
171:
172:             case 'l':
173:             case 'L':
174:                 System.out.println (deposito);
175:                 System.out.println ();
176:                 break;
177:
178:
179:             case 's':
180:             case 'S':
181:                 break;
182:
183:
184:             default :
185:                 System.err.println ("Opcao invalida");
186:                 System.err.println ();
187:             }
188:         }
189:         while ((opcao != 's') && (opcao != 'S'));
190:     }
191: }
```

Para compilar e executar este programa, daremos os seguintes comandos:

```
C:\ExplsJava\Expl_14> javac *.java
C:\ExplsJava\Expl_14> java TesteDeDeposito
```

Isto poderá produzir no console a seguinte saída:

```
Capacidade desejada para o Deposito: bla bla bla
Nao foi digitado um numero inteiro
```

Capacidade desejada para o Deposito: -7  
Tamanho invalido

Capacidade desejada para o Deposito: 3

Digite sua Opcão (G=Guardar/I=obter Item/T=ver se Tem/J=Jogar fora/L=Listar/S=Sair): g  
Item: Andre

Digite sua Opcão (G=Guardar/I=obter Item/T=ver se Tem/J=Jogar fora/L=Listar/S=Sair): g  
Item: Luis

Digite sua Opcão (G=Guardar/I=obter Item/T=ver se Tem/J=Jogar fora/L=Listar/S=Sair): g  
Item: Carvalho

Digite sua Opcão (G=Guardar/I=obter Item/T=ver se Tem/J=Jogar fora/L=Listar/S=Sair): g  
Item: Reis  
Deposito cheio

Digite sua Opcão (G=Guardar/I=obter Item/T=ver se Tem/J=Jogar fora/L=Listar/S=Sair): i  
Número de ordem: 1  
Luis

Digite sua Opcão (G=Guardar/I=obter Item/T=ver se Tem/J=Jogar fora/L=Listar/S=Sair): t  
Item: Carvalho  
Tem

Digite sua Opcão (G=Guardar/I=obter Item/T=ver se Tem/J=Jogar fora/L=Listar/S=Sair): t  
Item: Reis  
Nao tem

Digite sua Opcão (G=Guardar/I=obter Item/T=ver se Tem/J=Jogar fora/L=Listar/S=Sair): l  
{Andre,Luis,Carvalho}

---

Digite sua Opcão (G=Guardar/I=obter Item/T=ver se Tem/J=Jogar fora/L=Listar/S=Sair): j  
Item: Luis

Digite sua Opcão (G=Guardar/I=obter Item/T=ver se Tem/J=Jogar fora/L=Listar/S=Sair): 1  
{Andre,Carvalho}

Digite sua Opcão (G=Guardar/I=obter Item/T=ver se Tem/J=Jogar fora/L=Listar/S=Sair): a  
Opcão invalida

Digite sua Opcão (G=Guardar/I=obter Item/T=ver se Tem/J=Jogar fora/L=Listar/S=Sair): s

## Recursão

Sabemos que o conceito de recursão se encontra entre os mais poderosos recursos de programação de que se dispõe e a linguagem Java, sendo uma linguagem completa, não poderia deixar de suportá-lo.

Trata-se da possibilidade de escrever funções que ativam outras instâncias de execução de si próprias na implementação de suas funcionalidades

Como pressupomos um bom conhecimento de programação em alguma linguagem estruturada completa (como Pascal ou C, por exemplo), entendemos que o estudo minucioso desta técnica de programação foge ao escopo deste texto e, por isso, limitar-nos-emos a mencionar sua existência a assumir que o leitor saiba como empregá-la.

## Anexo I

# Exercícios

## I. Classes e Objetos

1. Responda verdadeiro ou falso e justifique: a definição de uma classe reserva espaço de memória para conter todos os dados presentes em sua definição.
