PROGRAMAÇÃO I

AUTOR
FABIANO GONÇALVES DOS SANTOS

PROGRAMAÇÃO I

AUTOR
FABIANO GONÇALVES DOS SANTOS

1ª EDIÇÃO
SESES
RIO DE JANEIRO 2017

Estácio

Conselho editorial ROBERTO PAES E LUCIANA VARGA

Autor do original FABIANO GONÇALVES DOS SANTOS

Projeto editorial ROBERTO PAES

Coordenação de produção LUCIANA VARGA, PAULA R. DE A. MACHADO E ALINE KARINA RABELLO

Projeto gráfico PAULO VITOR BASTOS

Diagramação BFS MEDIA

Revisão linguística BFS MEDIA

Revisão de conteúdo OSWALDO BORGES PERES

Imagem de capa RAWPIXEL.COM | SHUTTERSTOCK.COM

Dados Internacionais de Catalogação na Publicação (CIP)

S237p SANTOS, FABIANO GONÇALVES DOS
Programação I. / Fabiano Gonçalves dos Santos.
Rio de Janeiro: SESES, 2017.
160 p: il.

ISBN: 978-85-5548-429-2

1. Linguagem Java. 2. Programação orientada a objetos. 3. Tratamento de exceções. 4.Coleções. I.SESES. II. Estácio.

CDD 004

Diretoria de Ensino — Fábrica de Conhecimento
Rua do Bispo, 83, bloco F, Campus João Uchôa
Rio Comprido — Rio de Janeiro — RJ — CEP 20261-063

Sumário

Prefácio

Prezados(as) alunos(as),

Uma vez que conhecemos a forma de montar programas por meio dos algoritmos, suas estruturas e controles fundamentais, é hora de aprendermos uma linguagem de programação para poder aplicar os algoritmos em situações reais e resolver problemas.

Nesta disciplina, vamos conhecer e estudar a linguagem Java que é sem dúvida uma das linguagens mais utilizadas no mundo. Com ela será possível futuramente aprender a desenvolver sistemas de backend para sites, desenvolver aplicações para sistemas que rodam em desktop e desenvolver sistemas para sistemas móveis, além de outras aplicações menos tradicionais (muitos eletrodomésticos usam programas em Java para funcionar, sabia?).

A linguagem Java é, sem dúvida, muito poderosa e estudá-la será um motivador para os seus conhecimentos e carreira futura. De acordo com alguns índices na internet, como o Tiobe Index, por exemplo (http://www.tiobe.com/tiobe-index/), ela é a líder de popularidade há anos, pelo menos, desde 2002.

Falando em orientação a objetos, vamos estudar este conceito fundamental e importantíssimo para todos aqueles que desejam trabalhar com programação de computadores. Este conceito é super importante para entender não somente a linguagem Java, mas outras também que são naturalmente orientadas a objeto como Python, C#, PHP e tantas outras. É o estado da arte em programação atualmente.

Como já dito, vale a pena estudá-la. Esforce-se, concentre-se, pois aprender uma nova linguagem de programação é como aprender um novo idioma: você sabe muitas vezes o que quer falar, mas é necessário conhecer o idioma para saber como deve ser falado!

Bons estudos! Não deixe de procurar o seu professor em caso de dúvidas.

Um grande abraço! Aproveite o livro.

Bons estudos!

1

Introdução à linguagem de programação

Introdução à linguagem de programação

Estudar uma linguagem de programação nova é como aprender um novo idioma: temos de entender seu vocabulário, sintaxe e semântica. Neste livro, vamos estudar uma linguagem de programação bastante usada mundialmente: a linguagem Java. Estudar Java será como aprender um novo idioma. Você já sabe a lógica. Lembra-se da disciplina passada? Agora é hora de aplicar aqui os conceitos de lá. E com a linguagem Java não será diferente.

A linguagem Java possui várias formas de realizar as tarefas e vamos apresentar já neste capítulo algumas das maneiras principais. Conhecer os blocos de estruturação e controle de programas é fundamental e esperamos que este capítulo o ajude nisso.

Porém é importante que você aprimore seus conhecimentos com muita prática e estudos em outras fontes de informação. Só assim você vai se tornar fluente na linguagem. Ou seja, use e abuse da internet pois existe muito material bom na grande rede e que deve ser consultado. Principalmente os sites da Oracle, empresa que mantém o Java atualmente.

Vamos lá?

Neste capítulo inicial, vamos aprender alguns assuntos fundamentais e importantes para iniciar os seus estudos. Bom trabalho!

OBJETIVOS

Ao final deste capítulo, você estará apto a:

- Usar o ambiente de programação e aplicar a estrutura da plataforma Java;
- Usar os vários tipos da linguagem nos seus programas, bem como as variáveis e constantes;
- Criar programas simples em Java.

Características da linguagem Java

A linguagem Java começou em 1991 com um projeto mantido pela Sun Microsystems, uma grande empresa da época, chamado Green. Este projeto tinha como objetivo integrar vários dispositivos eletrônicos, entre eles os computadores,

por meio de uma mesma linguagem de programação. Uma vez que os dispositivos eletrônicos eram compostos por microprocessadores, a ideia era desenvolver uma linguagem na qual um mesmo programa pudesse ser executado em diferentes dispositivos.

Com o passar do tempo, o projeto recebeu outro nome, Oak (carvalho, em inglês) e, depois, Java. O mercado de dispositivos eletrônicos não evoluiu como a Sun esperava e o projeto correu o risco de ser abortado. Porém, com o advento da Internet e a necessidade de gerar conteúdo dinâmico para as páginas web, o projeto tomou outro rumo e o Java foi lançado oficialmente em 1995, chamando atenção pelas várias facilidades que possuía para desenvolver aplicativos para a internet. Além disso, o Java possibilitava desenvolver também para aplicações em computadores desktop, bem como outros dispositivos, como os pagers e celulares.

Usos e aplicações do java: a plataforma java é composta por várias tecnologias. Cada uma delas trata de uma parte diferente de todo o ambiente de desenvolvimento e execução de software. A interação com os usuários é feita pela máquina virtual java (java virtual machine, ou jvm) e um conjunto padrão de bibliotecas de classe. Uma aplicação java pode ser usada de várias maneiras: applets, que são aplicações embutidas em páginas web, aplicativos para desktops, aplicativos para aparelhos celulares e em servidores de aplicações para internet.

O maior legado da linguagem Java é ser portável, ou seja, ter a possibilidade de se ter um código capaz de ser executado em qualquer tipo de plataforma ou sistema operacional: Linux, Windows, OS X, em aplicações desktop, web ou em dispositivos móveis. Seu slogan há alguns anos era: "*Write once, runeverywhere*" ("Escreva uma vez, rode em qualquer lugar").

Mas como que um mesmo programa vai ser executado em lugares com características diferentes?

A plataforma Java responde a essa pergunta por meio de um recurso chamado Java Virtual Machine (ou JVM). Para entender isso melhor, observe a figura 1.1 e a figura 1.2:

Figura 1.1 – Método tradicional de compilação de um programa (K19, 2016).

A figura 1.1 mostra o processo natural de um código fonte que precisa de um compilador específico para uma determinada plataforma para poder ser executado. Na prática, um programa em C++, por exemplo, quando compilado no Linux e em um processador Intel, vai rodar apenas neste ambiente e em um processador equivalente.

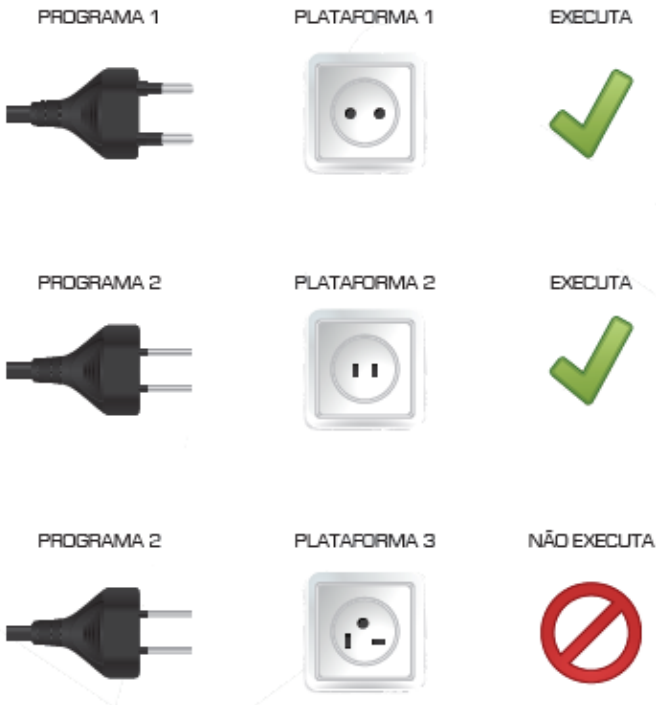


Figura 1.2 – Plataformas diferentes, executáveis diferentes (K19, 2016).

A figura 1.2 mostra 2 programas. Observe que, na primeira parte da figura, o programa 1 consegue ser executado na plataforma 1. Em seguida o exemplo mostra um programa 2 conseguindo ser executado na plataforma 2, mas não consegue executar na plataforma 3 e é neste ponto que o Java entra! A linguagem Java possui o mecanismo da JVM que cria uma camada entre o programa e a plataforma na qual ele consegue ser executado, como mostra a figura 1.3.

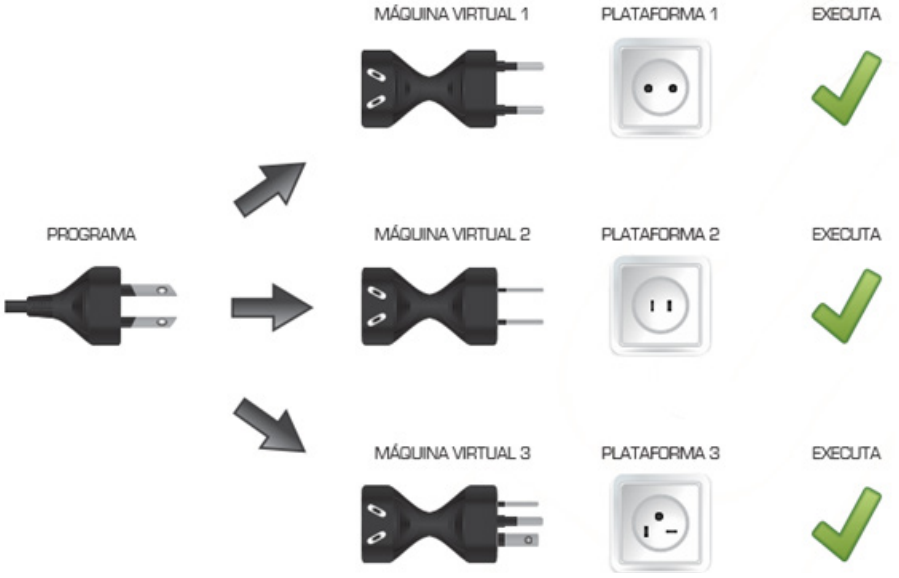


Figura 1.3 – Funcionamento da máquina virtual (K19, 2016).

Observe na figura que a máquina virtual (JVM) é específica para cada plataforma. Deste modo, um programa que foi escrito em Java, por meio da JVM, pode ser executado em várias plataformas diferentes sem ter que ser modificado. Isso é bem legal, não é?

Além dessa característica, a linguagem Java possui mecanismos para rodar em diferentes ambientes, por exemplo:

- Em aplicações desktop, como por exemplo o Eclipse, o IntelliJ, o jogo Minecraft;
- Em aplicações web por meio de vários componentes como o JSP (*Java Server Pages*);
- Em aplicações embarcadas, como, por exemplo, o *software* de vários receptores de TV a cabo existentes no mercado;
- Em aplicações para dispositivos móveis que usam o Java ME (*Micro Edition*);

• Em aplicações para o sistema Android: a linguagem Java é a principal linguagem para desenvolvimento nesta plataforma;

• Em aplicações na forma de *applets*, que são aplicativos que rodam embutidos no navegador, como foi o caso de vários teclados virtuais em ambientes de internet banking de vários bancos brasileiros.

A linguagem Java é orientada a objetos. Vamos estudar este paradigma nos próximos capítulos.

Uma característica importante da linguagem é com relação à segurança: o programa é verificado antes de ser executado. Esta característica é encontrada por exemplo nos *applets* executados nos navegadores. Devido às questões de segurança, o Java previne que estes aplicativos causem danos no computador no qual está sendo executado.

A linguagem Java também possui elementos para programação concorrente, ou seja, programas que necessitam de execução em paralelo.

Existem outras características mais técnicas, porém as que foram citadas são as mais importantes.

Vamos estudar um pouco onde podemos criar nossos programas.

A plataforma Java

É comum encontrarmos alguma referência na Internet ou literatura chamando Java de linguagem ou plataforma. Existe uma pequena diferença: a plataforma Java é um ambiente no qual é possível, por meio de várias linguagens, desenvolver programas. Entre as linguagens, existe uma específica chamada Java, a qual possui o mesmo nome da plataforma (só para nos confundir). Como vimos no tópico anterior, a grande característica da plataforma Java é não estar amarrada a apenas um sistema operacional ou hardware, porque a JVM se incumbe de criar a camada de execução.

A plataforma Java possui várias tecnologias:

• Java SE (*Standard Edition*): é a linguagem que vamos usar nas nossas aulas e a mais usada pela comunidade;

• Java EE (*Enterprise Edition*): é a linguagem usada principalmente para aplicações empresariais e para internet;

• Java ME (*Micro Edition*): usada em dispositivos móveis e softwares embarcados (por exemplo em telefones celulares);

• Java Card: usada em dispositivos móveis com limitações de recursos, por exemplo em *smartcards;*

• Java FX: usada em aplicações multimídia para *desktop* ou web.

Para executar uma aplicação em Java sabemos que precisamos da Java Virtual Machine (JVM). A JVM fica dentro de um recurso chamado Java Runtime Environment (JRE). Vamos dar um exemplo para explicar isso:

Imagine que você acabou de instalar o Windows no seu computador. O sistema está "limpo" e pronto para que alguns recursos essenciais sejam instalados, como, por exemplo, um leitor de PDF, um (des)compactador de arquivos e também o "Java". O usuário às vezes fala: "Preciso instalar o 'java' no meu computador".

Dependendo do tipo de usuário, ele vai instalar versões diferentes do que a plataforma oferece. Se for um usuário comum, o qual não é desenvolvedor e precisa de uma máquina apenas para trabalho de escritório, é necessário instalar o JRE nesta máquina. O JRE virá com a JVM e, assim, o usuário poderá usar os aplicativos em Java que quiser. Isso será o suficiente.

Se o usuário for um programador igual nós, aí a instalação necessária é a do JDK (Java Development Kit). O JDK já vem com o JRE embutido e, neste caso, dependendo do tipo de aplicativo que o programador irá trabalhar, deverá escolher entre o JDK SE, JDK EE ou JDK ME.

No nosso caso, vamos usar o JDK SE.

Ambiente de programação

Basicamente, é possível desenvolver em Java usando apenas dois componentes principais: um editor de texto comum e o JDK.

Porém existem vários ambientes de desenvolvimento para Java. Entre eles não podemos deixar de mencionar o Netbeans e o Eclipse, ambos muito usados pela comunidade Java por possuírem código aberto e serem gratuitos. São produtos livres, sem restrições à sua forma de utilização e foram por muito tempo as principais referências em ambientes de desenvolvimento para Java.

Outros programas merecem destaque e não podem deixar de ser mencionados. Entre eles, temos:

• IntelliJ da empresa JetBrains: Este programa é um IDE bem completo e muito usado. Ele possui várias ferramentas que auxiliam o programador de várias maneiras, como controle de versão depurador, criação de janelas gráficas etc. Vale a pena dar uma olhada nele quando você estiver navegando na internet;

• Sublime Text: Este na verdade é um editor de texto muito versátil e que serve para vários tipos de linguagens. Ele não é um ambiente integrado de desenvolvimento como o Netbeans ou Eclipse, porém, por meio de vários e plug-ins muito úteis, ele torna o Sublime Text uma alternativa excelente aos grandes IDEs do mercado. Além de ser extremamente rápido na sua execução;

• JDeveloper: o JDeveloper foi desenvolvido pela Oracle antes de esta comprar a Sun em 2009 (por mais de US$7 bilhões!). Para quem vai trabalhar com Java e banco de dados Oracle exclusivamente, JDeveloper é uma boa opção, pois possui vários elementos específicos da Oracle para o desenvolvimento, principalmente se o programador vai usar o framework da própria Oracle chamado ADF (Oracle Application Development Framework).

Nas nossas aulas, vamos usar o Netbeans como ferramenta "oficial".

O Netbeans é usado tanto para desenvolvimento de aplicações simples até grandes aplicações empresariais e, assim como o Eclipse, suporta outras linguagens de programação como C, C++, PHP etc. O Netbeansé um ambiente de desenvolvimento, ou seja, uma ferramenta para programadores o qual permite escrever, compilar, depurar e instalar programas. O IDE é completamente escrito em Java, mas pode suportar qualquer linguagem de programação. Existe também um grande número de módulos para estender as funcionalidades do NetBeans.

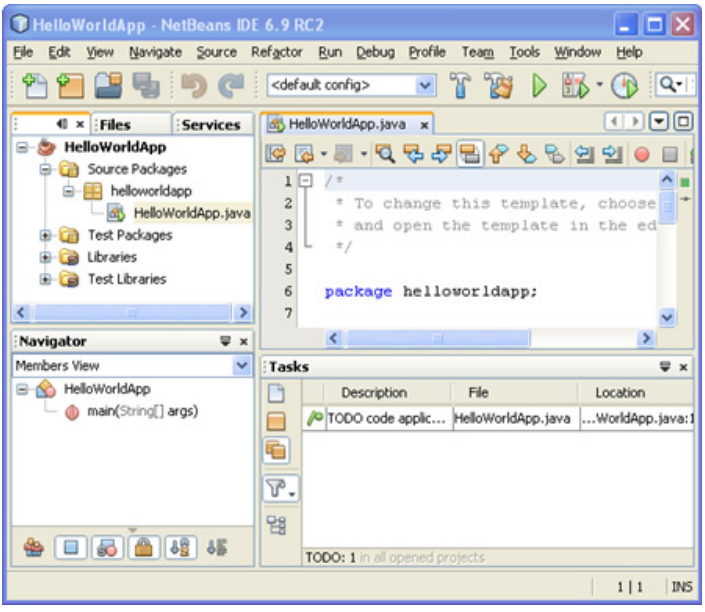


Figura 1.4 – Tela típica do *Netbeans*. Fonte: <www.netbeans.org>.

Com o Netbeans podemos desenvolver vários tipos de programas em Java, como, por exemplo:

- Aplicativos em Java para console;
- Java para *desktop* com os pacotes *Swing* e *JavaFX*;
- Java para *web*, com o uso de vários *frameworks* como o *Struts, Java Server Faces* e outros, integrado a servidores de aplicações como o *JBoss, Glassfish*, *Tomcat* e bancos de dados;
- Aplicativos para dispositivos móveis;
- *Applets*;
- Aplicativos para web em geral com HTML5;
- E outros.

Vamos agora passar ao estudo da linguagem. Começaremos igual ao estudo de qualquer linguagem: pelos seus tipos primitivos, depois operadores e expressões e, por fim, estruturas de controle. Vamos lá!

Tipos de dados

Quando vamos escrever um programa de computador, em qualquer linguagem, vamos ter de usar variáveis. As variáveis servem para guardar valores dos mais variados tipos: podem guardar números inteiros ou decimais, valores alfanuméricos, somente alfabéticos, valores lógicos (como verdadeiro ou falso) e muitos outros. Existem linguagens que permitem que o programador crie seus próprios tipos.

Vamos usar um algoritmo bem simples para ilustrar:

```
algoritmo Soma
numero1, numero2, soma: inteiro
inicio
     soma = 0
     escreval(“Digite o primeiro número inteiro:”)
     leia(numero1)
     escreval(“Digite o segundo número inteiro:”)
     leia(numero2)
     soma = numero1 + numero2
     escreval(“A soma é”,soma)
fim
```

Como podemos ver no algoritmo de exemplo, é necessário criar 3 variáveis para guardar os dados que vamos ler do usuário e para armazenar o valor da soma dos dados lidos. Estas variáveis são declaradas antes de serem usadas e o seu tipo não é alterado durante a execução do programa. Em algumas linguagens é obrigatório declarar o tipo e a variável antes de serem usadas pelo programa. Em outras linguagens, isso não é obrigatório.

Linguagem fortemente tipada: é aquela que a declaração do tipo da variável é obrigatória. Exemplo: java, c, c++, ect..
Linguagem fracamente tipada: é aquela que pode alterar o tipo durante a execução do programa. Exemplo: php, python, ruby, javascript.
Linguagens não tipada: é aquela em que só existe um tipo genérico para todo o programa ou nem existe. Exemplo: perl
Linguagem de tipo estático: neste tipo de linguagem, o compilador deve conhecer o tipo antes da execução do programa. Exemplo: java, c, c++, etc.
Linguagem de tipo dinâmico: o tipo da variável é conhecido somente na execução do programa. Exemplo: php, ruby, python

Como podemos ver no Box Explicativo, a linguagem Java é fortemente tipada e possui tipos estáticos, ou seja, antes de usar qualquer variável será obrigatório declarar a variável e seu tipo.

A linguagem Java, assim como outras linguagens, possui um conjunto de tipos necessários para as construções básicas da linguagem. Este conjunto é chamado de tipos primitivos e na prática são implementados por palavras-chave. Cada tipo primitivo possui um tamanho de memória (em bits) que é usado para armazenar o seu valor. Além disso, ele possui uma escala de valores, ou seja, possui um conjunto de valores específicos. Veja a tabela 1.1 para conhecer os tipos primitivos da linguagem Java e demais informações.