2. A finalidade de definir classes é:
  - a) Reservar uma quantidade de memória;
  - b) Indicar que o programa é orientado a objetos;
  - c) Agrupar dados e funções protegendo-os do compilador;
  - d) Descrever novos tipos abstratos de dados antes desconhecidos do compilador.
3. A relação entre classes, objetos e instâncias é a mesma existente entre:
  - a) Tipos básicos, variáveis e valores desses tipos;
  - b) Variáveis, funções e procedimentos;
  - c) Constantes, variáveis e valores;
  - d) Constantes, variáveis e tipos básicos.
4. Os membros de classe designados como privativos na definição de uma classe:
  - a) São acessíveis por qualquer função do programa;
  - b) São acessíveis por qualquer função de sua classe;
  - c) São acessíveis apenas para a funções de classe de sua classe;

- d) São acessíveis apenas para a funções de instância de sua classe.
5. Os membros de instância designados como privativos na definição de uma classe:
- São acessíveis por qualquer função do programa;
  - São acessíveis por qualquer função de sua classe;
  - São acessíveis apenas para a funções de classe de sua classe;
  - São acessíveis apenas para a funções de instância de sua classe.
6. Responda verdadeiro ou falso: membros privativos definem a atividade interna da classe e de suas instâncias, enquanto os membros públicos definem o padrão de comunicação da classe e de suas instâncias com o mundo exterior a elas.
7. Para acessar um membro de classe presente na definição de uma classe, o operador ponto conecta:
- O nome da classe e o nome do membro;
  - O nome do membro e o nome de um objeto da classe;
  - O nome de um objeto da classe e o nome do membro;
  - O nome da classe e o nome de um objeto da classe.
8. Para acessar um membro de instância presente na definição de uma classe, o operador ponto conecta:
- O nome da classe e o nome do membro;
  - O nome do membro e o nome de um objeto da classe;
  - Uma instância da classe, necessariamente em um objeto da classe, e o nome do membro.
  - Uma instância da classe, eventualmente em um objeto da classe, e o nome do membro.
9. Métodos são:
-

- a) Classes;
- b) Dados-membro;
- c) Funções-membro;
- d) Chamadas a funções-membro.

10. Mensagens são:

- a) Classes;
- b) Dados-membro;
- c) Funções-membro;
- d) Chamadas a funções-membro.

11. Construtores são funções:

- a) Que constroem classes;
- b) Executadas automaticamente quando uma instância é criada;
- c) Executadas automaticamente quando um objeto é declarado;
- d) Executadas automaticamente quando uma instância é destruída.

12. O nome de um construtor é sempre \_\_\_\_\_.

13. Responda verdadeiro ou falso: numa classe, é possível haver mais de um construtor de mesmo nome.

14. Responda verdadeiro ou falso: um construtor é do tipo retornado por meio do comando *return*.

15. Responda verdadeiro ou falso: um construtor não pode ter argumentos.

16. Destrutores são funções:

- a) Que destroem classes;

- b) Executadas automaticamente quando uma instância é criada;
  - c) Executadas automaticamente quando um objeto é declarado;
  - d) Executadas automaticamente quando uma instância é destruída.
17. Assuma que C1, C2 e C3 sejam objetos de uma mesma classe. Quais das seguintes instruções são válidas?
- a) C1 = C2;
  - b) C1 = C2 + C3;
  - c) C1 = C2 = C3;
  - d) C1 = C2 + 7;
18. Um método de classe pode apenas acessar os dados:
- a) Da instância à qual está associada;
  - b) Da classe onde foi definida;
  - c) Da classe e de qualquer instância da classe onde foi definida;
  - d) Da parte pública da classe onde foi definida.
19. Um método de instância pode apenas acessar os dados:
- a) Da instância à qual está associada;
  - b) Da classe onde foi definida;
  - c) Da classe e de qualquer instância da classe onde foi definida;
  - d) Da parte pública da classe onde foi definida.
20. Suponha que cinco instâncias de uma classe sejam criadas. Quantas cópias dos dados-membro de classe de sua classe serão armazenadas na memória? Quantas cópias dos dados-membro de instância de sua classe serão armazenadas na memória? Quantas cópias de suas funções-membro de classe de sua classe serão
-

armazenadas na memória? Quantas cópias de suas funções-membro de instância de sua classe serão armazenadas na memória?

21. Escreva uma classe chamada Horario com 3 membros de instância inteiros chamados Horas, Minutos e Segundos. Todas as validações cabíveis de serem realizadas por seus métodos deverão ser feitas e exceções deverão ser lançadas no caso de incorretudes serem detectadas.

- Inclua um construtor que, recebendo como parâmetros 3 valores inteiros, inicie os dados internos da instância à qual se refere;
- Crie getters e setters;
- Crie um método chamado mais que, recebendo uma instância da classe Horario, some-a com a instância à qual se refere, retornando uma instância da classe Horario que represente o resultado;
- Crie um método chamado menos que, recebendo uma instância da classe Horario, o subtraia da instância à qual se refere, retornando uma instância da classe Horario que represente o resultado;
- Crie os métodos canônicos necessários e cabíveis.

22. Escreva uma classe chamada Angulo. Suas instâncias deverão ser capazes de armazenar um valor angular expresso em graus (este será o único atributo). Todas as validações cabíveis de serem realizadas por seus métodos deverão ser feitas e exceções deverão ser lançadas no caso de incorretudes serem detectadas.

- Escreva um construtor que, recebendo um valor angular expresso em graus, o armazena internamente;
  - Escreva métodos de instância chamadas getValorEmGraus, getValorEmGrados e getValorEmRadianos que, sem receber parâmetros, retornam, respectivamente, o valor angular do objeto expresso em graus, grados e radianos.
-

- Escreva métodos de instância chamadas setValorEmGraus, setValorEmGrados e setValorEmRadianos que, recebendo como parâmetro um valor angular expresso, respectivamente, em graus, em grados e em radianos, ajustam o valor angular, sempre expresso em graus, da instância chamante.
- Crie os métodos canônicos necessários e cabíveis.

23. Escreva uma classe chamada Circulo. Suas instâncias deverão ser capazes de armazenar as coordenadas do centro, bem como o valor do raio de um círculo. Todas as validações cabíveis de serem realizadas por seus métodos deverão ser feitas e exceções deverão ser lançadas no caso de incorretudes serem detectadas.

- Escreva um construtor que, recebendo números reais expressando as coordenadas e o valor do raio de um círculo, armazena essas informações internamente.
- Escreva um método chamado getComprimento que, recebendo um objeto da classe Angulo, retorna o comprimento do arco do círculo com aquela varredura angular.
- Escreva um método chamado getArea que, recebendo um objeto da classe Angulo, retorna a área do setor do círculo com aquela varredura angular.
- Escreva um método chamado setCentro que, recebendo como parâmetro dois números reais expressando as coordenadas do círculo, armazena estas informações internamente.
- Escreva um método chamado setRaio que, recebendo como parâmetro um número real expressando o raio do círculo, armazena esta informação internamente.
- Crie os métodos canônicos necessários e cabíveis.

24. Suponha a existência uma classe Sensor que implementa funções de instância capazes de controlar um sensor e que dispõe de (1) um construtor que, recebendo

---

um inteiro que representa o número de seu identificador, inicia apropriadamente a instância à qual se refere; e (2) um método de instância chamado isAtivado sem parâmetros que retorna valor lógico que indica a detecção ou não de algo se movendo em seu “campo de visão”.

Suponha a existência uma classe Porta que dispõe de (1) um construtor que, recebendo um inteiro que representa o número de seu identificador, inicia apropriadamente a instância à qual se refere; (2) uma função (de nome abraSe) sem parâmetros e sem retorno que, quando executada, faz com que a porta se abra; (3) uma função (de nome fecheSe) sem parâmetros e sem retorno que, quando executada, faz com que a porta feche; e (4) uma função (de nome isAberta) sem parâmetros e que retorna true, caso a porta esteja aberta, ou false, caso contrário.