TIPO	TAMANHO (BITS)	ESCALA DE VALORES
BOOLEAN	-	true ou false
CHAR	16	'\u0000' a '\uFFFF' (0 a 65535 – caracteres Unicode ISO)
BYTE	8	-128 a +127

TIPO	TAMANHO (BITS)	ESCALA DE VALORES
SHORT	16	-32768 a 32767
INT	32	-2.147.483.648 e 2.147.483.647
LONG	64	9223372036854775808 a 9223372036854775807
FLOAT	32	1,40129846432481707e-45 a 3,40282346638528860e+38
DOUBLE	64	4,94065645841246544e-324d a 1,79769313486231570e+308d

Tabela 1.1 – tipos primitivos da linguagem Java.

Vamos explicar um pouco sobre a aplicação destes tipos:

- **boolean:** O boolean é fácil, ele só pode armazenar dois valores: true ou false. Java só aceita estes dois valores. Outras linguagens aceitam 0 ou 1, V ou F, mas Java não;
- **char:** este é tranquilo também. O char guarda qualquer caractere Unicode;
- **byte:** o byte aceita números compreendidos entre -127 e 128. É possível usar um modificador para usarmos somente números positivos também;
- **short:** o propósito do short é o mesmo do byte, porém ele guarda o dobro de valores do byte;
- **int:** este tipo guarda somente valores inteiros compreendidos no intervalo mostrado na tabela 1.1;
- **float e double:** estes dois tipos fazem a mesma coisa: guardam números decimais (também chamados de ponto flutuante). A diferença é que o double, naturalmente, guarda valores maiores que o float. Eles são usados em programas que requerem grande precisão de contas e resultados.

Quando declaramos uma variável e não atribuímos nenhum valor a ela, o Java atribui um valor automaticamente a ela de acordo com alguns padrões conforme mostra a tabela 1.2.

TIPO	VALOR PADRÃO
BYTE	0
SHORT	0
INT	0
LONG	0L
FLOAT	0.0f
DOUBLE	0.0d
CHAR	'\u0000'
BOOLEAN	false

Tabela 1.2 – Valores padrões para os tipos em Java.

Literais

Um literal é uma representação de algum valor fixo que o programador literalmente estabelece. Cada tipo primitivo possui um literal correspondente e, na tabela 1.2, estão representados os literais padrões para cada tipo. Observe os exemplos a seguir. São todos exemplos de literais.

```
boolean resultado = true;
charcMaiusculo = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

E se pegássemos como exemplo o literal 100? Ele é um int ou um long? Você percebeu, na tabela 1.2, que os valores long, float e double tem letras no final do valor? Estas letras servem para distinguir o tipo do literal em algumas situações.

Portanto, se 100 é um long, podemos representar como 100L ou 100l (é preferível usar o "L" maiúsculo). Se o 100 não tiver uma letra no final, será considerado um int.

Os valores literais para byte, short, int e long podem ser criados a partir de um literal int. Valores do tipo long que excedem a capacidade de um int podem ser criados por meio de literais do tipo long. Os literais do tipo int podem ser decimais, binários, octais ou hexadecimais. O binário e o hexadecimal são sistemas de numeração usados em programas que lidam e manipulam endereços de memória, por exemplo. Veja os exemplos a seguir:

```
// O número 26 em decimal
int decimal = 26;
// Em hexadecimal
int hexa = 0x1a;
// A declaração octal começa com 0
int octal  =  032
// E em binário
intbinario = 0b11010;
```

Como podemos perceber, a linguagem Java usa alguns caracteres para denotar os números hexadecimais, octais e binários. Isso se deve porque um dos pressupostos que a linguagem Java deveria ter é que ela fosse parecida com a linguagem C. Esta, por sua vez, é parecida com outra linguagem chamada B, que, por sua vez, é derivada da linguagem BCPL. Como a linguagem BCPL definiu esta notação, ela persistiu ao longo do tempo até chegar na linguagem Java.

Comentários em java

Quando você quiser comentar um trecho ou uma linha em java existe alguns caracteres para isso. Você pode usar o `//` para comentar uma linha inteira, ou usar o `/* */` para comentar o que estiver entre eles.

Por exemplo:

```
/* Começa o comentário aqui
…
E vai até aqui */
// Uma linha de comentário
Int tamanho;
```

Literais de ponto flutuante

Um literal de ponto flutuante é do tipo float se terminar com a letra “F” ou “f”, caso contrário, seu tipo será double e pode terminar com “D” ou “d” opcionalmente.

Os tipos de ponto flutuante (float e double) também podem ser expressos na notação científica usando a letra "E" ou "e", "F" ou "f" para um literal float e "D" ou "d" para um literal double. Observe os exemplos:

```
double d1 = 123.4;
// é o mesmo valor que d1 mas em notação científica
double d2 = 1.234e2;
float f1  = 123.4f;
```

Literais caracteres e Strings

Os literais dos tipos char e String podem conter qualquer caractere Unicode. Dependendo do editor de texto que você estiver usando para programar, você pode digitar diretamente caracteres como "Ç", "ã" e outros ou usar uma sequência chamada "escape para Unicode" para poder gerá-los. Esta sequência são os caracteres \u. Veja os exemplos:

```
//literal char
char c = 'a';  //observe que o literal char tem que estar entre aspas simples!

char n = '\u004E';   //corresponde à letra E
```

As strings em Java na verdade são instâncias de uma classe Java chamada String. Existem várias maneiras de se representar uma string em Java e todas elas usam aspas duplas para a string desejada.

Existe também o literal nulo, representado pela palavra-chave null. O literal nulo representa ausência de valor e dado. O literal nulo pode ser atribuído a qualquer variável exceto variáveis de tipos primitivos.

```
//exemplo de literal String
//observe que a classe String começa com S maiúsculo!

String s = "Exemplo de uma string";
```

Constantes e variáveis

Vimos os tipos primitivos da linguagem Java e percebemos que a linguagem obrigatoriamente precisa que o programador declare a variável antes de usá-la. Dentro de um bloco, a sintaxe para a declaração de variáveis em Java é;

```
tipoDaVariável nomeVariável;

exemplo:

int tamanho; // declaramos uma variável do tipo int chamada
// tamanho

int tamanho=40;  //neste caso, declaramos a variável tamanho do
//tipo int (inteiro) e definimos que seu valor              //inicial é 40

Outras formas:
float soma = 1.5+2.5;
float outraSoma = soma;
```

Já sabemos que o nome "variável" não é à toa, não é? Significa que a variável tamanho definida no exemplo pode ter seu valor redefinido durante a execução do programa.

Mas e quando existir variáveis que não mudam o seu valor? São as constantes, certo? Aprendemos sobre elas na outra disciplina. Em Java, a sintaxe para a criação de constantes é a seguinte:

finaltipoVariávelnomeVariável = valor;

```
Exemplo:

finalfloat PI = 3.14159265f;
finalString MES1 = "Janeiro";
```

Por convenção, todas as constantes que definirmos em Java, e nas linguagens orientadas a objeto em geral, terão seus nomes escritos em letras maiúsculas.

Operadores e expressões

Assim como em qualquer linguagem de programação, uma expressão é uma unidade de código de programa que resulta em um valor por meio do uso de operadores e operandos.

Normalmente podemos classificar os operadores em três grupos:

- Aritméticos: quando ocorre algum cálculo matemático;
- Relacionais: quando envolvem condições e comparações;
- Lógicos: quando lidam com valores booleanos e quando queremos criar condições mais complexas.

Operadores Aritméticos

Em Java, os operadores aritméticos estão mostrados na Tabela 3.

OPERAÇÃO	OPERADOR	EXEMPLO
Adição	+	2+2, a+b, a+2
Subtração	-	4-3, a-b, a-3
Divisão	/	8/5, a/b, b/4
Multiplicação	*	8*7, 8*a, a*b
Resto	%	8%4, a%b, a%2

Tabela 1.3 – Operadores aritméticos em Java.

Quando existir expressões com mais de um operador, pode-se usar parênteses, os quais têm prioridade maior do que a dos operadores. Quando houver parênteses aninhados será realizada primeiro a expressão contida nos parênteses mais internos.

Caso exista mais de um operador e/ou grupo de parênteses em uma expressão, estes serão avaliados no sentido da esquerda para a direita. No caso de parênteses aninhados, os parênteses mais internos são realizados em primeiro lugar. Veja o exemplo para entender melhor:

```
(32-2)/(2*10-(4+1))
→  →
A ordem de execução é:
    I.  (32-2)= 30
    II. (4+1)= 5
    III.    2*10 = 20
    IV. III-II = 20-5 = 15
    V.  I/IV = 30/15 = 2
```

Operadores relacionais

Os operadores relacionais mais comumente usados em Java estão mostrados na tabela 1.4:

OPERAÇÃO	OPERADOR	EXEMPLO
Igualdade	==	a==b, a==2
Diferença	!=	a!=b, a!=2
Maior	>	a>b, b>2
Menor	<	a<b, b<3
Maior ou igual	>=	a>=b, b>=3
Menor ou igual	<=	a<=b, b<=3

Tabela 1.4 – operadores relacionais em Java.

Assim como nos operadores aritméticos, é possível usar parênteses para priorizar as partes desejadas em uma expressão. Além disso, existe uma precedência de operadores quando a expressão possuir vários operadores juntos:

1. Os operadores >, <, >=, <= são aplicados primeiro
2. Os operadores ==, != são aplicados em seguida.

Lembre-se de que, quando numa expressão houver vários destes operadores, assim como em qualquer linguagem de programação, a sequência de execução é da esquerda para a direita.

Operadores lógicos

Os operadores lógicos usados na linguagem Java são mostrados na tabela 1.5.

OPERAÇÃO	OPERADOR
E	&&
Ou	||
Negação	!
Ou exclusivo	^

Tabela 1.5 – Operadores lógicos em Java.

As regras de precedência dos operadores lógicos são:

1. A negação tem a maior precedência.
2. Depois temos o "E" (&&).
3. Depois o "Ou exclusivo" (^).
4. E finalmente o "Ou" (||).

Curiosidade: além dos operadores && e || existem os operadores "&" e "|". Os operadores "&&" e "||" são mais rápidos que o "&" e "|" pois quando percebem que a resposta não mudará mais, eles param de verificar as outras condições. Por isso, eles são chamados de operadores de curto circuito (short circuit operators).

Conhecemos as características da linguagem, seus tipos, a forma de criar variáveis e constantes, como criar expressões e usar operadores; agora é hora de saber como a linguagem realmente funciona. Vamos estudar seus principais comandos de controle de fluxo.

Comandos de controle de fluxo

Você já estudou lógica de programação e aprendeu que, para construir programas, basicamente temos três principais formas de estruturas de fluxo:

1. Os comandos sequenciais
2. As estruturas de decisão
3. As estruturas de repetição

Vamos aprender como essas estruturas são feitas em Java.

Estruturas de decisão

As estruturas de decisão também são chamadas de estruturas de seleção ou condicionais e são executadas caso uma condição especificada seja verdadeira. Assim como outras linguagens, é possível construir algumas combinações deste tipo de estrutura.

Para poder mostrar o funcionamento dos comandos em Java vamos usar o diagrama de atividades da UML. Desta forma, você pode entender visualmente como o comando é executado e compará-lo com o código em Java correspondente.

Vamos lá:

IF-THEN

A figura 1.5 mostra o diagrama de atividade de uma instrução de seleção básica e o código correspondente em Java.

Por enquanto, adotaremos que todas as saídas para a tela dos nossos programas serão exibidas no console de comandos do sistema operacional. O comando para gerar uma saída no console em Java é:

```
System.out.println(String s);
//o println mostra a string s e coloca o cursor na próxima linha

System.out.print(String s);
//o print mostra a string s e deixa o cursor na mesma linha
```

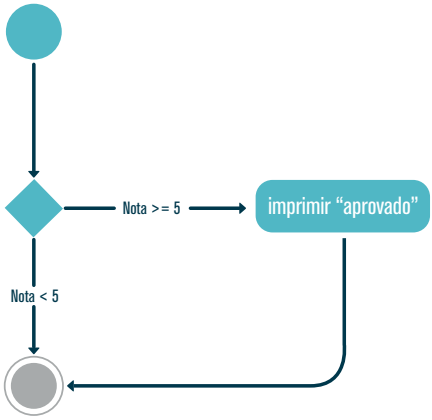


Figura 1.5 – O comando IF

Observe que, neste caso, temos apenas uma instrução que é executada se a condição do *if* for verdadeira. É claro que é possível ter várias instruções sendo executadas, porém, neste caso, teremos que abrir um bloco, como no exemplo a seguir, onde temos 2 instruções dentro de um bloco:

```
if (nota>=5) {       // início do bloco. Use o { para abrir
    System.out.println(“Aprovado!”);
    System.out.println(“Você pode agora imprimir o seu boletim.”);
}           // fim do bloco. Use o } para fechar o bloco
```

Outra observação a ser feita é que obrigatoriamente a condição do if tem de estar entre parênteses.

IF-THEN-ELSE

O IF-THEN-ELSE em Java pode ser representado na figura 1.6. Podemos observar que só foi colocada uma instrução após a condição verdadeira e uma após a condição falsa. E, assim como ocorre no IF-THEN, é possível ter um bloco após cada condição.

Uma boa prática de programação que você já deve ter aprendido anteriormente é a identação do código. Identar significa deslocar o código interno de um bloco para melhor legibilidade do código.

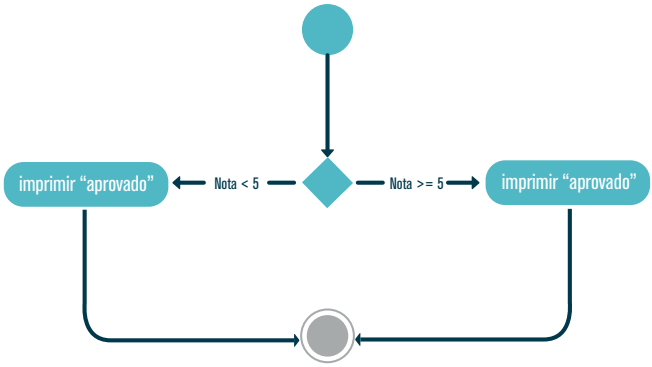


Figura 1.6 – O IF-THEN-ELSE em Java.

No Java existe um operador condicional (?:) que pode ser usado no lugar do IF-THEN-ELSE. Esse na verdade é o único operador **ternário** do Java, ou seja, ele recebe três operandos juntos.

```
System.out.println(nota >=5 ? "Aprovado" : "Reprovado");

Podemos ler a instrução assim:
"Nota é maior ou igual a 5?
Caso verdadeiro, imprima "Aprovado",
em caso contrário (após o :), imprima "Reprovado")
```

Outra possibilidade é aninhar várias instruções IF-THEN ou IF-THEN-ELSE, observe com atenção o código a seguir. O que ocorre se a nota for igual a 5? Observe que foi criado um bloco para o último else (apenas como exemplo).

```
if (nota>=9)
    System.out.println("Conceito A");
else
    if (nota>=8)
        System.out.println("Conceito B");
    else
        if (nota>=7)
            System.out.println("Conceito C");
        else
                if (nota>=6)
                System.out.println("Conceito D");
                else {
                System.out.println("Conceito E");
                System.out.println("Você está reprovado!");
                }
```

Switch

Podemos perceber que o IF-THEN é uma estrutura do tipo seleção única, ou seja, se a condição existente for verdadeira, um bloco de código é executado e a estrutura é finalizada.

No IF-THEN-ELSE a estrutura possui dupla seleção: se a condição for verdadeira, um bloco é executado ou senão o bloco correspondente à condição falsa será executado. Porém o Java possui uma estrutura na qual uma instrução de seleção múltipla pode realizar diferentes ações baseadas nos possíveis valores de uma variável ou expressão. A instrução switch possui sua estrutura geral mostrada na figura 1.7.

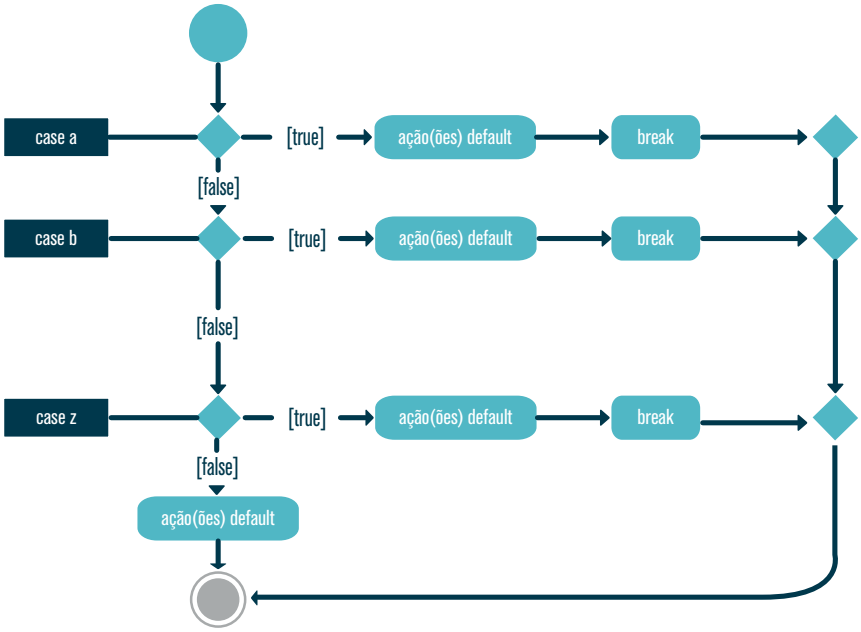


Figura 1.7 – Instrução de seleção múltipla (SWITCH) com instruções break.

O *switch* pode ser usado com os tipos byte, short, char e int, além de tipos enumerados, com a classe String e algumas classes especiais que encapsulam alguns tipos primitivos como Character, Byte, Short e Integer (veremos essas classes ao longo deste livro).

Para ilustrar esse comando, vamos usar um exemplo. No exemplo, é declarada uma variável inteira que representa o valor do mês. O programa mostra o nome do mês de acordo com o valor numérico associado.

```
1   public class Selecao {
2   public static void main(String[] args) {
3
4       int mes = 8;
5           String nomeMes;
6           switch (mes) {       //inicio do bloco
7               case 1:  nomeMes = "Janeiro";
8                        break;   //perceba o comando break no final de cada condição
9               case 2:  nomeMes = "Fevereiro";
10                       break;
11              case 3:   nomeMes = "Março";
12                       break;
13              case 4:  nomeMes = "Abril";
14                       break;
15              case 5:  nomeMes = "Maio";
16                       break;
17              case 6:  nomeMes = "Junho";
18                       break;
19              case 7:  nomeMes = "Julho";
20                       break;
21              case 8:  nomeMes = "Agosto";
22                       break;
23              case 9:  nomeMes = "Setembro";
24                       break;
25              case 10: nomeMes = "Outubro";
26                       break;
27              case 11: nomeMes = "Novembro";
28                       break;
29              case 12: nomeMes = "Dezembro";
30                       break;
31              default: nomeMes = "Mês inválido";
32                       break;
33      }                        //fim do bloco
34      System.out.println(nomeMes);
35   }
36 }
```

Neste caso, o mês “Agosto” será impresso na tela.

O corpo de um comando *switch* é chamado de bloco *switch*. Uma declaração dentro do bloco *switch* pode ser rotulada com um ou mais comandos case (ou default). O comando switch avalia sua expressão e depois executa todas as declarações que "batem" com o rótulo case correspondente. No exemplo, ele é mostrado na linha 21 e 22.

O mesmo código mostrado como exemplo pode ser convertido em vários comandos IF-THEN-ELSE:

```
int mes = 8;
if (mes == 1) {
        System.out.println(   "Janeiro");
    }
    else if (mes == 2) {
        System.out.println("Fevereiro");
    }
... // e assim por diante
```

A decisão em usar declarações IF-THEN-ELSE ou uma instrução switch é baseada na legibilidade do código e na expressão que a declaração está testando. Uma declaração IF-THEN-ELSE pode testar expressões com base em faixas de valores ou condições enquanto que um *switch* testa expressões baseadas somente em um inteiro, valor enumerado, ou *String*.

Outro ponto de observação é a instrução *break*. Cada instrução break termina a declaração de fechamento do switch. O fluxo de controle continua com a primeira declaração após o bloco switch. O break é necessário porque sem eles as declarações nos blocos *switch* seriam executadas em sequência e indistintamente até que algum break fosse encontrado.

Observe atentamente o código a seguir:

```
public class SwitchDemoFallThrough {
    public static void main(String[] args) {
        java.util.ArrayList<String>meses = new java.util.ArrayList<String>();
        int mes = 8;
       switch (mes) {
            case 1: meses.add("Janeiro");
            case 2: meses.add("Fevereiro");
            case 3: meses.add("Março");
            case 4: meses.add("Abril");
            case 5: meses.add("Maio");
```

```
            case 6:  meses.add("Junho");
            case 7:  meses.add("Julho");
            case 8:  meses.add("Agosto");
            case 9:  meses.add("Setembro");
            case 10: meses.add("Outubro");
            case 11: meses.add("Novembro");
            case 12: meses.add("Dezembro");
            break;
            default: break;
        }
        if (meses.isEmpty()) {
            System.out.println("Número de mês inválido");
        }
        else {
            for (String nomeMes :meses) {
                System.out.println(nomeMes);
            }
        }
    }
}
```

Embora existam comandos que nós não vimos ainda, dá para perceber que temos vários case sem o comando break para finalizar cada bloco. A saída na tela desse programa será:

```
Agosto
Setembro
Outubro
Novembro
Dezembro
```

Veja no exemplo que o programa busca o case que "bate" com o valor da variável mês e encontra no "case 8". O programa adiciona a palavra "Agosto" na lista de meses e assim prossegue com o "case 9", "case 10" até o "case 12", pois não há um comando break para parar a execução.

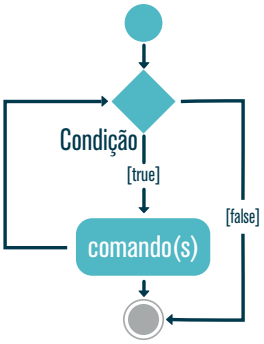
Assim, tecnicamente o break final acaba sendo desnecessário. Usar o break é recomendado para a legibilidade e deixar o código menos propenso a erros. O default trata todos os valores que não foram listados anteriormente nos case.

Estruturas de repetição

Você já aprendeu sobre as estruturas de repetição anteriormente. Estas estruturas permitem que uma ação possa ser executada várias vezes. Estas estruturas também são chamadas de laços ou loops. Existem vários tipos de repetição e todos fazem basicamente a mesma coisa: repetir uma ação várias vezes (ou não). Os tipos diferentes de repetição oferecem várias formas de determinar quando este irá começar ou terminar. Existem situações em que é mais fácil resolver um problema usando um determinado tipo de laço do que outro. Vamos ver o que a linguagem Java nos oferece:

While

O comando while executa repetidamente um bloco enquanto uma condição particular for verdadeira. A sintaxe do comando e o seu diagrama de atividade estão representados na figura 1.8.




```
while   (condição)   {
    comandos (s)
}
```

Figura 1.8 – While.

O comando *while* avalia a condição a qual sempre retorna um valor *boolean* (*true ou false*). Se a condição retorna *true*, o *while* executa o(s) comando(s) no bloco. O *while* continua testando a condição e executando o bloco até que a condição seja false.

```
class Repete {
    public static void main(String[] args){
        int conta = 1;
        while (conta < 11) {
            System.out.println("Contando: " + conta);
            conta = conta+1;
        }
    }
}
```

A linguagem Java também possui um comando DO-WHILE, o qual possui a seguinte sintaxe:

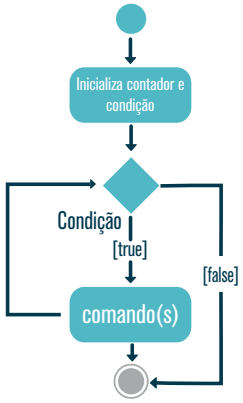
```
do {
    comando(s)
} while (condição);
```

A diferença entre o DO-WHILE e WHILE é que o DO-WHILE analisa a expressão no final da repetição ao invés do início, ou seja, pelo menos uma vez o bloco de comandos será executado.

```
class Repete2 {
    public static void main(String[] args){
        int conta = 1;
        do {
            System.out.println("Contando: " + conta);
            conta = conta+1;
        } while (conta < 11);
    }
}
```

For

O comando for é um meio compacto de fazer uma repetição sobre um intervalo de valores. A figura 1.9 mostra o diagrama de atividade e a sintaxe.




```
for (inicialização;término;incremento) {
    comando (s);
}
```

Figura 1.9 – O comando FOR.

Algumas observações devem ser feitas para o comando for:

• A expressão de inicialização começa o loop. É executada uma única vez assim que o loop começa:

• Quando a expressão de término é analisada e é falsa, o loop finaliza:

• A expressão de incremento é chamada a cada interação dentro do loop. Normalmente é feito um incremento ou um decremento.

```
class Repete3 {
    public static void main(String[] args){
        for(int i=1; i<11; i++){
            System.out.println("Contando: " + i);
        }
    }
}
```

```
    A saída do programa é:
Contando: 1
Contando: 2
Contando: 3
Contando: 4
Contando: 5
Contando: 6
Contando: 7
Contando: 8
Contando: 9
Contando: 10
```

Observe como o código declara a variável dentro da expressão de inicialização. O escopo desta variável se estende desde sua declaração até o fim do bloco do for, de modo que pode ser usada nas expressões de terminação e de incremento. Se a variável que controla um for não é necessária fora do loop, é melhor declarar a variável na expressão de inicialização. Frequentemente e tradicionalmente usamos variáveis com nomes i, j e k para controlar os loopsfor. Declarando-as na inicialização faz com que sua vida útil seja limitada, assim como seu escopo, e isso reduz erros. Veja o uso da variável i como exemplo no código anterior.

O *for* tem uma função estendida para vetores e classes *Collections*. Esse tipo de for é chamado de *enhancedfor* e pode ser usado para fazer iterações dentro dessas estruturas de dados, percorrendo-as. Por exemplo, considere o programa a seguir (trataremos sobre vetores em Java mais à frente):

```
class EnhancedFor {
    public static void main(String[] args){
            //declarando um vetor inteiro contendo uma lista de números
        int[] numeros = {1,2,3,4,5,6,7,8,9,10};
            //o for abaixo percorre o vetor e mostra na tela cada um dos números
        for (int item : numeros) {
            System.out.println("Contando: " + item);
        }
    }
}
```

No exemplo, a variável item guarda o valor atual dos números do vetor. A saída do programa está mostrada a seguir:

```
Contando: 1
Contando: 2
Contando: 3
Contando: 4
Contando: 5
Contando: 6
Contando: 7
Contando: 8
Contando: 9
Contando: 10
```

Essa forma de usar o *for* deve ser usada sempre que possível, ok? Isso se deve ao fato de que o for extendido (ou *enhanced* for) é mais eficiente para iterações com conjuntos de dados como os arrays. Portanto, sempre que for manipular um array no qual seja necessário sua travessia, use este tipo de for.

Entrada e Saída de dados via console e com JOptionPane

Vimos nos exemplos anteriores a necessidade de exibir informações na tela. Existem várias formas de se fazer isso, e a maneira mais simples é exibir as informações em uma tela de texto simples, chamada console.

Em Java, a maneira mais comum é usar o método println() ou print() da classe System.out. Já fizemos isso várias vezes nos exemplos anteriores. No último exemplo, do enhanced for, usamos o println(). A diferença principal entre o println() e o print() é que o "ln" significa line. Quando usamos o "ln", após a exibição, o cursor é colocado na linha posterior, ou seja, pula para a próxima linha.

Certo, fizemos a exibição dos dados. Mas e como ler os dados do teclado pelo console? Vamos ver o exemplo a seguir:

```
Scanner sc = new Scanner(System.in);
System.out.println("Qual e' o seu nome?");
String valor = sc.nextLine();
System.out.println("Oi "+valor+", tudo bem?");

Veja a execução deste programa:
Qual e' o seu nome?
Fabiano
Oi Fabiano, tudo bem?
```

Como podemos perceber, é possível obter valores do teclado por meio da classe Scanner. Esta classe está dentro do pacote java.util.Scanner, logo, para usá-la é necessário importar este pacote obrigatoriamente. O objeto System.in é o responsável por obter os dados do teclado. Observe que criamos a variável valor para receber o que foi lido do teclado. Como vamos ler uma String, definimos a variável valor como String. Verifique abaixo outras formas possíveis de ler entradas de dados pelo teclado com a classe Scanner. Perceba que só conseguimos ler valores relacionados com os tipos primitivos da linguagem (exceto com a classe String).

```
float num = sc.nextFloat();
intnum = sc.nextInt();
bytebaite = sc.nextByte();
longlg = sc.nextLong();
booleanbool = sc.nextBoolean();
double num = sc.nextDouble();
Stringstr = sc.nextLine();
```

Portanto, fazer a entrada de dados em Java pelo Console é simples e prático com a classe Scanner e o objeto System.in.

Também é possível realizar entrada e saída em Java com diálogos da biblioteca Swing. Esta biblioteca possui uma classe chamada *JOptionPaneque* e é também muito utilizada para a entrada e saída de dados de usuários. Mostraremos o seu uso com um programa:

```
importjavax.swing.JOptionPane;

1   public classAdicao {
2       public static void main(String[] args) {
3           String num1, num2;
4           int n1, n2, soma;
5
6           num1 = JOptionPane.showInputDialog("Digite o primeiro número");
7           n1 = Integer.parseInt(num1);
8
9           num2 = JOptionPane.showInputDialog("Digite o segundo número");
10          n2 = Integer.parseInt(num2);
11
12          soma = n1+n2;
13
14          JOptionPane.showMessageDialog(null, "A soma eh:"+soma);
15      }
16 }
```

A figura 1.10 mostra o resultado do programa Adicao.java. Perceba no programa que, para usar as classes *Swing*, é obrigatória a importação do pacote *javax.swing.JOptionPane*. O método *showInputDialog()* serve para pedir uma entrada do usuário. Esta entrada é sempre uma String e, como vamos somar as duas entradas (e fazer uma conta aritmética), é preciso converter a entrada para int. Vamos estudar sobre a conversão de tipos no próximo tópico.

O método s*howMessageDialog()* possui vários construtores e foi usado na linha 14 a forma mais simples. Observe que o segundo parâmetro do método é a *String* que desejamos mostrar na tela. O primeiro parâmetro normalmente é usado com null, porém ele se refere ao container no qual o componente está inserido. Caso o componente seja mostrado em outro container, ele deve ser especificado aqui. Mas isso não ocorre com frequência, portanto, cuidado quando for usá-lo no caso de não ser *null*.

Novamente enfatizamos o estudo da api do java e neste caso especificamente a api do swing para poder saber as possíveis formas de uso da classe joptionpane e seus inúmeros (e úteis) métodos para criar diálogos com o usuário. Não deixe de ver o link: <http://docs.Oracle.Com/javase/6/docs/api/javax/swing/package-summary.html>.

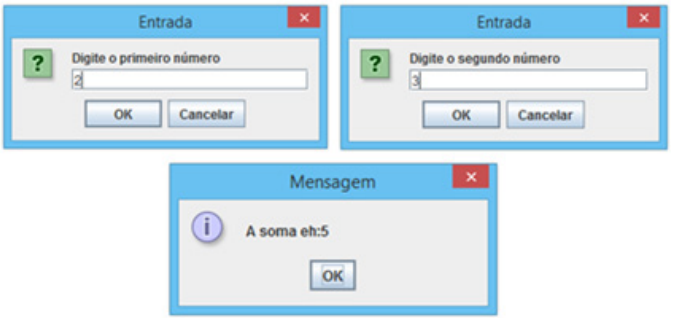


Figura 1.10 – Exemplo da classe JOptionPane.

Conversão de tipos

Qualquer linguagem de programação possui várias formas de se fazer conversões entre seus tipos suportados, e Java não é diferente. Porém, as conversões mais simples que podemos ter são as conversões entre números e strings, e vamos mostrar algumas das formas. Não se esqueça de que as outras formas você precisa investigar na API do Java para verificar outras mais apropriadas para o seu problema.

Os links a seguir mostram onde encontrar mais referências sobre a linguagem java: <http://www.Oracle.Com/technetwork/java/api-141528.html>: especificações de todas as api java <http://docs.Oracle.Com/javase/7/docs/api/>: especificação da api do java 7

Convertendo *strings* para números

É muito comum armazenar dados numéricos em objetos *string*. O CPF, por exemplo, é um caso. Todo CPF é um número com 11 dígitos e, dependendo do programa que é desenvolvido, é necessário fazer operações aritméticas com este dado. Porém dependendo do programa, ao obter o valor do CPF do usuário pelo teclado, o valor é guardado em uma string sendo necessária sua conversão posteriormente.

Para tratar deste tipo de situação, a classe Number possui subclasses que envolvem os tipos primitivos numéricos (*Byte, Integer, Double, Float, Longe Short* – perceba que todas essas são classes, veja as iniciais maiúsculas, e não os tipos primitivos). Cada uma dessas classes possui um método de classe chamado valueOf() que converte uma string em um objeto daquele tipo. Veja o seguinte exemplo:

```
1   public class TesteValueOf {
2       public static void main(String[] args) {
3           if (args.length == 2) {
4               // converte strings emnumeros
5           float a = (Float.valueOf(args[0])).floatValue();
6           float b = (Float.valueOf(args[1])).floatValue();
7
8               // algumas contas
9           System.out.println("a + b = " + (a+b));
10          System.out.println("a - b = " + (a-b));
11          System.out.println("a * b = " + (a*b));
12          System.out.println("a / b = " + (a/b));
13          System.out.println("a % b = " + (a%b));
14          } else {
15              System.out.println("Digite dois números:");
17          }
18      }
19 }
```

Este programa possui várias lições e uso de conceitos que já aprendemos.

A linha 2 possui a declaração de um vetor sem tamanho definido do tipo String chamado args. Como já estudamos na seção anterior, é possível saber o tamanho do vetor por meio da propriedade length e a linha 3 usa esta propriedade para verificar se o tamanho do vetor é igual a 2 elementos. Em caso positivo, o resto do programa é executado. Lembra-se do if?

O método main de um programa em Java aceita como parâmetros os valores digitados na linha de comando. Por exemplo: se um programa Java que converte valores de graus Celsius para Fahrenheit chama converte.java, é possível estruturar o programa para ele executar pela linha de comando da seguinte forma:

```
C:\>java converte 100

// Neste caso, o programa vai converter 100 graus Celsius para Fahrenheit. A saí-
da seria:

212.0

C:\>
```

Como podemos ver, o 100 na linha de comando será o primeiro valor do índice 0 do vetor args[] usado no programa main, ou seja, args[0]=100. Se houvesse outros valores separados por espaço, eles seriam atribuídos aos próximos índices do vetor args[].

As linhas 5 e 6 são semelhantes. Vamos analisar a linha 5 e você, posteriormente, analise a linha 6.

```
5   float a = (Float.valueOf(args[0])).floatValue();
```

A variável a vai receber o resultado da execução de 2 métodos: o valueOf() em primeiro lugar, pois está dentro dos parênteses, e depois o resultado será passado para o método floatValue(). Como exemplo, vamos supor que o usuário executou o programa com os seguintes valores: 4.5 e 87.2, portanto args[0]=4.5 e args[1]=87.2.

Na linha 5, o método valueOf() obtém a string "4.5" e a transforma para o objeto Float. Em seguida o método floatValue() transforma o valor do objeto Float para o float 4.5. Isto ocorre também na linha 6 para o args[1].

Com as variáveis a e b contendo seus valores convertidos para float, seguem algumas operações aritméticas comuns nas linhas seguintes.

Uma observação a ser feita é que as subclasses da classe Number que implementam tipos numéricos primitivos também possuem um método chamado parse____(), por exemplo Integer.parseInt(), Double.parseDouble(), que podem ser usados para converter strings para os números primitivos. Desde que um tipo

primitivo é retornado ao invés de um objeto, o parseFloat() é mais direto que o valueOf(). A linha 5 (e 6) poderia ser escrita de uma maneira mais direta assim:

```
float a = Float.parseFloat(args[0]);
```

Convertendo números para strings

Também é frequente converter um número para uma string. Há várias formas de se fazer isso e vamos mostrar uma delas.

```
int i;
// Concatena "i" com uma string vazia; a conversão é feita “na mão”, usando o “+”
String s1 = "" + i;

ou

// O método de classe valueOf().
String s2 = String.valueOf(i);
```

Cada uma das classes Number possui um método toString() para converter um número em uma string. Veja o exemplo a seguir:

```
public class TestaToString {

    public static void main (String[] args) {
        double d = 858.48;
        String s = Double.toString(d);

        int ponto = s.indexOf('.');

        System.out.println(ponto + " dígitos " + "antes do ponto decimal.");
        System.out.println((s.length()-ponto-1)+"   dígitos   depois   do   pon-
to decimal.");
    }
}
```

```
A saída do programa é:

3 dígitos antes do ponto decimal.
2 dígitos depois do ponto decimal.
```

ATIVIDADES

01. Faça um programa em Java que verifique se os clientes de uma loja excederam o limite do cartão de crédito. Para cada cliente, temos os seguintes dados:

– número da conta corrente
– saldo no início do mês
– total de todos os itens comprados no cartão
– total de créditos aplicados ao cliente no mês
– limite de crédito autorizado

Todos esses dados são inteiros. O programa deve mostrar o novo saldo de acordo com a seguinte fórmula (saldo inicial + despesas – créditos) e determinar se o novo saldo excede o limite de crédito. Para aqueles clientes cujo novo saldo excedeu, o programa deve mostrar a frase: "Limite de crédito excedido".

02. Considere o seguinte fragmento de código:

```
if (umNumero >= 0)
    if (umNumero == 0)
        System.out.println("Primeira string");
    else
        System.out.println("Segunda string");
    System.out.println("Terceira string");
```

a) O que você acha que será impresso se *umNumero* = 3?
b) Escreva um programa de teste contendo o código acima; assuma que umNumero = 3. Qual a saída do programa? Foi o que você respondeu na questão a? Explique a saída; em outras palavras, qual é o fluxo de controle do fragmento do código?
c) Usando somente espaços e quebras de linha, reformate o fragmento para torná-lo mais legível.
d) Use parênteses, colchetes, chaves e o que for necessário para deixar o código mais claro.

REFLEXÃO

O desenvolvimento em Java requer um esforço além dos livros didáticos. Aliás, qualquer linguagem é assim. Cabe ao estudante se esforçar em conhecer outros recursos da linguagem os quais não são explorados nos livros. Estudar a documentação, saber navegar dentro

da documentação, saber pesquisar dentro dela é fundamental para o bom aprendizado de qualquer linguagem. Nunca deixe de explorar a API, pois é ali que você vai encontrar a documentação mais completa do recurso que você está precisando.

REFERÊNCIAS BIBLIOGRÁFICAS

CORNELL, G.; HORSTMANN, C. **Core Java 2**: Recursos Avançados. São Paulo: Makron Books, 2001.

CORNELL, G.; HORSTMANN, C. S. **Core Java - Vol. 1 - Fundamentos**. 8. ed. São Paulo: Pearson Education, 2010.

DEITEL, H. M.; DEITEL, P. J. **Java**: como programar. 8. ed. Rio de Janeiro: Pearson, 2010.

DEVMEDIA. **Conhecendo o Eclipse** - Uma apresentação detalhada da IDE. Disponível em: <http://www.devmedia.com.br/conhecendo-o-eclipse-uma-apresentacao-detalhada-da-ide/25589>. Acesso em: 27 jul. 2016.

ECKEL, B. **Thinking in Java**. 4. ed. New Jersey: Prentice Hall, 2006.

FLANAGAN, D. **Java**: O guia essencial. 5. ed. Rio de Janeiro: Bookman, 2006.

HUBBARD, J. R. **Programação com Java**. 2. ed. Rio de Janeiro: Bookman, 2006.

K19. Apostilas gratuitas. **K19 Online**, 2016. Disponível em: <http://online.k19.com.br/libraries/handouts>. Acesso em: 01 ago. 2016.

SANTOS, F. G. D. **Linguagem de programação**. Rio de Janeiro: SESE, 2015.

SIERRA, K.; BATES, B. **Use a cabeça:** Java. 2ª. ed. Rio de Janeiro: Alta Books, 2007.

ZEROTURNAROUND. **Java Tools and Technologies Landscape for 2014**, 2014. Disponivel em: <https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/6/>. Acesso em: 27 jul. 2016.

2

Conceitos de orientação a objetos

Conceitos de orientação a objetos

Certamente você já deve ter lido ou ouvido o termo "orientado a objetos". Quem estuda ou lê bastante sobre desenvolvimento já esbarrou neste termo. E de fato: é um dos temas fundamentais para o estudo de qualquer profissional que trabalha com desenvolvimento.

A maioria das linguagens modernas é orientada a objetos: Java, C#, Python, PHP e outras mais antigas também, como Smalltalk, C++, Object Pascal etc.

Estudar orientação a objetos é superimportante para você que está começando na carreira de desenvolvimento. Os conceitos aprendidos podem ser aplicados em qualquer linguagem da "família" orientada a objetos, portanto, preste bastante atenção neste capítulo.

A linguagem Java sem dúvida é uma grande plataforma para este aprendizado. Além deste livro, você vai encontrar milhares de referências na internet e em livros sobre o assunto. E o legal é que, devido à grande portabilidade da linguagem, você poderá aproveitar os conceitos em desenvolvimento para dispositivos móveis, para a web etc.

Neste capítulo, vamos estudar os principais conceitos sobre orientação a objetos e abrir o caminho para novos conceitos e permitir que você siga por conta própria. Tenho certeza de que você vai curtir. Divirta-se e bom trabalho!

OBJETIVOS

Ao final deste capítulo:

- Você estará apto a desenvolver programas em Java usando classes e objetos.

Introdução

É claro que a maioria dos livros e tutoriais na internet que você encontrar sobre orientação a objetos irão mostrar exemplos bem simples destas estruturas. Porém, garanto a você que, quando começar a programar em projetos reais, você sentirá que precisa programar com mais produtividade, reaproveitar código e principalmente: olhar para o mundo real e saber como representá-lo com uma linguagem de programação.

Imagine o mundo da programação na década de 1970. Creio que os problemas existentes naquela época, principalmente nas empresas, não eram tão diferentes quanto são agora. Naquela época, as empresas precisam calcular a folha de pagamento, pagar impostos, controlar estoque, matéria prima etc. Mas como fazer tudo isso usando a tecnologia existente da época em relação às linguagens de programação? Era um pouco complicado.

Os grandes estudiosos da computação começaram a criar estruturas de dados e abstrair o mundo real na forma de objetos. Perceberam que tudo tem praticamente dois elementos: propriedades e comportamento. Pense nisso: acho que tudo que você pensar terá esses dois elementos. Um lápis. Tem propriedades (cor, tamanho, marca etc.) e comportamento (escreve, aponta etc.). Um mouse. Tem marca, DPIs, tipo, tamanho e também pode ser clicado, arrastado, ter a bolinha rolada etc. Um filósofo talvez vai achar um absurdo, mas, para quem programa, tudo será um objeto!

Classes e objetos

Observe a figura a seguir. Ela mostra a evolução das bicicletas.

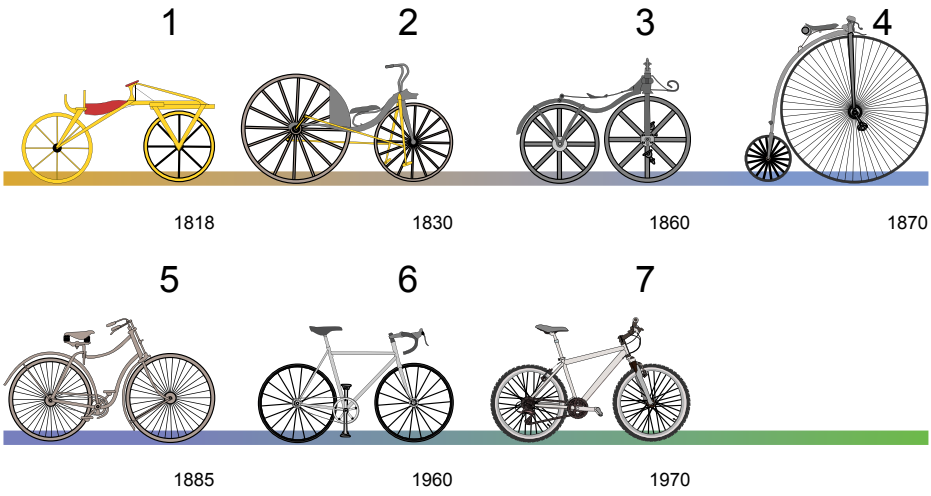


Figura 2.1 – Bicicletas.

O que elas têm em comum? Vamos tentar listar as características:

– rodas;

– tamanho;

– velocidade (se estiver parada, a velocidade é zero);
– marchas (ou não);
– modelo, marca, cor, material etc.

Elas têm comportamentos em comum:
– aceleram;
– freiam (essas duas características afetam a propriedade velocidade, certo?);
– viram à direita, viram à esquerda (logo, possuem uma característica que não listamos: a direção);
– mudam a marcha (quando possuem a característica marchas).

Logo, você percebe que toda bicicleta terá essas características. Todas, sem exceção. Se uma delas faltar, não será bicicleta. Além disso, não esqueça que temos as BMX, as de corrida de rua, as de corrida de pista, *mountain bikes*, infantis e outras.

O que estou querendo dizer é que, para ser uma bicicleta, é necessário um molde, um padrão, uma forma de fazer bicicletas. Isso será chamado de **classe**. Cada bicicleta criada a partir deste molde, deste padrão será chamado de **objeto**.

A orientação a objetos teve sua origem no laboratório da Xerox e na linguagem Smalltalk, liderada por um pesquisador chamado Alan Curtis Kay. Pelos seus estudos, ele pensou em como construir softwares a partir de agentes autônomos que interagiam entre si, originando, assim, a orientação a objetos com estas características:

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

Segundo Kay, uma classe é uma estrutura que abstrai um conjunto de objetos com características similares. Lembre-se da bicicleta: as bmx, as *mountain bikes*, as infantis e outras possuem características similares e podem ser agrupadas em uma estrutura chamada bicicleta. Tente fazer este exercício para outras "coisas".

Vamos parar de dar exemplos abstratos e vamos tratar de computação, certo? Observe a figura a seguir. Todos os exemplos da figura formam um conjunto de "coisas" similares chamadas janelas. Logo, um diálogo de arquivo, de impressão,

de entrada de dados etc. são exemplos de **objetos**. E a **janela** é o agrupamento de todos esses objetos, ou seja, uma **classe**. Na definição clássica, um **objeto** é, então, uma **instância** de uma classe.

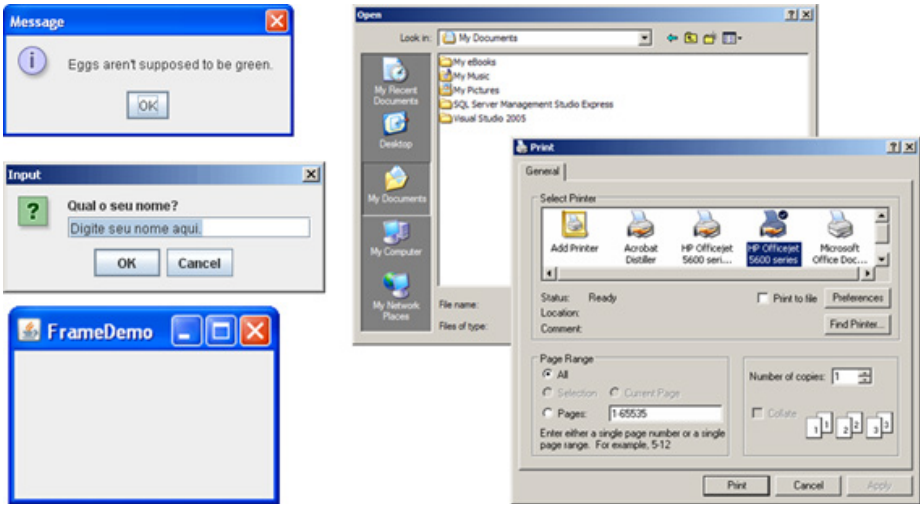


Figura 2.2 – Exemplos de janelas. Todas as janelas têm características (tamanho, cor, tipo, menus etc.) em comum, assim como seus comportamentos (abrir, fechar, maximizar, minimizar, mover etc.).

Em Java, para criar uma classe é bem fácil, observe:

```
class Bicicleta {
    ...
                    /* aqui nesta parte são inseridos as caracterís-
ticas e comportamentos da classe

}
```

Bem simples, não é? Para criar uma classe em Java, usamos a palavra-chave **class** seguida pelo nome da classe que desejamos criar. Observe que a primeira letra da classe sempre será escrita em maiúscula por questões de padronização. Lembra quando falamos do Camel Case no capítulo anterior? Não deixe de usar este padrão.

Para criar um objeto da classe Bicicleta, a sintaxe é:

```
Nome_classe nome_objeto = newNome_classe;

Bicicleta minhaBike = new Bicicleta();
```

O objeto é criado por meio do comando **new**. O exemplo anterior criou um objeto chamado "minhaBike" da classe Bicicleta. Vamos ver muitos exemplos de criação de classes e objetos neste livro, portanto estes exemplos simples são suficientes por enquanto.

Entendido sobre classes? Vamos ao próximo passo!

Atributos, métodos e construtor

A orientação a objetos tem um vocabulário próprio que precisa ser entendido, pois tudo o que você ler e estudar sobre isso terá este vocabulário.

Até agora falamos sobre propriedades e comportamento. A partir de agora, o vocabulário para estes dois termos será: atributos e métodos.

Como já deve ter percebido, o conjunto de atributos formam uma parte da estrutura de uma classe. Os atributos podem ser conhecidos também como variáveis de classe e podem ser classificados em duas formas: atributos de instância e atributos de classe. Vamos falar desses conceitos mais tarde. Por hora, voltaremos ao exemplo das bicicletas:

Uma bicicleta da marca X possui 21 marchas e é do tipo *mountain bike*. Ela possui o número de série 12345 do lote 54321 e no momento está parada (velocidade = 0), logo também possui aceleração igual a 0. A marcha atual dela é a número 5.

Viu que definimos uma bicicleta de acordo com suas características, ou melhor, seus atributos? Então, para esta bicicleta especificamente, ou seja, para este objeto, temos um conjunto de valores atribuídos a variáveis que determinam o estado do objeto naquele momento (a bicicleta está parada, por exemplo). É possível modificar estes valores? Sim. Por meio dos métodos.

Os métodos representam o comportamento dos objetos. No caso da bicicleta, podemos ter os seguintes métodos como exemplo: acelerar, parar, mudar a marcha, virar, etc. Na prática, o que aprendemos sobre funções lá na disciplina de lógica de programação serão os métodos agora.

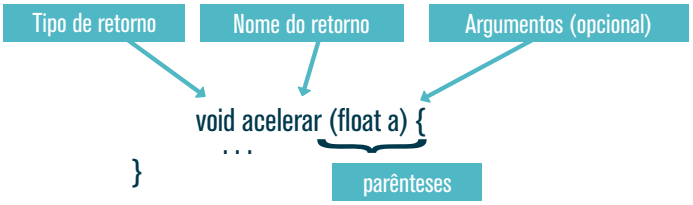
Os métodos mudam os atributos de um objeto. Por exemplo, o método parar torna a velocidade igual a 0 em uma bicicleta. O método acelerar faz com que a bicicleta aumente (ou diminua) sua velocidade, e assim por diante. O próximo conceito será importante para entender estes mecanismos.

Em Java, usamos os métodos e atributos em uma classe conforme o exemplo a seguir:

```
class Bicicleta {
            //atributos
    String numeroDeSerie;
    String lote;
    String marca;
    String tipo;
    String direcao;
    float velocidade;
    float aceleração;
    int numeroDeMarchas;
    int marchaAtual;

            //métodos
    void parar(){
            ...
    }

    void acelerar(float a){
            ...
    }

    void virar(Stringdirecao){
            ...
    }

    void mudarAMarcha(int novaMarcha){
            ...
    }
}
```

No exemplo, não colocamos o código que implementa os métodos. Ainda não é a hora. Por enquanto, só vamos mostrar como ficaria a estrutura de uma classe em Java.

Os únicos elementos requeridos na declaração de métodos são: o tipo de retorno do método, o nome, o par de parênteses e o corpo entre chaves { e }. Observe:



Em geral, a definição de um método tem os seguintes componentes:

1. Modificadores: como public, private e outros. Quando não é usado, a classe e/ou seus membros são acessíveis somente por classes do mesmo pacote. Neste caso, na sua declaração não é definido nenhum tipo de modificador, sendo este identificado pelo compilador.

2. O tipo de retorno: o tipo de dado do valor retornado pelo método ou void se não há retorno.

3. O nome do método.

4. A lista de argumentos entre parênteses: os argumentos são separados por vírgula, seguido pelo tipo de dados e pelo nome do argumento. Se não houver parâmetros, usa-se os parênteses sem parâmetros, como é o caso do método parar() na classe Bicicleta.

5. O corpo do método, entre chaves.

> Os métodos possuem uma assinatura. A assinatura é composta pelo nome do método, tipos dos argumentos em ordem. A assinatura do método anterior é:
> acelerar(float)

Assim como os nomes de classes e variáveis, os nomes dos métodos também devem obedecer à convenção usada no Java (CamelCase). Além disso, é recomendável que todo método comece por um verbo (com a primeira letra em minúscula). Veja alguns exemplos:

correr, correrRapido, getBackground, getDados, compareTo, setX, isEmpty

Alguns nomes estão em inglês porque são métodos bastante usados em vários programas e, para manter uma maior legibilidade, ficam em inglês.

Normalmente um nome de método é único dentro da classe. Entretanto, um método pode ter o mesmo nome que outros métodos. Isto é chamado de sobrecarga. Guarde esta palavra, pois vamos estudar este conceito no tópico 1.2.3.

Construtores

Quando um objeto é criado ele automaticamente recebe todos os atributos e métodos de sua classe. Como exemplo, suponha que a classe Janela seja definida assim:

```
public class Janela{
            //atributos
    int posicaoX;
    int posicaoY;
    int largura;
    int altura;
    String tipo
           // outros atributos

           //métodos
    void maximizar()      {...}
    void minimizar()      {...}
    void mover()  {...}
}
```

Toda vez que uma janela é criada no Windows por exemplo, ela receberá uma posição na tela por padrão, uma altura, uma largura, tipo etc. O construtor é um método especial que é executado quando um objeto é criado e serve principalmente para inserir valores nas variáveis de instância da classe. Tenha o hábito de sempre criar pelo menos um construtor para suas classes.

As declarações de construtores se parecem com declarações de métodos, exceto que eles têm exatamente o mesmo nome da classe e não possuem tipos de retorno. Veja novamente a classe Janela:

```
public class Janela{
          //atributos
    int posicaoX;
    int posicaoY;
    int largura;                    observe que o nome do
    int altura;                     método é exatamente igual
    String tipo                     ao nome da classe
          // outros atributos

          //constutor
    Janela(){
          posicaoX = 100;
          posicaoY = 100;
          largura = 50;
          altura = 50;
          tipo = "Formulario"
    }

          //métodos
    void maximizar()      {...}
    void minimizar()      {...}
    void mover()  {...}

}
```

No final do tópico 1.2, vimos que, para criar um novo objeto, usamos o comando **new**. Quando este comando é executado, a primeira coisa que o compilador faz após encontrar a classe é procurar o seu construtor. Se houver, ele é executado. No exemplo da classe Janela, um novo objeto é criado na posição X = 100, Y=100, com altura e largura igual a 50, do tipo "formulário". Ou seja, todo objeto que for criado desta classe terá inicialmente estes valores. E este é o objetivo do construtor. Não é uma boa prática criar objetos e deixar suas variáveis de instâncias sem serem inicializadas. Aliás, também não é uma boa prática deixar qualquer variável sem um valor inicial.

Para criar um novo objeto da classe Janela chamado janelaPrincipal usamos a seguinte linha:

```
Janela janelaPrincipal = new Janela();
```

Portanto, janelaPrincipal terá os valores definidos conforme o construtor foi criado.

É possível ter mais do que um construtor para uma classe. Este recurso também é uma sobrecarga e vamos estudá-lo no tópico 1.2.3. Você não precisa escrever um construtor para sua classe, porém isso não é recomendado. Quando não é criado um construtor, o compilador automaticamente cria internamente um construtor sem argumentos para a classe. Vamos retomar este assunto no próximo capítulo, pois, neste caso, outros conceitos estão envolvidos e não é o momento de falar deles.

Encapsulamento

O encapsulamento é um dos quatro pilares da orientação a objetos. Quando falamos sobre encapsulamento tenho certeza de que a primeira coisa que vem à sua cabeça é cápsula. E o conceito está relacionado com isso. A cápsula é um dispositivo no qual temos mais contato com sua parte externa não é? Na orientação a objetos, o encapsulamento evita que o usuário da classe saiba detalhes de como os métodos são implementados. Imagine um ciclista durante uma prova. Você acha que ele precisa saber o que ocorre com os mecanismos, catracas, engrenagens da bicicleta para que a marcha seja mudada? Não. Ele quer acionar o comando na bicicleta e ter a marcha mudada! Outro exemplo: considere uma Conta Bancária como uma classe. Quando você vai realizar uma operação (um saque, depósito) você não precisa, e nem quer saber, quais são os detalhes internos que ocorrem no banco, só quer ver seu dinheiro entrando ou saindo da conta. Logo, os processos são ocultos e encapsulados em um nome: uma Conta, uma Bicicleta etc.

Este conceito nos leva a pensar em algumas coisas que estão relacionadas diretamente com a teoria de orientação a objetos. Uma delas é que o usuário não precisa saber quais são as variáveis de instância de uma classe, basta saber quais são os métodos que ela possui e quais deles estão disponíveis para uso.

Temos então o conceito de **interface**. Uma interface lista os serviços fornecidos por um componente. No caso da Bicicleta, teríamos como serviços os

métodos parar, acelerar, virar, *mudarAMarcha.* No caso da Janela, maximizar, mover, minimizar. Ou seja, basicamente quem compõe a **interface** de uma classe são seus métodos **públicos**.

E isso nos leva a outro conceito: os modificadores de acesso.

Os **modificadores de acesso** servem para deixar os componentes de uma classe disponíveis ou não para outras partes dentro da classe ou mesmo para outras classes. Em Java temos alguns modificadores, porém, por enquanto, estudaremos o public e o private.

- O **public** faz com que o componente (atributo ou método) seja acessível para todas as outras classes.
- O **private** faz com que o componente (atributo ou método) seja acessível somente dentro da própria classe.

De acordo com o que já vimos sobre encapsulamento, o usuário da classe só poderá ter acesso aos componentes públicos, então é comum e recomendável que os campos sejam private. Isso significa que eles somente podem ser acessados diretamente por sua classe. Porém, é necessário acessar esses campos e seus valores. Isso pode ser feito indiretamente por meio de métodos públicos que obtém o valor para nós.

Vamos reescrever a classe Janela usando os modificadores. Eles estão escritos em negrito, veja:

```
public class Janela{
    //atributos
    private int posicaoX;
    private int posicaoY;
    private int largura;
    private int altura;
    private String tipo
    // outros atributos

    //constutor
    Janela(){
        posicaoX = 100;
        posicaoY = 100;
        largura = 50;
        altura = 50;
        tipo = “Formulario”
    }
    //métodos
    public void maximizar()  {...}
```

```
    public void minimizar()  {...}
    public void mover()  {...}
}

Vamos agora a um exemplo no qual mostra o objeto encapsulado:

public classExemploEncapsulamento{
public static void main(String[] args){
    // vamos criar um objeto chamado dialogo da classe Janela
    // ele terá a posição 100,100 e largura e altura = 50
    // conforme o construtor

    Janela dialogo = new Janela();

    // a linha abaixo gerará um erro pois a variável posicaoX
    // não é mais acessível, ela está oculta (encapsulada)
    dialogo.posicaoX = 10;  ← ERRO!!!
    dialogo.mover(10,10);   ← Agora sim podemos mudar as variáveis
    ...
    }
}
```

Muito bem, aparentemente não mudou muita coisa.

Sobrecarga de métodos e de construtores

Vamos agora cumprir a promessa sobre o conceito de sobrecarga ou overload. Basicamente a sobrecarga de métodos ocorre quando dois métodos possuem o mesmo nome.

A linguagem Java suporta a sobrecarga de métodos. A diferenciação de métodos com o mesmo nome é feita por meio de suas assinaturas. Isso significa que métodos dentro de uma classe podem ter o mesmo nome se eles possuírem uma lista de parâmetros diferentes. Veja o próximo trecho de código. Perceba que temos 3 métodos soma, porém com assinaturas diferentes. Eles servem para o mesmo propósito, porém seus tipos de retornos e parâmetros são diferentes.

```
public class Soma{
    public int soma(int x, int y){...}
    public int soma(String x, String y){...}
    public double soma(double x, double y){...}
}
```

O método que será chamado depende somente do tipo do parâmetro passado e essa decisão será tomada pelo compilador.

Você não pode declarar mais do que um método com o mesmo nome e o mesmo número de tipos de argumentos a fim de evitar erros de compilação. O compilador também não considera os diferentes tipos de retorno em métodos com o mesmo nome, portanto você também não poderá criar métodos com a mesma assinatura e tipos de retorno diferentes.

Ou seja, para explicar o parágrafo anterior, o exemplo a seguir gerará um erro de compilação devido às observações citadas:

```
public class Soma{
    public int soma(int x, int y)       { ... }
    public int soma(int aa, int yy)     { ... }
    public float soma(int w, int z)     { ... }
}
```

Métodos e atributos estáticos

Em Java uma variável pode ser declarada com a palavra-chave static. Quando isso ocorre, essa variável é chamada de variável de classe. Preste atenção: até agora os atributos da classe podiam ser chamados de variáveis de instância, porém, com a palavra static, ela possui outro conceito.

Quando temos uma variável de classe, todas as instâncias da classe compartilharão o mesmo valor que a variável da classe possui. Uma variável de classe pode ser acessada diretamente com a classe, sem a necessidade de se criar uma instância. Variáveis estáticas são inicializadas somente uma vez, no início da execução. As variáveis de classe são inicializadas antes, e as variáveis de instância, depois.

Melhor exemplificar, não é? Vamos lá! Vamos usar novamente o exemplo das bicicletas, mas agora você será o dono de uma loja de bicicletas. Cada bicicleta que for comprada deveria ser contada para que você pudesse saber quantas bicicletas foram compradas até o momento. Poderíamos criar um atributo "total" que será incrementado toda vez que uma bicicleta for comprada. Na prática, poderíamos fazer assim: toda vez que uma bicicleta for comprada, será criado um novo objeto no programa e podemos colocar no construtor um comando para incrementar o número total de bicicletas. Assim:

```
class Bicicleta {
        //atributos
    private String numeroDeSerie;
    private String lote;
    private String marca;
    private String tipo;
    private String direcao;
    private float velocidade;
    private floataceleracao;
    private int numeroDeMarchas;
    private int marchaAtual;
    private stat ic int total;
        //métodos
    Bicicleta (String n, String l, String m, String t, String d, float v, float a,
int nm, int ma) {
        numeroDeSerie = n;
        lote = l;
        marca = m;
        tipo = t;
        direcao = d;
        velocidade = v;
        aceleracao = a;
        numeroDeMarchas = nm;
        marchaAtual = ma;
        total = total+1;
    }

    void parar(){ ... }
    void acelerar(float a){...}
    void virar(String direcao){ ...   }
    void mudarAMarcha(int novaMarcha){...}
}

Vamos fazer alguns testes:

Bicicleta b1 = new Bicicleta(“1010”,”lote1”,”Kaloy”,”Mountain
bike”,”norte”,0,0,21,1);
Bicicleta b2 = new Bicicleta(“1011”,”lote1”,”Kaloy”,”Infantil”,”norte”,0,0,1,1);
Bicicleta b3 = new Bicicleta(“1012”,”lote2”,”Kaloy”,”Corrida”,”sul”,0,0,10,1);
Bicicleta b4 = new Bicicleta(“1013”,”lote1”,”Kaloy”,”Feminina”,”norte”,0,0,21,1);

int numeroDeBicicletas = Bicicleta.total; ←
```

Veja a última linha. Criamos uma variável no programa chamada "numeroDeBicicletas", que recebe o número de objetos criados. Veja que a forma de acessar a variável da classe "total" é direta, ou seja, usa-se o nome da classe "ponto" nome da variável.

As variáveis de classe são úteis principalmente para armazenar o número de objetos de uma classe ou para propagar uma informação a todos os objetos criados da classe.

Assim como existem os atributos estáticos, existem também os métodos estáticos. Eles são chamados de métodos da classe e não dependem de nenhuma variável de instância. Eles têm um funcionamento semelhante aos atributos estáticos, ou seja, quando o método estático é executado, ele atua em toda a classe ao contrário dos métodos não estáticos, que atuam somente nos objetos onde foram chamados. No próximo tópico, vamos encontrar vários exemplos de métodos estáticos na classe Math.

Classes predefinidas: Math e String

A linguagem Java ajuda bastante o programador com alguns recursos prontos. Na verdade, a linguagem Java já possui uma biblioteca de classes incorporada que ampliam o poder de desenvolvimento do programar. Para saber sobre essas classes, é importante que você dê uma boa navegada na API do Java.

> Para navegar no site da api do java e conhecer as classes incorporadas, acesse o link: <https://docs.Oracle.Com/javase/7/docs/api/>.

Duas classes são particularmente úteis para qualquer estudante de programação em Java: a classe Math e a classe String. Como você deve esperar, a classe Math contém inúmeros métodos relacionados a funções matemáticas, e a classe String, por sua vez, oferece vários métodos para tratamento de Strings. Ambas as classes estão presentes no pacote java.lang, logo, não precisamos importá-las para usá-las. Além disso, são classes estáticas, ou seja, também não precisamos instanciar seus objetos.

Vamos exemplificar com alguns métodos existentes nestas classes:

- **Math.PI:** constante que retorna o valor de π

Exemplo:

```
System.out.println("O valor de pi é: " + Math.PI);
```

- **Math.pow():** método que calcula potências

Exemplo:

```
System.out.println("O valor de 2 elevado ao cubo(3) é: " + Math.pow(2,3) );
```

- **Math.sqrt():** método que calcula a raiz quadrada de um número

Exemplo:

```
System.out.println("O valor da raiz quadrada de 49 é: " + Math.sqrt(49) );
```

- **Math.sin():** método que calcula o seno de um ângulo

Exemplo:

```
System.out.println("O seno de 90 é: " + Math.sin(90) );
```

Creio que você já entendeu como usar os métodos, certo? Outros métodos muito úteis dessa classe são:

Math.abs(numero): Para calcular o módulo de um número.

Math.min(n1,n2): Para calcular o valor mínimo de dois números 'num1' e 'num2'.

Math.max(n1,n2): Para calcular o valor máximo de dois números 'num1' e 'num2'.

Math.ceil(n): Para arredondar um número para cima.

Math.floor(n): Para arredondar um número para baixo.

A classe Math possui muitos métodos. Agora é com você para explorá-los. Lembre-se: consulte sempre a API!

Vamos agora ver a classe String. Ela é bem útil também e possui funcionamento semelhante à classeMath. Antes de estudar a classe com mais detalhes, vamos apenas dar uma olhada nas Strings:

Sabemos que as Strings são criadas da seguinte forma:

```
String saudacao = “HelloWorld!”;
```

Mas também podemos criá-las assim:

```
char[] saudacaoVetor = {'H','e','l','l','o',' ','W','o','r','l','d','!'};
String saudacao = new String(saudacaovetor};
```

É um pouco mais trabalhoso, mas é bastante útil também. A seguir vamos exemplificar com alguns métodos desta classe.

• **length():** este método retorna o tamanho da String em bytes. Observe o exemplo e a forma de executar o método:

```
String bla = "Blablabla";
int len = bla.length();   //usamos a string "ponto" nome do método
System.out.println("O tamanho é: " + len);

O resultado deste código será:
O tamanho é: 11
```

• **concat():** este método concatena (junta) duas strings, formando uma. Observe o exemplo e a forma de executar o método:

```
"A sacada ".concat(" da casa");

O resultado será:
"A sacada da casa"

Outra forma para obter o mesmo resultado:
String string1 = "A sacada ";
String string2 = " da casa";
    System.out.println(string1.concat(string2));
```

• **charAt():** retorna o caractere de uma determinada posição da String.

• **compareTo():** compara uma String com outra String ou com um objeto.

• **compareToIgnoreCase():** compara duas Strings ignorando se são maiúsculas ou minúsculas.

• **equals():** este método é muito usado. Ele compara a String com outro objeto. Ele é muito usado em comandos de comparação dentro de laços como while ou em condicionais.

• **replace():** este método também é bastante usado. Ele troca os caracteres de uma String de acordo com o parâmetro passado. Ele possui algumas variações como o replaceAll() e replaceFirst()

• **split():** outro método bastante usado. Ele separa uma String em duas de acordo com o parâmetro passado.

• **substring():** "super" usado também. Ele retorna uma nova String que é um pedaço da String original.

• **toLowerCase():** retorna uma String escrita somente em caracteres minúsculos. Se existe um método assim, é natural supor que existe um método que converte todas as letras para maiúsculas! E existe mesmo: o **toUpperCase()**.

Novamente incentivamos o uso e investigação da API do Java para estudar com mais detalhes as possibilidades da classe String. Existem muitos métodos bastante úteis e que podem ser muito bem empregados em várias aplicações, portanto, não tenha preguiça de navegar dentro da API.

Vetor

Já estudamos sobre vetores na disciplina de algoritmos. Um vetor (ou *array*) é uma estrutura de dados que armazena uma sequência de dados, ou objetos, todos do mesmo tipo em posições consecutivas da memória.

O tamanho de um vetor é fixado assim que o vetor é criado.

Cada item em um vetor é chamado de elemento e cada elemento é acessado por meio de um valor chamado índice. Em Java, os índices dos vetores começam com 0 (zero).

Veremos um exemplo no qual é criado um vetor de inteiros; alguns valores são adicionados e depois impressos na tela.

```
class Array1 {
    public static void main(String[] args) {   // declarando um vetor de inteiros
        int[] umVetor;           // definindo o tamanho do vetor para 10 inteiros
        umVetor = new int[10];   // definindo o primeiro elemento
        umVetor[0] = 100;        // definindo o segundo elemento
        umVetor[1] = 200;        // e assim por diante
        umVetor[2] = 300;
        umVetor[3] = 400;
        umVetor[4] = 500;
        umVetor[5] = 600;
        umVetor[6] = 700;
        umVetor[7] = 800;
        umVetor[8] = 900;
        umVetor[9] = 1000;
```

```
        System.out.println("Elemento com índice 0: " + umVetor[0]);
        System.out.println("Elemento com índice 1: " + umVetor[1]);
        System.out.println("Elemento com índice 2: " + umVetor[2]);
        System.out.println("Elemento com índice 3: " + umVetor[3]);
        System.out.println("Elemento com índice 4: " + umVetor[4]);
        System.out.println("Elemento com índice 5: " + umVetor[5]);
        System.out.println("Elemento com índice 6: " + umVetor6]);
        System.out.println("Elemento com índice 7: " + umVetor[7]);
        System.out.println("Elemento com índice 8: " + umVetor[8]);
        System.out.println("Elemento com índice 9: " + umVetor[9]);
    }
}
```

A saída do programa será:

```
Elemento com índice 0: 100
Elemento com índice 1: 200
Elemento com índice 2: 300
Elemento com índice 3: 400
Elemento com índice 4: 500
Elemento com índice 5: 600
Elemento com índice 6: 700
Elemento com índice 7: 800
Elemento com índice 8: 900
Elemento com índice 9: 1000
```

Em uma situação real você provavelmente usará as estruturas de repetição que já vimos neste capítulo para percorrer o vetor pois ficará muito mais legível e elegante do que escrever linha a linha como foi feito no exemplo.

Declarando uma variável como vetor

No programa anterior, criamos um vetor de inteiros (chamado umVetor) com 10 posições.

Assim como nas declarações para variáveis, uma declaração de vetor tem dois componentes: o nome do tipo e o nome do vetor. O tipo do vetor é escrito como tipo[], onde o tipo é o tipo dos dados do conteúdo do vetor. Os colchetes indicam que o tipo declarado conterá um vetor. O tamanho do vetor não é obrigatório na declaração. Lembre-se: declarar um vetor não significa que ele estará criado. A declaração conta ao compilador que a variável declarada conterá um vetor de um determinado tipo. A criação se dá com o comando new.

Você pode declarar vetores de outros tipos:

```
byte[] vetorBytes;
short[] vetorShorts;
long[] vetorLongs;
float[] vetorFloats;
double[] vetorDoubles;
boolean[] vetorBooleanos;
char[] vetorChars;
String[] vetorStrings;
```

Criando, acessando e manipulando um vetor

Uma forma de criar um vetor é usar o operador new, como já falamos. A linha do programa Array1.java anterior cria um vetor com 10 posições.

```
// declarando um vetor de inteiros
int[] umVetor;

// definindo o tamanho do vetor para 10 inteiros
umVetor = new int[10];

//também poderia ser feito as seguinte forma:
int[] umVetor = new int[10];
```

Se o new for esquecido ou omitido, o compilador gerará um erro dizendo que a variável umVetor não foi inicializada.

As linhas abaixo atribuem um valor a cada posição do vetor:

```
umVetor[0] = 100; // inicializa o primeiro elemento
umVetor[1] = 200; // inicializa o segundo elemento
umVetor[2] = 300; // e assim por diante
```

Cada elemento do vetor é acessado pelo seu índice numérico:

```
System.out.println("Elemento 1 no índice 0: " + umVetor[0]);
System.out.println("Elemento 2 no índice 1: " + umVetor[1]);
System.out.println("Elemento 3 no índice 2: " + umVetor[2]);
```

É possível usar a seguinte forma para criar e inicializar um vetor:

```
int[] umVetor = {100, 200, 300,400, 500, 600,700, 800, 900, 1000};

// nesta forma o tamanho do vetor é determinado pelo número de valores fornecidos
// entre as chaves e separados por vírgulas
```

Também é possível usar vetores de vetores, ou seja, vetores com mais de uma dimensão, chamados de multidimensionais, usando dois ou mais conjuntos de colchetes, como, por exemplo, String [][] valores. Cada elemento, portanto, deve ser acessado pelo valor do número do índice correspondente.

Na linguagem Java, um vetor *multidimensional* é um vetor que contém vetores como componentes. Como consequência, as linhas podem ter tamanhos variados.

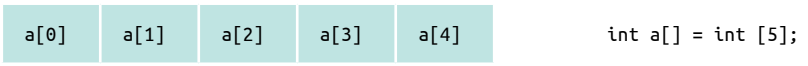


Figura 2.3 – Um vetor dimensional com 5 posições.

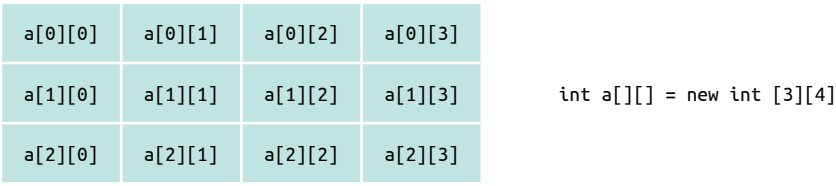


Figura 2.4 – Um vetor bidimensional com 3 linhas e 4 colunas.

Na figura 3.3 e na figura 3.4 cada elemento é identificado por uma expressão de acesso ao vetor da forma a[linha][coluna]; a é o nome do vetor e linha e coluna são os índices unicamente identificados para cada elemento do vetor, por número de linha e coluna.

Vamos usar um programa como exemplo:

```
class MultiDim {
    public static void main(String[] args) {
        String[][] nomes = {{"Sr.","Sra.","Srta. "},{"Silva","Santos"}};
        // Sr. Silva
        System.out.println(nomes[0][0] + nomes[1][0]);
        // Srta. Santos
        System.out.println(nomes[0][2] + nomes[1][1]);
    }
}
```

A saída do programa será:
Sr. Silva
Srta. Santos

Você pode usar a propriedade length para determinar o tamanho de cada vetor. Veja o exemplo:

```
System.out.println(umVetor.length); //esta linha imprime o tamanho do vetor umVetor
```

Copiando vetores

Como já dissemos, a linguagem Java possui muitas classes que auxiliam o desenvolvedor em tarefas como manipulação de vetores. Uma tarefa útil e muito usada é a cópia de vetores. A classe System possui um método arraycopy que você pode usar para fazer a cópia dos dados de um vetor para outro. O método possui a seguinte sintaxe:

```
public static void arraycopy  (Object src, intsrcPos, Object dest, intdestPos, int
length);
```

Os dois argumentos srce dest especificam o vetor de origem e destino, respectivamente. Os três argumentos int especificam a posição inicial no vetor de origem, a posição inicial no vetor de destino e o número de elementos a serem copiados. Observe o programa a seguir:

```
public class CopiaParte {
    public static void main(String[] args) {
        //(1) cria o array "a" e o preenche com os caracteres da palavra "telefone"
        char[] a = { 't','e','l','e','f','o','n','e'};
        //(2) Copia apenas a parte "ele" para o array b, usando “System.arraycopy()”
        char [] b = new char[3];
        //primeiro é preciso reservar espaço para b
        System.arraycopy(a, 1, b, 0, 3);
        //agora copiamos a palavra "ele"
        //(3) exibe o conteúdo de "b"
        for (int i=0; i<b.length; i++) {
            System.out.println("b[" + i + "]=" + b[i]);
        }
    }
}
```

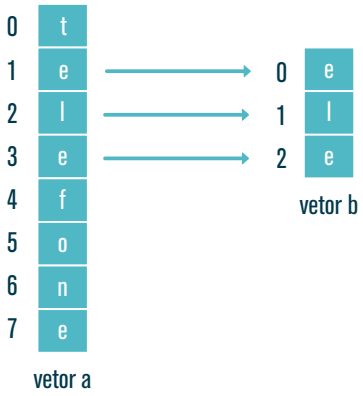


Figura 2.5 –

Existem outras abordagens de cópia de vetores: usando o loop for, o método clone() e o método copyOf() da classe Arrays. Nesta classe, existem vários métodos muito úteis para a manipulação de vetores, como, por exemplo:

- Procurar dentro de um vetor por um valor específico e obter o índice ao qual ele se refere (método binarySearch);
- Comparar dois vetores para determinar se eles são iguais ou não (método equals);
- Preencher um vetor com um determinado valor (método fill);
- Classificar um vetor em ordem crescente (método sort);
- Etc.

Relacionamento entre objetos

As coisas no mundo real se relacionam por meio de diversas formas, não é? Por exemplo:

→ Um objeto motor é parte de um objeto carro;

→ Uma turma (objeto) possui vários alunos (objetos);

→ Uma janela (objeto) possui vários botões (objetos):

Em Java, os relacionamentos são chamados de associações e podem ser de vários tipos:

→ Agregação: estabelecem um vínculo entre objetos:

→ Composição: é uma associação do tipo todo-parte;

→ Uso: um objeto usa a funcionalidade de outro sem estabelecer um vínculo duradouro (referência).

Vamos dar alguns exemplos para você entender melhor:

→ Uma conta corrente de um banco é formada por várias transações de crédito e de débito → isto é composição;

→ Um cadastro de clientes é formado por vários clientes → isto é a agregação.

→ Um cliente tem uma conta corrente → outra agregação;

→ Um documento possui um conjunto de parágrafos → isso é composição.

→ Uma turma é um conjunto de alunos → outra composição.

Os relacionamentos são melhor entendidos por meio de um diagrama chamado diagrama de classes;

> O diagrama de classes faz parte de uma linguagem de modelagem chamada uml (unified modeling language). A uml 2.2 Possui 14 tipos de diagramas divididos em duas categorias principais: estruturais e comportamentais. Vale a pena dar uma olhada no site oficial para conhecer melhor este conjunto de diagramas. Você vai usá-los muito no desenvolvimento em java. Site: <http://www.uml.org/>.

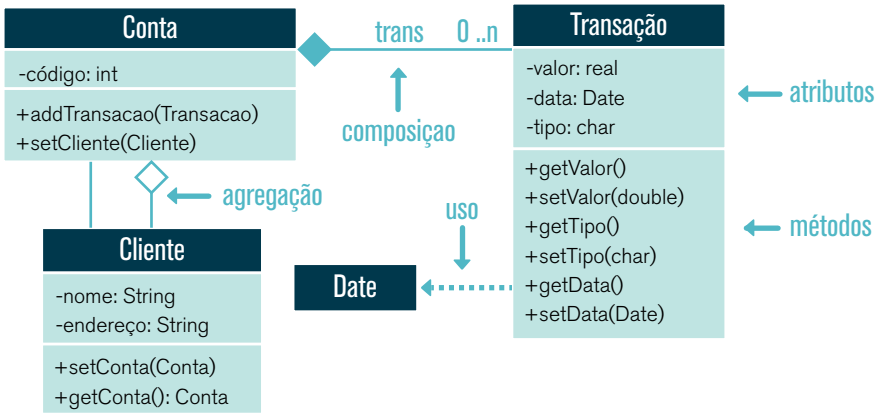


Figura 2.6 – Tipos de associação em Java (http://www.cin.ufpe.br/).

Observe a figura 1.6. As classes em UML são representadas por um retângulo dividido em 3 partes: a parte superior mostra o nome da classe, a do meio contém os atributos e a inferior mostra os métodos. O sinal "-" significa que o atributo ou método é private, o sinal "+" significa que o atributo ou método é public.

Veja na figura que, na classe Conta, terá um método (addTransacao) que recebe uma Transação como parâmetro, assim como o setCliente recebe um Cliente. Ou seja, teremos uma instância de Transação e de Cliente associada à Conta. Na

classe Cliente teremos um objeto da classe Conta como variável de instância, caracterizando a agregação. O código abaixo traduz em Java o diagrama mostrado na figura 1.6.

```
class Transacao{
    private real valor;
    private Date data;
    private char tipo;

    public real getValor(){...}
    public void setValor(real v){...}

    public Date getData(){...}
    public void setData(Date d){...}

    public char getTipo(){...}
    public void setTipo(char t){...}
}

class Conta{
    private int codigo;

    public voidad dTransacao(Transacao t){...}
    public void setCliente(Cliente c){...}
}
class Cliente{
    private String nome;
    private String endereco;
    private Conta conta;

    public void setConta(Conta c){...}
    public Conta getConta(){...}
}
```

As associações entre objetos ficarão mais claras ao longo dos próximos capítulos. O estudo mais detalhado sobre UML não faz parte do escopo deste livro, mas seu estudo e compreensão serão muito úteis para o seu aprendizado sobre orientação a objetos.

ATIVIDADES

Observe as classes a seguir e responda:

Hora.java

```
public class Hora {
    private int horas, minutos, segundos;
    public Hora (int hours, int minutes, intseconds) {...}

    public int getHora(){...}
    public int getMinuto(){...}
    public int getSegundo(){...}
}
```

Funcionario.java

```
public class Funcionario {
    private static totalFuncionarios;
    private String nome;
    private boolean atraso;
    private double tempoTrabalhado, tempoAtraso;

    public Funcionario(String nome, Hora horaChegada, Hora horaSaida){...}

    public double tempo Atraso(Hora horaChegada){...}
    public double horas Trabalhadas(Hora horaChegada, Hora horaSaida){...}
    public double get HorasTrabalhadas(){...}
}
```

01. Quais são as variáveis de instância de Hora?

02. Quais são as variáveis de classe de Hora?

03. Quais são as variáveis de instância de Funcionário?

04. Quais são as variáveis de classe de Funcionário?

05. Qual é o tipo de associação entre Hora e Funcionário?

06. Como seria um programa principal, chamado ControleHorario.java, usado para controlar as horas trabalhadas e verificar se o funcionário está atrasado?

REFLEXÃO

Você vai notar que o paradigma da orientação a objetos é muito útil para representar o mundo real. A linguagem Java é uma grande representante deste paradigma e seu estudo constante é fundamental para o pleno conhecimento. Não deixe de investigar, praticar e principalmente usar a API da linguagem para saber como os comandos e métodos das classes mencionadas neste capítulo funcionam. Até mesmo o YouTube pode ajudá-lo com vários exemplos práticos e de uma maneira mais multimídia, mas tome cuidado com alguns conteúdos. Tenha cuidado em escolher suas fontes de estudo para não atrapalhar o que você já sabe. Há quem diga que a curva de aprendizado em Java é muito íngreme (difícil), mas depende de você. Lembre-se de que é a linguagem mais usada no mundo e os profissionais que realmente entendem seus mecanismos são profissionais muito procurados pelo mercado.

REFERÊNCIAS BIBLIOGRÁFICAS

CORNELL, G.; HORSTMANN, C. **Core Java 2**: Recursos Avançados. São Paulo: Makron Books, 2001.

CORNELL, G.; HORSTMANN, C. S. **Core Java - Vol. 1 - Fundamentos.** 8. ed. São Paulo: Pearson Education, 2010.

DEITEL, H. M.; DEITEL, P. J. **Java**: como programar. 8. ed. Rio de Janeiro: Pearson, 2010.

ECKEL, B. **Thinking in Java.** 4. ed. New Jersey: Prentice Hall, 2006.

FLANAGAN, D. **Java**: O guia essencial. 5. ed. Rio de Janeiro: Bookman, 2006.

HUBBARD, J. R. **Programação com Java**. 2. ed. Rio de Janeiro: Bookman, 2006.

SANTOS, F. G. D. **Linguagem de programação**. Rio de Janeiro: SESE, 2015.

SIERRA, K.; BATES, B. **Use a cabeça**: Java. 2. ed. Rio de Janeiro: Alta Books, 2007.

ZEROTURNAROUND. Java Tools and Technologies Landscape for 2014, 2014. Disponivel em: <https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/6/>. Acesso em: 27 jul. 2016.

3

Conceitos de herança

Conceitos de herança

Neste capítulo, vamos estudar um dos quatro pilares da orientação a objetos: a herança. A orientação a objetos possui quatro pilares, que são os elementos que sustentam este paradigma: a abstração, que é a forma de representar um objeto do mundo real, o encapsulamento, que aprendemos no capítulo anterior, a herança e o polimorfismo, que vamos estudar neste capítulo.

A herança quando bem empregada é um grande aliado do programador. Com ela é possível reutilizar o código que já foi feito anteriormente no mesmo programa ou em outros programas, até mesmo programas de empresas diferentes. A herança de fato otimiza a produção da aplicação em tempo e linhas de código.

Você pode estar pensando agora que deve ter algum relacionamento com o que realmente entendemos de herança, ou seja, algo que passa de pai para filho e forma uma família. Sim, o conceito é exatamente esse. O legal é que os conceitos que estamos aprendendo aqui em Java podem ser aplicados em qualquer linguagem orientada a objetos como C++, C#, Python, PHP e outras. É claro que, nas outras, há alguma variação, mas a ideia central é a mesma.

E como em Java tudo o que fazemos é por meio de classes e objetos, é natural supor que a herança estará baseada nas classes. Quer dizer que é possível criar uma hierarquia de classes? Sim. E isto é super importante e muito, muito prático, útil e fundamental.

Leia com atenção o capítulo. Busque outras referências sérias e consistentes na internet, você só tem a ganhar com isso.

Bons estudos!

OBJETIVOS

Ao final deste capítulo, você estará apto a:

- Criar estruturas hierárquicas de classes em Java usando herança;
- Aplicar os conceitos de classes abstratas nos seus programas;
- Usar interfaces para reutilização de código.

Introdução

Antes de entrarmos nos assuntos do capítulo, vamos adotar algumas convenções e apresentar um exemplo para, posteriormente entendermos, os conceitos propostos para este capítulo.

Podemos representar uma classe por meio de uma notação, um diagrama. Existe uma linguagem de modelagem chamada UML (*UnifiedModelingLanguage*) cujo objetivo é usar diagramas para representar um sistema orientado a objetos. A UML serve para diagramar classes, objetos e várias outras situações encontradas na análise de sistemas. Ela serve para qualquer linguagem orientada a objetos e não é exclusiva do Java. Já tratamos um pouco sobre UML no capítulo anterior.

A Figura 1 mostra uma classe Conta. Ela representa uma conta bancária qualquer simplificada. Vamos usar essa classe nos nossos exemplos daqui para frente. Na UML, alguns diagramas representam muito bem o código fonte em Java (ou C#, Python, PHP etc.) e tem uma correspondência direta. Veja logo após a figura, o código fonte correspondente à figura.

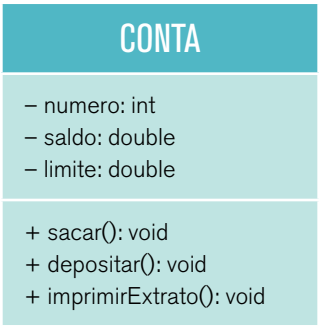


Figura 3.1 – Representação da classe Conta em UML