Valendo-se dessas duas classes, crie a classe PortaAuto para representar uma porta automática como as que costumamos encontrar em aeroportos. Pede-se:

- Implemente um construtor para inicializar apropriadamente as instâncias das classes Sensor e Porta que existem no interior da instância à qual se refere;
- Implemente um método chamado entreEmOperacao que, sem receber parâmetros e nem produzir retornos, faz com que a porta automática entre em operação ininterrupta.

ATENÇÃO: a porta nunca deve ser aberta, quando já se encontra aberta, e nem fechada, quando já se encontra fechada.

25. Suponha a existência uma classe Sensor que implementa funções de instância capazes de controlar um sensor de ficha e que dispõe de (1) um construtor que, recebendo um inteiro que representa o número de seu identificador, inicia apropriadamente a instância à qual se refere; e (2) um método de instância chamado isAtivado sem parâmetros que retorna um valor lógico que, se for verdadeiro, indica que o sensor detectou o depósito de uma ficha, e, se for falso, indica que o sensor não detectou o depósito de uma ficha (essa função retorna verdadeiro apenas uma vez por ficha depositada).
-

Suponha a existência uma classe Cancela que implementa funções capazes de controlar uma cancela eletrônica e que dispõe de (1) um construtor que, recebendo um inteiro que representa o número de seu identificador, inicia apropriadamente a instância à qual se refere; e (2) um método de instância chamado abraSe, sem parâmetros e sem retorno, que, quando executada, faz com que a cancela se abra para a passagem de um veículo, fechando-se automaticamente após a passagem deste.

O objetivo deste exercício é, valendo-se dessas duas classes, implementar, DE FORMA COMPLETA E ADEQUADA, a classe EstacaoDePedagio que implementa funções capazes de controlar uma estação automática de cobrança de pedágio. Pede-se:

- Declarar objetos privativos da classe Sensor e Cancela;
- Implementar um construtor que, recebendo 2 inteiros (um deles para o construtor de Sensor e o outro para o construtor de Cancela), inicia apropriadamente a instância à qual se refere; e
- Implementar um método chamado funcione que, sem receber parâmetros e sem produzir retornos, faz com que a estação automática de cobrança de pedágio passe a funcionar de forma ininterrupta.

## II. Pacotes

1. Assuma que as classes C1 e C2 estejam definidas em pacotes diferentes. Para que C1 possa ser empregada na definição de C2, é necessário que C1 seja \_\_\_\_\_.
  2. Para que membros definidos em uma classe possam ser acessados em funções da classe que se encontra no mesmo pacote, eles devem ter sido declarados como:
    - a) public ou private;
    - b) protected ou private;
-

- c) public ou protected;
  - d) Nenhuma das anteriores.
3. Para que membros definidos em uma classe pública possam ser acessados em funções da definidas em uma classe que se encontra em outro pacote, eles devem ter sido declarados como:
- a) public;
  - b) protected;
  - c) private;
  - d) Todas as anteriores.

### III. Herança

1. Herança é um processo que permite:
  - a) A inclusão de um objeto dentro de outro;
  - b) Transformar classes genéricas em classes mais específicas;
  - c) Implementar uma classe semelhante a uma classe existente sem a necessidade de copiá-la ou reescrevê-la
  - d) Relacionar objetos por meio de seus argumentos.
2. As vantagens do uso de herança incluem:
  - a) Aproveitamento de classes existentes na elaboração de novas classes;
  - b) Uso de bibliotecas;
  - c) Concepção top-down de algorítmos;
  - d) Melhor uso da memória.
3. Para derivar uma classe de outra já existente, deve-se:

- a) Alterar a classe existente;
  - b) Reescrever a classe existente;
  - c) Indicar que a nova classe incorpora as características da classe existente, acrescentando à nova classe novas características ou redefinindo na nova classe características herdadas;
  - d) Nenhuma das anteriores.