```
public class Conta {
   private int numero;
   private double saldo;
   private double limite;

    public void sacar() { ... }

    public void depositar() {...}
```

```
    public void imprimirExtrato() { ... }
}
```

Vamos explicar a figura; acompanhe também pelo código correspondente em Java. Ao ler o texto, olhe para a figura e para o código e acompanhe a explicação.

O nome da classe é fácil descobrir: ela fica na parte superior do retângulo do diagrama.

Na próxima parte do diagrama, ficam os campos, os atributos da classe: no caso, temos 3 variáveis de instância:

– *numero: int* → corresponde a um atributo private do tipo int chamado numero

– *saldo: double*→ um atributo private do tipo double chamado saldo

– *limite: double*→ um atributo private do tipo Double chamado limite

O sinal de subtração (-) significa que o elemento é private. O sinal de adição (+) significa que o elemento seguinte é public.

Na parte inferior do diagrama, encontram-se os métodos. Temos no exemplo 3 métodos:

+ *sacar(): void*→ corresponde a um método public chamado sacar sem retorno

+ *depositar(): void* → corresponde a um método public chamado depositar, sem retorno

+ *imprimirExtrato(): void*→ um método public chamado imprimirExtrato, sem retorno

Você pode perceber que o código em Java é totalmente compatível com os símbolos e anotações da figura. Essa é uma das grandes vantagens de se usar a UML para modelar sistemas orientados a objetos.

Vamos agora conhecer alguns pilares da orientação a objetos: a herança e o polimorfismo.

Herança e polimorfismo

A herança é um mecanismo fundamental para a orientação a objetos. Na verdade, é outro pilar da orientação a objetos. Na vida do desenvolvedor moderno, usar herança faz parte do dia a dia do trabalho. Usamos este mecanismo toda hora e nem percebemos.

Uma das grandes vantagens do uso da herança é sem dúvida a reutilização de código. Como exemplo, vamos pensar na modelagem do banco da Figura 2. Um banco normalmente oferece serviços para os seus clientes como empréstimos, financiamentos, seguros e outros.

Todo serviço, independente de qual seja, tem características semelhantes: possui um cliente, tem um valor, uma data de vencimento, data de abertura, entre outras. O programador desse sistema naturalmente vai desenvolver uma classe para um serviço com esses atributos, não é? Afinal, trabalhar com Java é olhar para o mundo real e traduzi-lo em classes e objetos.

Porém, quando ele for escrever o código para os tipos de serviços especificamente como o empréstimo, o financiamento, ele teria de escrever novamente as classes com esses atributos, repetindo todo o trabalho que teve anteriormente. A herança evita esse retrabalho.

Observe os exemplos no código em Java. A classe serviço completa está mostrada a seguir:

```
public class Servico{
    private Cliente contratante;
    private Funcionarioresponsavel;
    private Date dataDeContratacao;

    //dados do empréstimo
    private double valor;
    private double taxa;

    //dados do seguro
    private Carro carro
    private double valorSeguroCarro;
    private double franquia;
}
```

Na aplicação, teremos de criar objetos Servico. Nesta criação, os objetos serviços terão dados de empréstimo e seguro numa mesma instância. Isso não soa um tanto estranho? Pois um empréstimo deveria ter somente os dados de empréstimo, idem para um seguro! Baseado nisso, fica claro que devemos separar um serviço em cada classe.

O melhor é criar uma classe genérica, que contenha os dados comuns a todos os tipos de serviços e criar classes específicas. A classe genérica é chamada de superclasse (ou classe base ou classe mãe) e suas dependentes chamadas de subclasses, classes derivadas ou classes-filha. Isso é feito de acordo com o diagrama em UML a seguir:

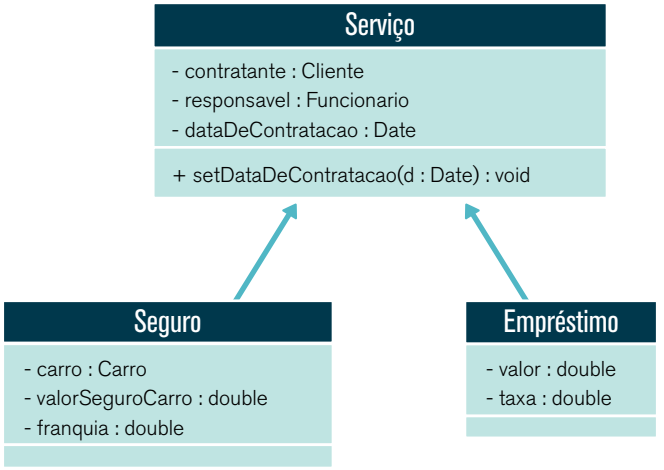


Figura 3.2 – Diagrama de classe representando herança.

O diagrama em UML é equivalente ao seguinte código em Java:

```
class Servico {
    private Cliente contratante;
    private Funcionarioresponsavel;
    private Date dataDeContratacao;

    setDataDeContratacao (Date d){
    dataDeContratacao = d;
}

class EmprestimoextendsServico {
    private double valor ;
    private double taxa ;
}
```

```
class Seguro extends Servico {
    private Carro carro ;
    private double valorDoSeguroCarro ;
    private double franquia ;
}
```

O comando extends faz o vínculo de herança entre as subclasses (Empréstimo e Seguro) e a superclasse Serviço. É importante entender que todo Empréstimo **é um** Servico, todo Gerente **é um** Funcionário, ou seja, toda subclasse antes de mais nada **é uma** parte da superclasse. Veja a figura 3.3.

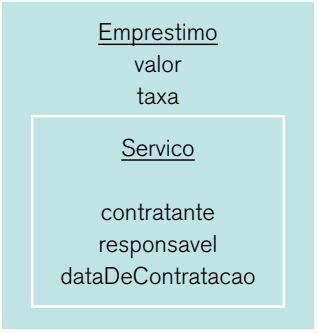


Figura 3.3 – Todo Empréstimo é um Serviço.

Ao criar um novo objeto Empréstimo, ele será também um Serviço:

```
Emprestimo e = new Emprestimo();
e.setDataDeContratacao("01/01/2015");
```

O atributo dataDeContratação não está definida na classe Empréstimo, e sim na classe Serviço. O objeto Empréstimo acessa o atributo da superclasse (porque ele é público e faz parte da família). Obs.: Por questão de simplificação do código, usamos a classe Date como parâmetro. O modo correto de uso desta classe deve ser observado na API pelo link <https://docs.oracle.com/javase/7/docs/api/java/util/date.html>.

Mexer com datas em qualquer linguagem de programação é uma questão delicada e cheia de detalhes. Consulte o link a seguir para ler e aprender mais sobre datas em java. É um link bem interessante: <http://blog.caelum.com.br/conheca-a-nova-api-de-datas-do-java-8/>.

Herança de métodos

É frequente o uso de métodos que são usados por todas as classes de uma determinada hierarquia. Podemos exemplificar com um atributo da classe Servico que represente uma taxa administrativa presente em todos os tipos de serviços. Este atributo será definido na classe Servico. Teremos também um método calcularTaxa() o qual é responsável por fazer os cálculos desta taxa. Todas as classes que herdam de Servico irão reaproveitar este método.

```
classServico {
   // atributos
    public doublec alcularTaxa(){
        return 100;
    }
}
Emprestimo e = new Emprestimo();
Seguro s = new Seguro();
e.calcularTaxa();   //veja que tanto o objeto e o objeto s utili-
zam o métodos.calcularTaxa();
```

O exemplo será incrementado: vamos supor que a forma de cálculo da taxa administrativa do empréstimo é diferente da forma das demais classes. Poderíamos criar um novo método dentro da classe Empréstimo chamado calcularTaxaEmprestimo(). Isso já resolveria o problema, mas ficaria um pouco estranho ter um método chamado calcularTaxa() e um calcularTaxaEmprestimo() disponível para o mesmo objeto. Além disso, com dois métodos que fazem praticamente a mesma coisa, algum programador poderia chamar o método errado. Podemos, então, reescrever o método calcularTaxa() e utilizar um método só, sobrecarregando-o (lembra-se desse conceito?). Dessa forma:

```
class Emprestimo extends Servico {
    // ATRIBUTOS
    public double calculaTaxa() {
        return this.valor*0.1;
    }
}
```

O método rescrito calculaTaxa() da classe Empréstimo possui a mesma assinatura da sua superclasse. Os métodos das subclasses têm maior prioridade sobre os métodos das superclasses. Ou seja, se o método chamado existe na subclasse ele será chamado, caso contrário o método será procurado na superclasse.

Vamos continuar incrementando o exemplo. A taxa agora será dada por um valor fixo somada a um valor que depende do tipo serviço (seguro ou empréstimo). A taxa do empréstimo terá como valor fixo 5 reais mais 10% do valor emprestado. O preço do seguro será 5 reais mais 5% do valor segurado.

```
class Emprestimo extends Servico {
    // ATRIBUTOS
    public double calculaTaxa() {
        return 5 + this.valor*0.1;     //10% do valor + R$5,00 fixos
    }
 }

class Seguro extends Servico {
    // ATRIBUTOS
    public double calculaTaxa () {
        return 5 + this.carro.getTaxa()*0.05; //5% do valor
    }
 }
```

O problema agora é: se o valor fixo tiver que ser alterado, todas as subclasses de Servico deverão ser alteradas. No nosso exemplo é fácil pois são 2 classes somente. Mas e se fossem bem mais do que isso? Seria um grande retrabalho. Uma solução é criar um método na superclasse para essa tarefa e depois replicar este método nas subclasses. Assim:

```
class Servico {
   public double calculaTaxa(){
       return 5 ;
   }
}
```

```
class Emprestimo extends Servico {
    // ATRIBUTOS
    public double calculaTaxa(){
        return super.calculaTaxa() + this.valor*0.1;
    }
}
```

Usando o comando super é possível acessar o método equivalente (mesmo nome e assinatura) da superclasse.

Quando tratamos os métodos e herança não podemos deixar de lado os construtores das classes. Como sabemos, os construtores são métodos especiais que inicializam os atributos e sendo assim merecem uma atenção especial quando existe uma hierarquia de classes.

Observe a figura 3.4 e lembre-se de que um Emprestimo **é um tipo de** Servico. Quando criamos um objeto empréstimo, o construtor da classe Emprestimo ou da classe Servico deve ser chamado. Se houver um construtor em cada uma dessas classes, o construtor da classe mais genérica é chamado antes (neste caso, o objeto empréstimo ao ser criado passa pelo construtor de Servico e depois Emprestimo). Quando não há construtores definidos, o compilador chamará o construtor sem argumentos da superclasse.

Construtores com parâmetros e herança

Vamos agora fazer um upgrade no exemplo. Agora de fato criaremos um diagrama bem completo envolvendo vários conceitos simultaneamente, portanto, preste atenção. É fácil! Mas requer atenção. Veja a figura 3.5.

O diagrama só é mais detalhado que os outros, mas é simples de ser entendido:

→ Um funcionário e um cliente são derivados de uma mesma classe, Pessoa;

→ Um funcionário participa de um Serviço sendo o responsável pelo contrato;

→ Um cliente também participa de um Serviço sendo o contratante;

→ Um Serviço neste caso pode ser de dois tipos: Seguro ou Empréstimo. Um Seguro é um Serviço e um Empréstimo é um Serviço também. Temos uma herança aqui, certo? Assim como em Pessoa, Funcionário e Cliente.

→ Um Carro participa de um Seguro.

O que queremos chamar a atenção aqui é no construtor de Empréstimo e Seguro.

Observe o construtor de Serviço: temos os parâmetros do tipo Cliente e Funcionário.

Observe agora as subclasses de Serviço: Um Seguro para ser instanciado precisa ter os mesmos parâmetros que a sua superclasse (Serviço) mais os parâmetros próprios: carro, valor e franquia. Observe o código fonte destas classes:

```
1     public class Servico {
2     private Cliente contratante;
3     private Funcionario responsavel;
4
5     public Servico (Cliente contratante, Funcionario responsavel) {
6         this.contratante= contratante;
7         this.responsavel=responsavel;
8     }
9
10    public Cliente getContratante() {...}
11    public void setContratante(Clientecontratante){...}
12    public Funcionario getResponsavel() {...}
13    public void setResponsavel(Funcionarioresponsavel) {...}
14 }
```

```
1  public class Seguro extends Servico{
2     private Carro carro;
3     private double valorSeguro;
4     private double franquia;
5
6     public Seguro (Cliente contratante, Funcionario responsavel,
Carro carro, double valor, double franquia) {
7         super(contratante, responsavel);
8         this.carro= carro;
9         this.valorSeguro= valor;
10    }
11
12    public Carro getCarro() {...}
13    public void setCarro(Carro carro) {...}
14    public double getValorSeguro() {...}
15    public void setValor Seguro(double valorSeguro) {...}
```

```
16      public double getFranquia() {...}
17      public void setFranquia(double franquia) {...}
18  }
```

```
 1
 2  public class Emprestimo extends Servico{
 3      private double taxa;
 4      private double valor;
 5
 6      public Emprestimo(Cliente contratante, Funcionario responsa-
vel, double taxa, double valor) {
 7          super(contratante, responsavel);
 8          this.taxa= taxa;
 9          this.valor= valor;
10      }
11
12      public double getTaxa() {...}
13      public void setTaxa(double taxa) {...}
14      public double getValor() {...}
15      public void setValor(double valor) {...}
16  }
```

Perceba que, nas classes Empréstimo e Seguro, especialmente nas linhas de seus construtores (linha 6), temos o construtor destas classes. Neles temos os mesmos parâmetros presentes na superclasse e mais os parâmetros para inicializar suas respectivas variáveis de instância.

Observe o uso do comando **super()** nas linhas 7 de cada classe. O nome "super" é sugestivo e faz com que a gente lembre que ele vai chamar algum método da **super**classe. Neste caso, o super() nas linhas 7 chama o método construtor da sua superclasse passando os parâmetros que instanciam um objeto da superclasse. Em seguida, na linha 8, o construtor prossegue a execução instanciando as variáveis da subclasse em questão. Percebeu que ao criar um objeto de uma subclasse, ele instancia primeiro a sua superclasse? No exemplo, todo empréstimo ou seguro é, antes de tudo, um serviço.

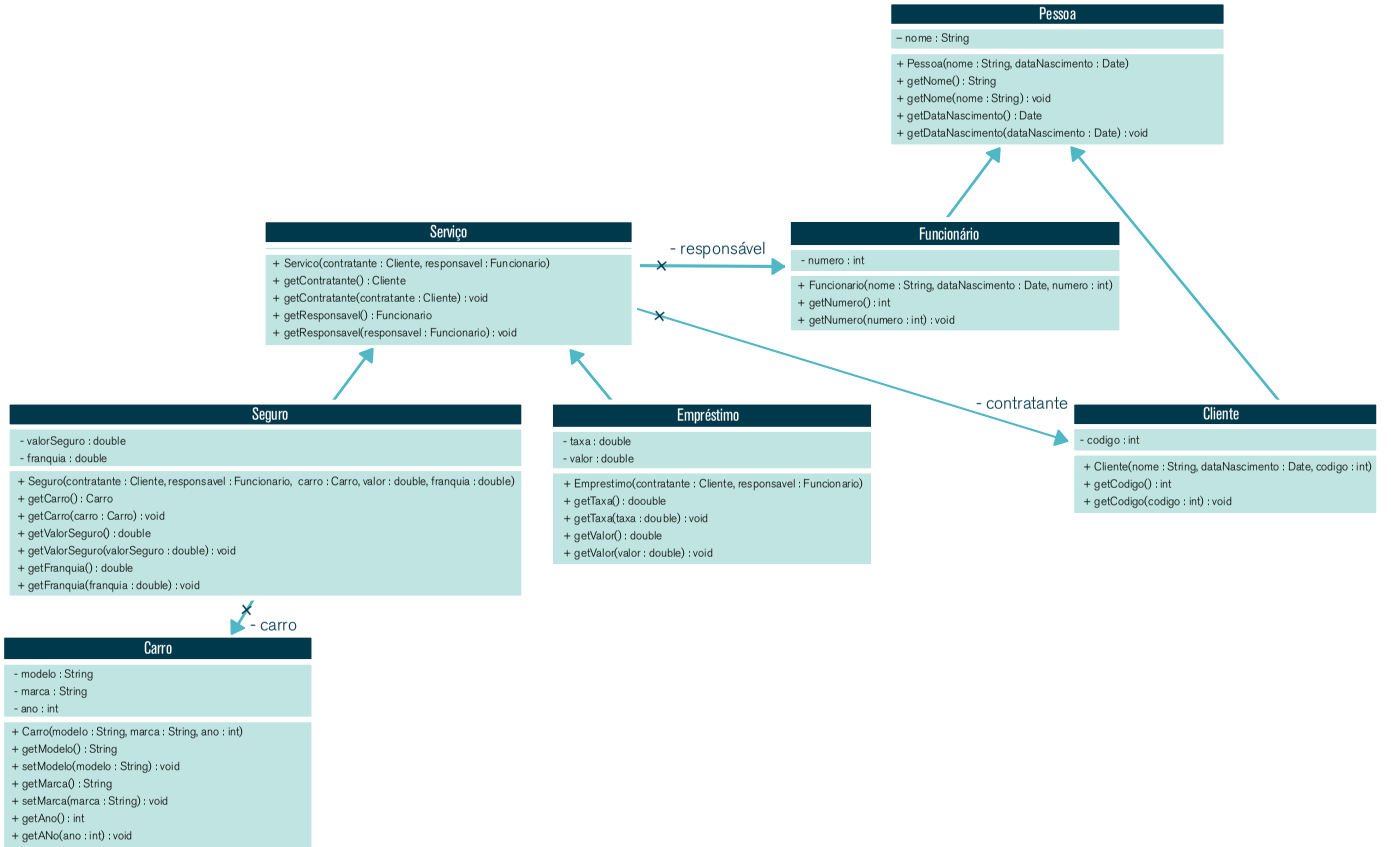


Figura 3.4 – Diagrama de classes mais detalhado.

Polimorfismo

A palavra polimorfismo possui 2 partes: poli (muitas); morfismo (formas) – muitas formas. O polimorfismo é outro recurso que contribui para a reutilização de código. No exemplo do banco que estamos usando, falta um controle de ponto dos funcionários.

Todo funcionário deve "picar o cartão" no início e no fim do expediente. Portanto, se pensarmos em uma classe para essa finalidade, precisaremos de 2 métodos: um para a entrada e outro para a saída. Porém em uma empresa, os tipos de funcionários possuem formas de controle de ponto diferentes. Logo, se para 1 tipo de funcionário precisamos de 2 métodos, para N tipos de funcionários precisaríamos de N * 2 métodos! E que fazem a mesma coisa! Imagine o retrabalho!

A solução para este problema é analisar o mundo real e abstraí-lo na programação. No mundo real temos vários tipos de funcionários: Gerentes, Atendentes, Caixas etc. Logo, percebe-se que existe uma relação de herança: Um funcionário que deriva outros tipos, ou seja, o gerente é um funcionário, o atendente é um funcionário, o caixa é um funcionário etc.

Na programação, a herança faz com que os objetos das subclasses criados sejam tratados como objetos da superclasse, ou seja, um objeto Gerente e um objeto Atendente são tratados como objetos da classe Funcionario. Já vimos isso quando usamos os construtores com o super().

```
class Gerente extends Funcionario // todo gerente é um funcionário ...
Gerente g = new Gerente();  //criamos um objeto Gerente
Funcionario f = g;   //tratando um gerente como da classe Funcionario
```

Vamos dar uma olhada na classe que poderia implementar o nosso ponto eletrônico:

```
class PontoEletronico {
    public void registraEntrada(Funcionario fun){
        //implementação do código
    }
    public void registraSaida(Funcionario fun){
        //implementação
    }
 }
```

Como já tratamos anteriormente, não faz sentido criar uma classe de ponto eletrônico para cada tipo de funcionário. A classe acima é uma boa forma de implementar a solução pois os métodos registraEntrada() e registraSaida() recebem como parâmetro um objeto da classe Funcionario e, sendo assim, podem receber referências de objetos de qualquer subclasse de Funcionario. Temos aqui então um exemplo de **polimorfismo**.

> Polimorfismo provém da biologia onde um organismo ou espécie pode ter várias formas ou estágios (sapo, borboleta etc). No nosso caso, polimorfismo é quando subclasses de uma classe pode definir seu próprio e único comportamento e ainda compartilhar as mesmas funcionalidades que sua superclasse.

A vantagem de usar o polimorfismo no ponto eletrônico é que qualquer alteração que for necessária só será feita em uma classe. Se novos tipos de funcionários forem criados, eles poderão acessar a classe PontoEletronico e seus métodos também.

Existem quatro tipos de polimorfismo na linguagem Java divididos em duas categorias: universal e ad-hoc. O polimorfismo ad-hoc é aquele que ocorre na compilação e divide-se em polimorfismo de sobrecarga e de coerção. Vamos ver rapidamente cada um deles:

Polimorfismo de sobrecarga (overloading)

Ocorre quando temos um método com o mesmo nome e assinatura diferente. Veja o exemplo:

```
class Calculadora {
    void somar(int a, int b){}           //1 parâmetro
    void somar(int a, int b, int c){}    //2 parâmetros
    void somar(float a, float b){}       //3 parâmetros
}
```

Observe que os três métodos têm o mesmo nome.

Polimorfismo de coerção

Ocorre quando há uma conversão implícita de alguns tipos para outro. O código abaixo exemplifica isso:

```
int a = 2;
double n = a;
```

Veja que a variável a foi definida como int. Porém, em seguida, o polimorfismo de coerção permite que ela seja implicitamente convertida para double.

Polimorfismo por inclusão

Alguns autores o consideram polimorfismo de subtipo ou de subclasse. A maioria das linguagens de programação possuem este tipo de polimorfismo. Ele também é chamado de *overriding* ou sobrescrita.

```
class Animal {
    public void correr() {
        System.out.println("Um animal pode correr.");
    }
}
class Cachorro extends Animal {
    public void correr() {
        System.out.println("Cachorros podem andar e correr");
    }
}

public class Teste {
    public static void main(String args[]) {
        // Cria uma referencia a um animal e um objeto
        Animal a = new Animal();
        // Cria uma referencia a um Animal e um objeto Cachorro
        Animal b = new Cachorro();

        a.correr();   // executa na classe Animal
        b.correr();   // executa na classe Cachorro
    }
}
```

O benefício da sobrescrita é sua capacidade de definir um comportamento específico para o tipo de subclasse, o que significa que uma subclasse pode implementar um método de classe pai com base na necessidade

Polimorfismo paramétrico

É o tipo de polimorfismo que permite que você use genéricos (Generics) para os subtipos. Todas as classes de coleção (Collections) do Java utilizam o conceito de polimorfismo paramétrico, que permite que você parametrize as classes dizendo qual tipo vai existir nas suas coleções. Veja o exemplo:

```
List <String> listaStrings = new ArrayList <List>();
listaStrings.add("string 1");
listaStrings.add("string 2");
```

Podemos ver aqui que a lista é uma lista de Strings. Quando usamos o operador <>, estamos dizendo ao compilador que tipo de classes que a coleção possui, parametrizando a coleção.

Classes abstratas

Em um banco, sabemos que existem vários tipos de contas: conta corrente, conta salário e conta poupança são alguns exemplos. Não é necessário explicar que, então, temos uma hierarquia de classes novamente na qual a superclasse é a Conta e suas subclasses são: ContaCorrente, ContaPoupanca e ContaSalario.

Na hora da criação de objetos, iremos sempre criar objetos de uma dessas três classes somente, não vamos precisar criar um objeto do tipo Conta porque ele será incompleto, ela não será uma conta com todas as suas propriedades e métodos, ela servirá apenas como uma base para poder criar outros tipos de contas. Logo, como criaremos somente objetos de um dos três tipos das subclasses, dizemos que essas classes são concretase Conta é uma classe abstrata. Uma classe concreta serve então para criar instâncias de objetos e a classe abstrata, não. Veja abaixo como definir uma classe abstrata:

```
abstract class Conta {
    //atributos, construtores e métodos
}
```

Dessa forma, não será mais possível criar um objeto Conta e a linha abaixo representará um erro ao compilar:

```
Conta con = new Conta();   // ERRO!! Conta é abstract, não pode
criar objetos
```

Métodos abstratos

É interessante que toda conta do nosso banco possa ter um método para imprimir o extrato dos seus lançamentos bancários. Você pode pensar que é um método a ser implementado na classe Conta e, assim, todas as outras subclasses poderiam usar este método para esta finalidade. Mas não esqueça que cada tipo de conta pode ter um extrato específico e assim a implementação desse método passaria a ser em cada subclasse, o que é mais natural.

O problema é que cada subclasse pode implementar de uma maneira diferente o método da impressão dos extratos e é sempre importante manter um padrão no desenvolvimento de software. Como a classe base será a classe Conta, ela poderia conter uma forma de padronizar o método de impressão de extrato e, assim, orientar que suas subclasses implementem o método de uma maneira uniforme.

Para garantir que cada subclasse (concreta) que provém direta ou indiretamente de uma superclasse tenha uma implementação de método para imprimir os extratos, inclusive com uma mesma assinatura (e manter o padrão comentado), usamos o conceito de métodos abstratos.

```
abstract class Conta {
    //atributos, construtores e métodos
    public abstract void imprimirExtrato (Date dataInicial, Date dataFinal){
    } // não há implementação aqui
}

class ContaCorrente extends Conta {
    //atributos, construtores, métodos
    public abstract void imprimirExtrato(Date dataInicial, Date dataFinal){
            // aqui vai a implementação do método
    }
}
```

Perceba que na classe Conta o método imprimirExtrato() não possui código de implementação. Isso só será feito nas subclasses que usarem este método. Nestas subclasses o método deverá ter o mesmo nome e a mesma assinatura que o método da superclasse e, além disso, ser obrigatoriamente implementado, senão ocorrerá um erro na compilação.

Interfaces

Em muitas situações da engenharia de software é importante que os desenvolvedores entrem em acordo com um "contrato", o qual mostra como o software irá interagir. Cada grupo de programadores deverá ser capaz de escrever o seu código sem conhecimento de como o código do outro grupo foi desenvolvido. Esses contratos mencionados, os acordos, são chamados de interfaces.

> Opa! Interface não seria a tela que o usuário vê em um programa? Você tem razão. Porém a palavra interface é abrangente na computação em geral. Temos diferentes tipos de uso da palavra interface. Uma delas é a interface gráfica, ou seja, o conjunto de telas que são apresentadas para o usuário em um programa. Podemos ter também uma interface como estamos estudando neste capítulo, como um conjunto de métodos relacionados sem a implementação de seus códigos. Na computação, especialmente na eletrônica, um modem é uma interface entre dois computadores, pois ele faz a interconexão de dois dispositivos. Na física, a interface é o meio que separa duas partes de um sistema, e por aí vai.

No exemplo do banco, podemos definir uma interface Conta para padronizar as assinaturas dos métodos oferecidos pelos objetos que representam as contas do banco.

```
interface Conta{
    void depositar(double quantia);
    void sacar(double quantia);
}
```

Os métodos das interfaces não possuem corpo nem implementação. Se uma interface funciona como um contrato, os métodos serão implementados obrigatoriamente nas classes concretas que "assinarem" este contrato.

```
class ContaSalario implements Conta{
    //atributos, construtores

    //implementação dos métodos da interface
    public void depositar(double quantia){
        ...
    }
```

```
    public void sacar(double quantia){
        ...
    }
}
```

A maior vantagem de usar interfaces é padronizar as assinaturas dos métodos que compõe a(s) interface(s) de um sistema. Outra vantagem é garantir que as classes concretas implementem o que foi estabelecido na interface.

> Um padrão de projeto é como uma especificação de uma determinada solução para um problema específico no software. Este conceito será mais bem explicado e desenvolvido em outras disciplinas. Mas, para antecipar, veja os padrões j2ee da oracle: eles estão "re-cheados" de interfaces. Um padrão muito útil e usado é o data access object (dao). Veja no link: <http://www.Oracle.Com/technetwork/java/dataaccessobject-138824.Html>.

As interfaces são usadas em Java para prover um mecanismo chamado herança múltipla. A herança múltipla permite que uma subclasse pode ser derivada de duas superclasses. A rigor, isso não existe em Java, mas existe em outras linguagens. Porém existe uma discussão muito grande, inacabada e inconclusiva em torno de herança, interfaces e herança múltipla.

Vamos usar as duas hierarquias a seguir para exemplificar:

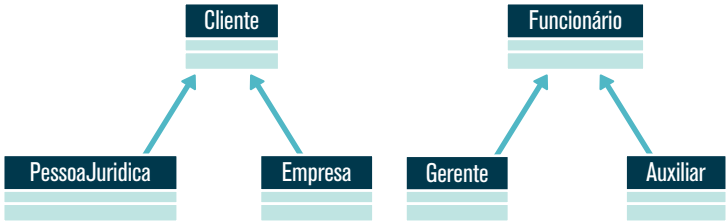


Figura 3.5 – Duas hierarquias independentes.

O banco pode ter uma classe para autenticar usuários no sistema. Os usuários podem ser funcionários do banco ou empresas externas, mas, neste caso, como implementar um método de autenticação que aceite tanto gerentes (que é de uma hierarquia) quanto empresas (de outra hierarquia)? Perceba que não há como usar polimorfismo neste caso e, para existir, devemos de alguma forma juntar as duas hierarquias, o que seria incorreto do ponto de vista natural (clientes e funcionários são entidades, coisas diferentes). Usando uma interface temos uma possibilidade:

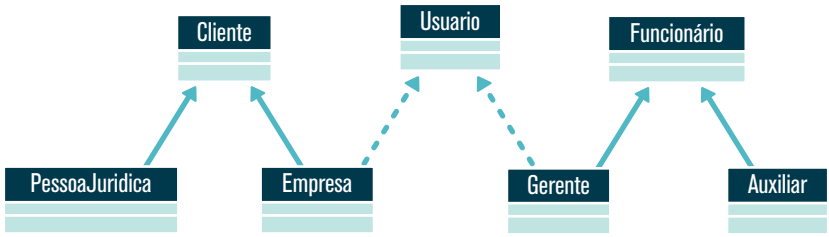
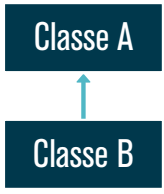


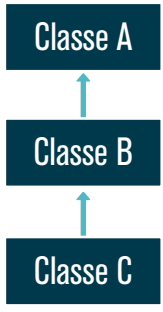
Figura 3.6 – Interface permitindo herança múltipla

Resumindo, baseado no que vimos neste capítulo podemos ter os seguintes tipos de herança:



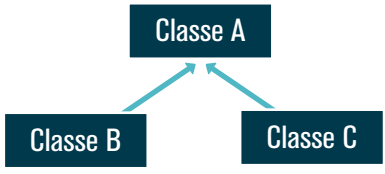

```
public class A {
        . . .
}
public class B extends A {
        . . .
}
```

Figura 3.7 – Herança simples.



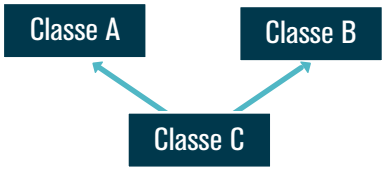

```
public class A {
        . . .
}
public class B extends A {
        . . .
}
public class C extends B {
        . . . .
}
```

Figura 3.8 – Figura 8: Herança multinível.




```
public class A {
        . . .
}
public class B extends A {
        . . .
}
public class C extends A {
        . . . .
}
```

Figura 3.9 – Herança hierárquica.




```
public class A {
        . . .
}
public class B extends A {
        . . .
}
public class C extends A,B {
        . . . .
}
Java NÃO suporta herança Múltipla
```

Figura 3.10 – Herança múltipla.

Lembre-se de que Java não suporta herança múltipla como mostrado na figura 3.10, ou seja, em Java não é permitido estender duas classes ao mesmo tempo, mas é um conceito existente e encontrado em outras linguagens. Entretanto, uma classe pode implementar uma ou mais interfaces e, neste caso, temos uma espécie de herança múltipla em Java.

Veja o exemplo a seguir:

```
interface I1 {
    abstract void teste(int i);
}
interface I2 {
    abstract void teste(String s);
}
```

Para mostrar o que acontece com os possíveis conflitos se você implementar várias interfaces: não há conflitos. Se várias interfaces têm exatamente o mesmo método exato, você simplesmente terá de implementá-lo. Se várias interfaces tiverem métodos semelhantes, você deve implementá-los. Ainda não terá conflito.

```
public class MultiplaInterface simplements I1, I2 {
    public void teste(int i) {
    System.out.println("I1.teste");
}
public void teste(String s) {
    System.out.println("I2.teste");
}
```

```
public static void main(String[] a) {
    MultiplaInterfaces t = new MultiplaInterfaces();
    t.teste(42);                //int
    t.teste("Hello");           //String
}
```

ATIVIDADES

Vamos fixar o que foi aprendido neste capítulo, propomos os seguintes exercícios.

01. Defina herança.

02. Quando a palavra-chave super é usada?

03. O que é polimorfismo?

04. O que é uma classe abstrata?

05. Crie uma classe chamada Ingresso que possui um valor em reais e um método imprimeValor().

a) crie uma classe VIP, que herda Ingresso e possui um valor adicional. Crie um método que retorne o valor do ingresso VIP (com o adicional incluído).

b) crie uma classe Normal, que herda Ingresso e possui um método que imprime: "Ingresso Normal".

c) crie uma classe CamaroteInferior (que possui a localização do ingresso e métodos para acessar e imprimir esta localização) e uma classe CamaroteSuperior, que é mais cara (possui valor adicional). Esta última possui um método para retornar o valor do ingresso. Ambas as classes herdam a classe VIP.

REFLEXÃO

Vimos, neste capítulo, alguns conceitos fundamentais para a orientação a objetos. Na verdade, daqui para frente, os seus programas terão muitas aplicações de herança e reuso de código. Cabe a você estudar um pouco mais além do que estamos mostrando aqui no livro para praticar e colocar em ação os conceitos envolvidos. Os conceitos aqui aprendidos serão

úteis em várias plataformas de execução de aplicativos como a web, dispositivos móveis, softwares embarcados e outros.

REFERÊNCIAS BIBLIOGRÁFICAS

CORNELL, G.; HORSTMANN, C. **Core Java 2:** Recursos Avançados. São Paulo: Makron Books, 2001.

CORNELL, G.; HORSTMANN, C. S. Core Java - Vol. 1 - Fundamentos. 8. ed. São Paulo: Pearson Education, 2010.

DEITEL, H. M.; DEITEL, P. J. **Java**: como programar. 8. ed. Rio de Janeiro: Pearson, 2010.

ECKEL, B. **Thinking in Java**. 4. ed. New Jersey: Prentice Hall, 2006.

FLANAGAN, D. **Java**: O guia essencial. 5. ed. Rio de Janeiro: Bookman, 2006.

HUBBARD, J. R. **Programação com Java**. 2. ed. Rio de Janeiro: Bookman, 2006.

SANTOS, F. G. D. **Linguagem de programação.** Rio de Janeiro: SESE, 2015.

SIERRA, K.; BATES, B. **Use a cabeça**: Java. 2. ed. Rio de Janeiro: Alta Books, 2007.

ZEROTURNAROUND. **Java Tools and Technologies Landscape for 2014**, 2014. Disponível em: <https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/6/>. Acesso em: 27 jul. 2016.

4

Tratamento de exceções

Tratamento de exceções

Durante a execução de um programa, independente da linguagem, podem ocorrer vários tipos de erros: um procedimento que busca dados de uma tabela que não existe, divisão por zero, abrir um arquivo em um diretório protegido e outros.

Todas estas interrupções geram uma parada na execução do programa e geram uma interrupção no sistema operacional. Se o sistema operacional falasse português, a pergunta seria mais ou menos assim: "Deu pau aqui! E agora, o que eu faço? Termino o programa, ignoro o problema e continuo ou o quê? ". E na prática é exatamente isso que o sistema operacional faz.

Se o programador não trata estes tipos de erros e interrupções, muitas vezes o sistema operacional vai gerar um código de erro bastante técnico que poderá ser exibido para o usuário e este, por sua vez, não saberá o que fazer com tal frase complicada. Nós que somos da área da informática temos uma chance maior sobre o que fazer, mas e quem não é? Não seria melhor tratar o erro e mostrar para o usuário uma mensagem mais amigável e menos "cabeluda"? Claro que sim.

E é disso que trata este capítulo: como pegar estas interrupções e tratá-las? Como tratar erros que podem ser causados pelos nossos programas? E por aí vai. Toda linguagem moderna de programação possui o tema "tratamento de exceção" e Java não é diferente. A diferença é que Java possui mecanismos muito poderosos e eficientes para o tratamento. Vamos estudá-los neste capítulo. Bom estudo!

OBJETIVOS

Ao final deste capítulo você estará apto(a) a:

- Capturar e tratar exceções ocorridas nos seus programas;
- Desenvolver programas mais eficientes por meio do tratamento de exceções.

Introdução

Em uma aplicação, podem ocorrer erros durante a execução de um programa:

→ uma divisão por zero imprevista;

→ uma falha na entrada de dados;

→ selecionar dados de um banco de dados que não está conectado no momento.

Enfim, existem várias possibilidades de erro que são provocadas por um desvio forçado no fluxo normal de execução do programa.

Estes erros são conhecidos como interrupções ou exceções. Eles podem ser resultados de algum tipo de sinalização de entrada/saída de um hardware ou, às vezes, o erro pode ser causado por uma ação do usuário. Existem alguns autores que diferenciam as palavras exceção e interrupção, outros consideram as duas palavras semelhantes.

O problema é que, quando isso ocorre, o programa termina de maneira anormal e isso não é recomendável. Temos, portanto, que de alguma forma tratar este tipo de problema.

Alguns programadores usam códigos alfanuméricos para representar os erros que ocorrem nos programas, como, por exemplo, no código abaixo, em que, se não houver valor suficiente para sacar o dinheiro em uma conta bancária, o método retorna o número 99 para o chamador. Depois disto, o chamador iria tratar este código (erro) devidamente e informar o usuário.

```
int sacar(double valor) {
    if(valorDisponivel< 0){
        return 99; // código de erro para valor negativo
    }
    else {
        this.valorDisponivel -= valor ;
        return 0; // sucesso
    }
}
```

Usar códigos de erros é uma alternativa e já foi bastante utilizado, porém isso representava um problema, pois documentar todos estes códigos era complicado e não produtivo. As linguagens de programação mais modernas possuem um mecanismo de tratamento de erro mais eficiente. Em Java não são usados códigos de erros ou outros tipos de retorno dos métodos. Como já foi dito em Java existe um mecanismo chamado Tratamento de Exceções.

Tipos de exceções

Baseado no que já foi dito, existem três categorias de exceções na linguagem Java que devem ser entendidas para poderem ser melhor tratadas:

Exceções checadas (checked)

Esta exceção ocorre na hora da compilação e, por isso, também são conhecidas como exceções em tempo de compilação. Por meio do uso de um bom IDE, o compilador just in time do Java verificará que poderá ocorrer uma exceção deste tipo e impedirá a compilação do código. Se o programador estiver usando um editor de texto como o Sublime, o editor permitirá salvar o arquivo; porém, ao chamar o compilador (javac), o compilador gerará um erro de compilação e o programador terá de corrigir o código para poder ser compilado.

Vamos ao exemplo:

É comum usar a classe FileReader para ler um arquivo de texto. Esta classe possui vários métodos que leem um arquivo de texto e manipulam seus dados. No construtor, se um determinado arquivo não existe ocorrerá uma exceção chamada FileNotFoundException e o compilador avisará o programador que isso é um erro, e não compilará o programa.

Tente compilar o programa a seguir:

```
import java.io.File;
import java.io.FileReader;
public class Excecao1 {
    public static void main(String args[]) {
        File file = new File("C://arquivo.txt");
        FileReaderfr = new FileReader(file);
    }
}
```

O resultado da compilação será:

```
C:\>javac Excecao1.java
FilenotFound_Demo.java:8: error: unreported exception
FileNotFoundException; must be caught or declared to be thrown
FileReaderfr = new FileReader(file);
^
1 error
```

O resultado acima é gerado quando se usa o compilador na linha de comando no Windows. Nos outros sistemas operacionais a mensagem será semelhante. O importante é notar que o código não será compilado.

Mas por que não será compilado? Como o compilador sabe distinguir isso?

Neste caso específico, a classe FileReader possui dois métodos read() e close() que lançam a exceção IOException. E, sendo assim, o compilador saberá que ela precisará ser tratada já na compilação. Vamos entender o que é esse "lançar a exceção" um pouco mais à frente.

Exceções não checadas (Unchecked)

Este tipo de execução passa pela compilação mas pode ocorrer durante a execução do programa. São chamadas de exceções em tempo de execução, ou runtimeexceptions.

Por exemplo, uma divisão por zero pode ocorrer em uma expressão. Além disso pode ser um erro de programação ou mesmo uma forma errada de usar alguma classe ou método da API.

Vamos ao exemplo:

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

Qual é o erro desta parte do programa?

Aparentemente nenhum não é? Porém se você observar atentamente, a linha 2 está tentando acessar o 6º elemento do array. Só que ele não existe. Mas sintaticamente as linhas estão corretas e isso não será barrado pelo compilador.

Porém, na execução, será gerada uma exceção chamada ArrayIndexOutOfBoundsException e interromperá a execução do programa. O duro é que, dependendo de onde estiver este código, o programa poderá ser executado milhões de vezes e nunca gerar o erro, mas dependendo do fluxo do programa poderá gerar logo na primeira execução. Ou seja, o compilador não saberá desta exceção, somente a ação do programa é que gerará a exceção.

E na hora que isso ocorrer, o JRE simplesmente parará o programa e vai gerar a seguinte mensagem na tela:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
    at Exceptions.Teste.main(Teste.java:xxx)
```

Obs.: xxx é o número da linha onde estará o comando que gerou a exceção.