4. Se em uma classe-base foi definida uma função de instância chamada F e em uma classe dela derivada não foi definida nenhuma função com este nome, responda: em que situação a referida função chamada F poderá ser aplicada a uma instância da classe derivada?
5. Responda verdadeiro ou falso: suponha que em uma classe-base foi definida uma função chamada F e em uma classe dela derivada também foi definida uma função chamada F, com os mesmos parâmetros e retorno. Uma outra função definida na classe-derivada pode empregar a função herdada apesar da redefinição.
6. Responda verdadeiro ou falso: suponha que em uma classe-base foi definida uma função chamada F e em uma classe dela derivada também foi definida uma função chamada F, com os mesmos parâmetros e retorno. Uma outra função definida fora da classe-derivada pode empregar a função herdada apesar da redefinição.
7. Classes derivadas incorporam:
- a) Todos os membros da classe-base;
  - b) Somente os membros públicos da classe-base;
  - c) Somente os membros protegidos da classe-base;
  - d) O segundo e o terceiro item são verdadeiros.
8. Responda verdadeiro ou falso: se nenhum construtor existir na classe derivada, objetos desta classe usarão o construtor sem argumentos da classe-base.
-

9. Responda verdadeiro ou falso: se nenhum construtor existir na classe derivada, objetos desta classe poderão iniciar os dados herdados de sua classe-base usando o construtor com argumentos da classe-base.
10. Uma classe é dita abstrata quando:
  - a) Nenhum objeto dela é declarado;
  - b) É representada apenas mentalmente;
  - c) Só pode ser usada como base para outras classes;
  - d) É definida de forma obscura.
11. A conversão de tipos implícita é usada para:
  - a) Armazenar instâncias de uma classe-derivada em objetos de sua classe-base;
  - b) Armazenar instâncias de uma classe-base em objetos de uma classe dela derivada;
  - c) Armazenar instâncias de uma classe-derivada em objetos de sua classe-base e vice-versa;
  - d) Não pode ser usada para conversão de objetos.
12. Responda verdadeiro ou falso: uma classe derivada não pode servir de base para outra classe.
13. Responda verdadeiro ou falso: um objeto de uma classe pode ser membro de outra classe.
14. Suponha implementada uma classe chamada Pto para representar um ponto no plano cartesiano. Todas as validações cabíveis de serem realizadas por seus métodos são feitas e exceções são lançadas no caso de incorretudes serem detectadas. As estruturas internas desta classe supostamente são desconhecidas e seus métodos públicos devem ser considerados como sendo os seguintes:

- Um construtor que, recebendo um par de números reais, respectivamente as coordenadas de um ponto no plano cartesiano, inicie apropriadamente a instância à qual se refere;
- Getters e setters.
- Crie os métodos canônicos necessários e cabíveis.

O objetivo desta questão é implementar uma classe chamada Ponto, derivada da classe Pto, para representar, de forma mais completa, um ponto no plano cartesiano.

Todas as validações cabíveis de serem realizadas por seus métodos deverão ser feitas e exceções deverão ser lançadas no caso de incorretudes serem detectadas.

Sabendo que:

- A distância entre os pontos  $P_1 = (x_1, y_1)$  e  $P_2 = (x_2, y_2)$  é dada pela seguinte expressão:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- A inclinação da reta que passa pelos pontos  $P_1 = (x_1, y_1)$  e  $P_2 = (x_2, y_2)$  é dada pela seguinte expressão:

$$\frac{(y_2 - y_1)}{(x_2 - x_1)}$$

**Pede-se:**

- Declare a classe Ponto e as estruturas internas que a definem;
  - Implemente um método de instância chamado getDistancia que, recebendo como parâmetro uma instância da classe Ponto, retorna a distância do ponto representado pela instância à qual se refere até o ponto representado pela instância recebida como parâmetro;
-

- Implemente um método de instancia chamado getInclinacao que retorna a inclinação da reta imaginária que une o ponto representado pela instancia à qual se refere à origem do sistema cartesiano.
  - Crie os métodos canônicos necessários e cabíveis.
15. Suponha a existência uma classe Sensor que dispõe de (1) um construtor que, recebendo um inteiro que representa o número de seu identificador, inicia apropriadamente a instância à qual se refere; e (2) um método (de nome getVelocidade) sem parâmetros que retorna um real que indica a velocidade do objeto detectado (indicará sempre zero no caso de nada estar sendo detectado).

Suponha a existência uma classe Camera que dispõe de (1) um construtor que, recebendo um inteiro que representa o número de seu identificador, inicia apropriadamente a instância à qual se refere; (2) uma função (de nome fotografe) que recebe um parâmetro real e sem retorno que, quando executada, faz com que a câmera bata uma foto na qual aparecerá no canto inferior direito o horário em que a foto foi batida (a câmera tem um relógio interno) e String que vem a esta função como parâmetro.

Pede-se:

- Derivada da classe Sensor e valendo-se da classe Camera, crie a classe Radar para representar um radar como estes que encontramos instalados nas principais ruas e avenidas das grandes cidades;
  - Implemente um construtor que recebe como parâmetro 2 números inteiros (um deles para o construtor de Sensor e o outro para o construtor de Camera) e um número real que representa a velocidade máxima da via na qual o radar vai ser instalado, iniciando apropriadamente a instância à qual se refere;
  - Implemente uma função de instância pública, chamada entreEmAcao, que coloca em operação de forma ininterrupta uma instância da classe Radar.
-

### III. Classes Genéricas (templates)

1. Escreva uma classe chamada Conjunto cujas instâncias descrevam conjuntos, conforme os conhecemos da matemática.
  - Escreva um construtor que, sem receber parâmetros, inicia a instância à qual se refere com o conjunto vazio.
  - Escreva um método de instância chamado inclua que, recebendo um valor inteiro, promove a inclusão deste na instância à qual se refere.
  - Escreva um método de instância chamado elimine que, recebendo um valor inteiro, o remove da instância à qual se refere.
  - Escreva um método de instância chamado intersecao que, recebendo uma instância da classe Conjunto, produz e retorna uma instância da classe conjunto que representa a interseção da instância à qual se refere com a instância recebida.
  - Escreva um método de instância chamado uniao que, recebendo uma instância da classe Conjunto, produz e retorna uma instância da classe conjunto que representa a união da instância à qual se refere com a instância recebida.
  - Escreva um método de instância chamado possui que, recebendo como parâmetro um valor inteiro, verifica se o mesmo pertence à instância à qual se refere, retornando o valor lógico verdadeiro, em caso afirmativo, ou o valor lógico falso, caso contrário.
  - Implemente um método de instância chamado toString que, sem receber nenhum parâmetro, produz e retorna um String que representa textualmente a instância à qual se refere.
  - Crie os métodos canônicos necessários e cabíveis.

## Anexo II

# Bibliografia

SUN Microsystems, "The Java™ Tutorial", (<http://java.sun.com/docs/books/tutorial/>).

SUN Microsystems, "The Java™ Language Specification",  
(<http://java.sun.com/docs/books/jls/index.html>).

SUN Microsystems, "The Java™ API Specification",  
(<http://java.sun.com/products/jdk/1.2/docs/api/index.html>).

SUN Microsystems, "The Java™ VM Specification",  
(<http://java.sun.com/docs/books/vmspec/index.html>).

Naughton, Patrick, "Dominando o Java", Makron Books, 1997.

Morrison, M.; "Java Unleashed", December, J.; et alii; SAMS Publishing.

Naughton, P.; "Dominando o Java"; Makron Books.

Linden, P.; "Just Java"; Makron Books.

Fraizer, C.; e Bond, J.; "API Java: Manual de Referência"; Makron Books.

Thomas, M.D.; Patel, P.r.; et alii; "Programando em Java para a Internet"; Makron Books.

Ritchey, T.; "Programando Java e JavaScript"; Quark Editora.

---