Ou seja, não é nada agradável ler isso durante a execução de um programa. Se o usuário for um programador Java ele saberá o que houve, mas e se for um usuário comum?

Erros

O último tipo de exceção que podemos ter na classificação são os erros. Eles não são bem uma exceção mas sabemos que erros podem aparecer mesmo sem o controle do programador.

Os erros são simplesmente ignorados no código porque na prática você não tem muito o que fazer com os erros. Por exemplo: um estouro de pilha de execução (stack overflow). Pode ocorrer a qualquer hora durante a execução de um programa, não será detectado pelo compilador tampouco será previsto pelo usuário. Esse tipo de erro pode ocorrer quando o programa está executando em uma máquina com pouco recurso. Neste caso, infelizmente não temos como tratar em Java. Aliás, em nenhuma linguagem.

Vamos ver como o Java organiza isso.

Hierarquia de classes

A resposta para estes problemas em Java vem do mecanismo de tratamento de exceções. E como tudo em Java está baseado em classes e objetos, este mecanismo não poderia ser diferente.

Todas as exceções em Java são derivadas, ou seja são subclasses, da classe Exception do pacote java.lang. Por sua vez, esta é uma subclasse de Throwable, que tem como "irmã" a classe Error.

> Os erros são condições anormais no fluxo de programa que ocorrem em falhas graves que não são planejadas pelo programador. Normalmente os programas não conseguem se recuperar quando um erro ocorre.

A classe Exception tem duas subclasses importantes: IOExceptione RunTimeException. Veja a hierarquia apresentada na figura 4.1.

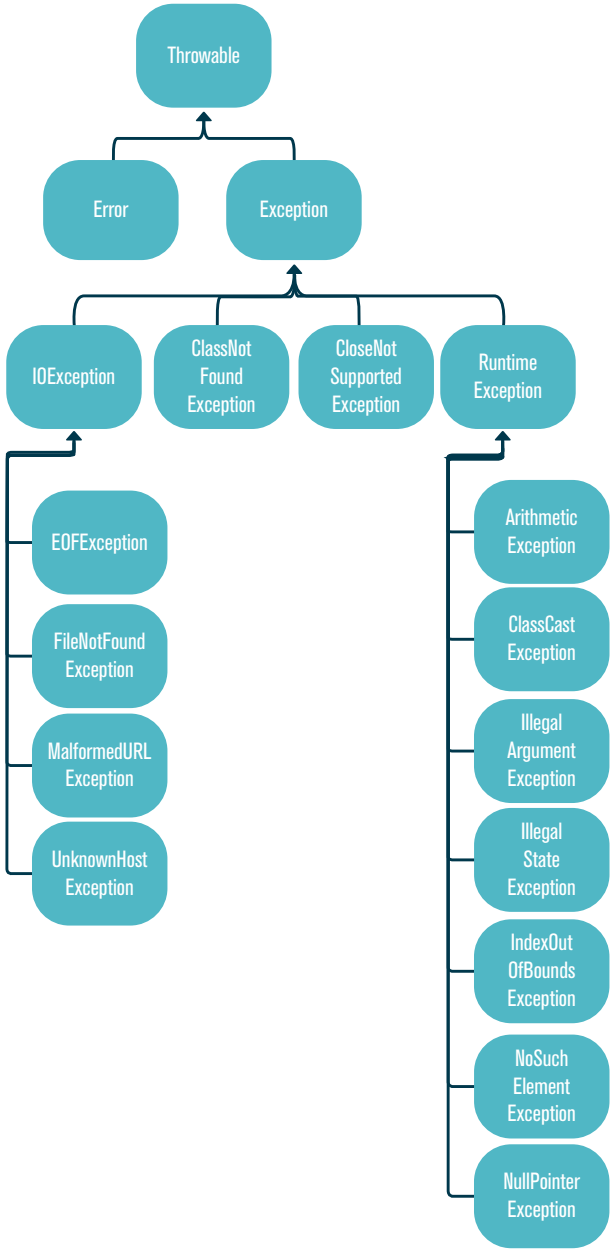


Figura 4.1 – Hierarquia de exceções em Java. As classes marcadas em vermelho são exceções checked e as azuis são as unchecked. Fonte: <https://docs.oracle.com/javase/7/docs/api/java/lang/package-tree.html>.

Portanto, a subclasse Exception possui métodos que definem erros nos quais a aplicação consegue tratá-los e continuar sendo executada. A subclasse Error possui métodos que definem os erros que não devem ser capturados pelas aplicações porque são erros graves e não possibilitam que a execução do programa continue satisfatoriamente.

A seguir apresentamos uma lista com as subclasses da classe RuntimeException. Veja na figura 4.1 que estas exceções são derivadas de java.lang, logo, não precisam ser importadas nos seus programas. Lembra-se das exceções unchecked explicadas no início do capítulo? Veja a lista delas na tabela 4.1 e tabela 4.2. Todas as classes que estão vinculadas à classe RunTimeException são unchecked. As outras classes que não fazem parte da RunTimeException são checked.

Nas tabelas a seguir você vai encontrar termos que não abordamos ainda nem vamos abordar neste livro. Porém, conforme você prosseguir nos seus estudos em Java, você vai acabar se deparando com estes termos e talvez com estas exceções. Por exemplo, quando você criar aplicações envolvendo bancos de dados, será obrigatório usar algumas classes de conexão com o banco de dados. Se estas classes não estiverem presentes você receberá uma exceção de classe não encontrada (ClassNotFound). Outros termos como thread, estados etc. aparecerão nas tabelas, mas não serão abordados aqui.

NÚMERO	EXCEÇÃO	BREVE DESCRIÇÃO
1	ArithmeticException	Erro aritmético, por exemplo, divisão por zero
2	ArrayIndexOutOfBoundsException	Quando um índice de um array não existe e tenta ser acessado
3	ArrayStoreException	Atribuição para um element de um array de tipo incompatível
4	ClassCastException	Cast (conversão de tipo) inválido
5	IllegalArgumentException	Argumento ilegal usado para chamar um método
6	IllegalMonitorStateException	Operação de monitoração ilegal, como o estado de espera em um thread desbloqueado.

NÚMERO	EXCEÇÃO	BREVE DESCRIÇÃO
7	IllegalStateException	Ambiente ou aplicativo está em estado incorreto.
8	IllegalThreadStateException	A operação solicitada não é compatível com o estado do thread atual.
9	IndexOutOfBoundsException	Algum tipo de índice está fora dos limites.
10	NegativeArraySizeException	Array criado com um tamanho negativo
11	NullPointerException	Uso inválido de uma referência nula (null)
12	NumberFormatException	Uso inválido de uma referência nula (null)
13	SecurityException	Uso inválido de uma referência nula (null)
14	StringIndexOutOfBounds	Uso inválido de uma referência nula (null)
15	UnsupportedOperationException	Uso inválido de uma referência nula (null)

Tabela 4.1 – Lista de exceções unchecked.

NÚMERO	EXCEÇÃO	BREVE DESCRIÇÃO
1	ClassNotFoundException	Classe não encontrada
2	CloneNotSupportedException	Tentativa de clonar um objeto que não implementa a interface Cloneable.
3	IllegalAccessException	Negação de acesso a uma classe
4	InstantiationException	Tentativa de criar um objeto de uma classe abstrata ou interface

NÚMERO	EXCEÇÃO	BREVE DESCRIÇÃO
5	InterruptedException	Uma thread foi interrompida por outra thread
6	NoSuchFieldException	Um campo solicitado não foi encontrado
7	NoSuchMethodException	Um método solicitado não existe

Tabela 4.2 – Lista de exceções checked.

Vamos exemplificar o tratamento de uma exceção do tipo ArithmeticException. Não se preocupe porque maiores detalhes serão mostrados ao longo deste capítulo:

```
public class Teste{
    public static void main(String []args)
        try{
            int x = 100;
            int y = 0;
            // Na linha abaixo encontramos o “problema”: não
            // podemos ter uma divisão por zero, sendo assim,
            // precisamos prever e tratar esta situação

            int calc = x / y;
            System.out.println("A resposta é: " + calc);
        }
        catch(ArithmeticException e){
            System.out.println("Não pode ser divido por zero!");
        }
    }
}
```

Métodos

Assim como mostramos a lista de exceções mais comuns, vamos apresentar uma lista de métodos disponíveis da classe Throwable:

- **getMessage():** retorna uma mensagem detalhada sobre a exceção que ocorreu. Esta mensagem é inicializada no construtor de Throwable.

• **getCause():** retorna a causa da exceção representada como um objeto Throwable.

• **toString():** retorna o nome da classe concatenada com o resultado de getMessage().

• **printStackTrace():** mostra o fluxo de saída de erro em uma exceção.

• **getStackTrace():** retorna um vetor contendo cada elemento na trilha da pilha. O elemento na posição 0 representa o topo da pilha de chamada, e o último elemento no vetor representa o método no fim da pilha de chamada.

• **fillInStacktrace():** preenche a pilha de rastreio de um objeto Throwable com o rastreio atual, somado a qualquer informação anterior na pilha de rastreio.

A seguir, vamos mostrar um exemplo de como usar o método printStackTrace(). Os outros têm um modo de funcionamento bem semelhante, mas, para você ter mais informações, não deixe de consultar a API da linguagem Java.

```
public class Teste{
    public static void main (Stringargs[]){
        int array[]={20,20,40};
        int num1=15, num2=10;
        int resultado=10;
        try{
            resultado = num1/num2;
            System.out.println("Resultado = " +resultado);
            for(int i =5;i >=0; i--) {
                System.out.println("Valor array = "+array[i]);
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

O resultado do programa é:

```
Resultado = 1
java.lang.ArrayIndexOutOfBoundsException: 5
at Teste.main(Teste.java:10)
```

O resultado mostra que o programa tenta varrer o array na ordem inversa dos seus índices e começa pelo índice 5, que não existe. Logo, o método printStaktrace() mostra a exceção que ocorreu, no caso, ArrayIndexOutOfBoundsException.

Capturando exceções

Quando um erro ocorre dentro de um programa, e isso normalmente ocorre dentro da execução de um método, este método cria um objeto e informa ao sistema de execução da JVM. O objeto, que é a exceção, contém informação sobre o erro, incluindo seu tipo e o estado do programa quando o erro ocorreu. Ao criar um objeto de exceção e fazer o seu tratamento no sistema de execução da JVM temos o que é chamado de **tratamento da exceção.**

Depois que um método lança uma exceção, o sistema tenta achar alguma coisa para tratá-lo. O conjunto possível de alternativas de tratamento é uma lista ordenada de métodos que são chamados para pegar o método onde o erro ocorreu. Essa lista de métodos é chamada de callstack (pilha de chamada), mostrada na figura 4.2.

O sistema de execução procura a pilha por um método que contenha um código que pegue (capture) a exceção. Este bloco é chamado de tratador de exceção. A procura começa com o método no qual o erro ocorreu e procede na pilha de chamadas na ordem reversa na qual o método foi chamado (veja a figura 4.2). Quando um tratador apropriado é encontrado, o sistema de execução passa a exceção para o tratador. Um tratador é considerado apropriado se o tipo do objeto lançado "casa" com o tipo que foi pego pelo tratador.

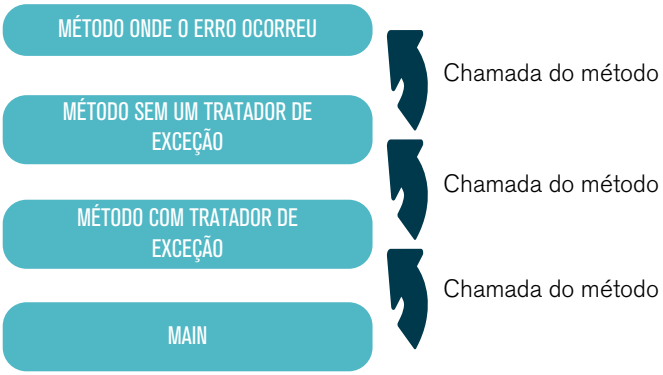


Figura 4.2 – pilha de chamadas (callstack).

O tratador escolhido é quem captura (catch) a exceção (vamos entender o catch um pouco mais à frente). Se o sistema de execução não encontrar um tratador apropriado como mostrado na figura 4.3, o programa termina e a mensagem de erro com a exceção é mostrada no console do sistema operacional executando o programa.

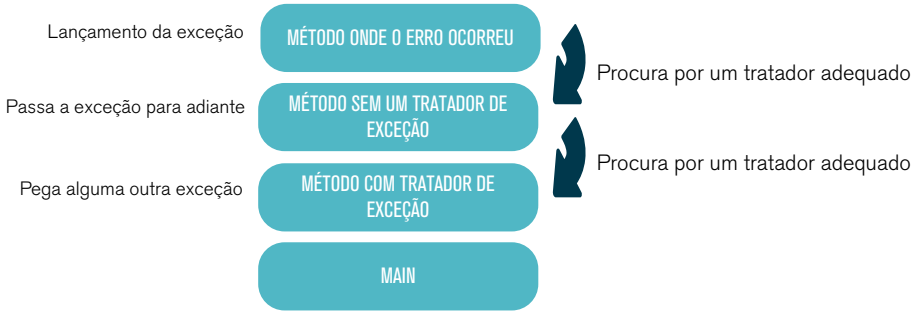


Figura 4.3 – Procura pelo tratador de exceção.

Um método captura uma exceção usando uma combinação dos comandos try e catch. Um bloco try-catch é colocado ao redor do código que pode gerar uma exceção, chamado de código protegido. Neste caso, o primeiro passo para construir um tratador de exceções é criar um bloco try. Este bloco normalmente tem a seguinte estrutura (os blocos catch e *finally* não são obrigatórios):

```
try {
    ... // código onde pode ser lançada uma exceção
}
catch (nome da exceção){
    ...,
}
catch (nome da exceção) {
...
}
finally {
    ...
}
```

O bloco catch contém o código que será executado se e quando o tratador de exceção for chamado. O sistema de execução da JVM chama o tratador de exceção quando o tratador for o primeiro da pilha de chamadas onde há o "casamento" do tipo da exceção com o tipo do tratador. Veja o exemplo:

```
1 try {
2  // código protegido onde pode conter uma exceção
3 } catch (IndexOutOfBoundsException e) {
4      System.err.println("IndexOutOfBoundsException: " +
e.getMessage());
5 } catch (IOException e) {
6      System.err.println("IOException: " + e.getMessage());
7 }
```

O bloco try executa a parte do código, na qual a exceção pode aparecer. Se a exceção for um índice não encontrado em um *array* (*IndexOutOfBoundException)* o tratador será o objeto "e" mostrado na linha 4. Caso ocorra uma exceção, como, por exemplo, uma falha ao escrever um arquivo, vai ocorrer a exceção IOException e o tratamento dela será feito a partir da linha 5. Neste caso, o tratamento é simples e só irá mostrar uma mensagem no console mas outros comandos mais elaborados podem ser colocados dentro do tratador de exceção, como, por exemplo mostrar uma janela de erros mais bonita, informar ao usuário o procedimento para evitar o erro novamente etc. Porém, neste caso, o programa terminará.

Para o programa não terminar, usamos o bloco *finally*. Ele sempre executa quando o bloco try termina. Isso garante que o bloco *finally* seja executado mesmo que uma exceção imprevista ocorra. Mas o *finally* é útil não somente para o tratamento de exceções – ele pode ser usado sempre.

Alguns links que podem ajudá-lo a entender um pouco mais sobre exceções em java:
<http://docs.Oracle.Com/javase/tutorial/essential/exceptions/>
<http://blog.Caelum.Com.Br/lidando-com-exceptions/>
<http://www.Tutorialspoint.Com/java/java_exceptions.htm>

É possível capturar e tratar múltiplas exceções dentro de um bloco de programa usando um simples catch. É claro que isso simplifica o código. Veja como pode ser feito:

```
catch (IOException|FileNotFoundException ex) {
    logger.log(ex);
    throw ex;
```

Uso de throw e throws

Se um método não trata uma exceção do tipo checked, o método deve declará-la usando o comando throws. Este comando é colocado no fim da assinatura do método.

Você pode lançar uma exceção ou mesmo uma instanciada recentemente, ou uma exceção que acabou de ser capturada usando o comando throw. Perceba que temos um "s" no fim da palavra agora.

Você deve entender a diferença entre throw e throws: throws é usada para adiar o tratamento de uma exceção checked e throw é usada para invocar uma exceção explicitamente.

Melhor ver um exemplo, não é? Observe o caso de um método que declara que lança uma RemoteException:

```
import java.io.*;
public class Conta {
    public void depositar(double quantia) throws RemoteException {
        // implementação do método
        throw new RemoteException();
    }
    // continuação da classe
}
```

Um método pode declarar que ele possui mais do que uma exceção. Neste caso, as exceções são declaradas em uma lista separada por vírgulas. Por exemplo, veja o exemplo a seguir onde temos um método que lança uma RemoteException e uma InsufficientFundsException:

```
import java.io.*;
public class Conta {
    public void saque(double amount) throws RemoteException,
InsufficientFundsException {
        // Implementação do método
    }
    // Resto da implementação da classe
}
```

Saiba mais sobre a insufficientefundsexception em <https://docs.oracle.com/cd/e22630_01/platform.1002/Apidoc/atg/commerce/claimable/insufficientfundsexception.Html>. Esta classe é derivada de commerce exception, que indica que um erro severo ocorreu durante uma operação comercial. Viu como java é abrangente? Existem muitas "coisas" prontas em java, basta explorar sua api e outras plataformas empresariais como sempre recomendamos. Neste caso, a classe insufficientfundsexception faz parte de uma plataforma da oracle específica para comércio na web chamada atg (atg web commerce).

Exceções definidas pelo usuário

Além de vários tipos de exceções já pré-programadas pela linguagem Java, pode ocorrer situações em que você necessite criar a sua própria exceção. Mas tenha em mente alguns pontos:

1. Todas as exceções são filhas da classe Throwable; logo, você tem de criar uma exceção que herde desta classe.

2. Se você quer criar uma exceção do tipo checked, que é automaticamente imposta pela regra Handle ou Declare, você precisa estender a classe Exception.

3. Se você quer criar uma execução de runtime, você deve estender a classe RuntimeException.

Quando determinado método está lançando qualquer exceção, o método que está chamando tem de lidar (handle) com isso, se o método que está chamando não quer fazer o tratamente, ele pode delegar a sua competência ao seu método de chamada. Essa regra é chamada de handle or declare.

Chega de "teoria", vamos à prática! Você pode criar sua própria exceção assim:

```
class MinhaExcecao extends Exception {
    ...
}
```

Você apenas precisa estender a classe Exception. Estas serão consideradas exceções verificadas (checked). Vamos criar uma classe chamada SemDinheiroException que estende a classe Exception, tornando-a uma exceção checked. Uma classe de exceção é como qualquer outra classe e possui métodos e atributos normalmente.

```
import java.io.*;
public class SemDinheiro Exception extends Exception {
    private double quantia;

    public SemDinheiroException(double quantia) {
        this.quantia = quantia;
    }
    public double getQuantia() {
        return quantia;
    }
}
```

Vamos agora usar esta exceção que acabamos de criar em uma classe chamada ContaBancaria que contém um método sacar() que lança a exceção SemDinheiroException. Observe e entenda o programa a seguir:

```
import java.io.*;

public class Contabancaria {
    private double saldo;
    private intnumero;

    public ContaBancaria(int numero) {
        this.numero = numero;
    }

    public void depositar(double quantia) {
        saldo += quantia;
    }

    public void sacar(double quantia) throws SemDinheroException {
        if(quantia<= saldo) {
            saldo-= quantia;
        }
```

```
        else {
            double valor = quantia - saldo;
            throw new SemDinheiroException(valor);
        }
    }

    public double getSaldo() {
        return saldo;
    }

    public int getNumero() {
        return numero;
    }
}
```

Vamos aumentar o exemplo e criar um programa mais completo envolvendo as classes criadas e usando os métodos depositar() e sacar().

```
public class Banco {
    public static void main(String [] args) {
        ContaBancaria c = new ContaBancaria(101);
        System.out.println("Depositando R$500...");
        c.depositar(500.00);

        try {
            System.out.println("\nSacando R$100...");
            c.sacar (100.00);
            System.out.println("\nSacando $600...");
            c.sacar(600.00);
        }
        catch (SemDinheiroException e) {
            System.out.println("Xi, falta R$" + e.getQuantia());
            e.printStackTrace();
        }
    }
}
```

Quando compilamos e executamos a classe Banco acima, temos o seguinte resultado:

```
Depositando R$500...

Sacando R$100...

Sacando $600...
Xi, falta R$200.0
SemDinheiroException
atContaBancaria.sacar(ContaBancaria.java:25)
        at Banco.main(Banco.java:13)
```

Vantagens do uso de Exceções

Segundo o site da Oracle, usar exceções em seus programas apresenta algumas vantagens:

1. Separar código sujeito a erro do código normal

Isso é muito importante!

O mecanismo do tratamento de exceções oferece formas de separar os detalhes do que fazer quando algo fora do normal ocorre da lógica principal de um programa. Na programação tradicional, a detecção, os relatórios e o tratamento de erros frequentemente levam a confundir o código. Como exemplo, veja o algoritmo a seguir:

Ler arquivo:
    Abra o arquivo
    Determine o seu tamanho
    Aloque a memória para ele
    Leia o arquivo no espaço alocado
    Feche o arquivo

É o algoritmo mais comum de leitura de um arquivo texto. Veja que, apesar de ser tarefas simples, temos várias situações de erros que podem ocorrer: o arquivo pode não ser lido, a memória pode não ser alocada por falta de espaço, o tamanho pode não ser determinado etc.

Com o mecanismo de exceções, podemos criar formas de tratar cada uma dessas exceções por meio de um bloco try-catch múltiplo.

2. Propagar os erros na pilha de chamada

O mecanismo de exceções possui a habilidade de propagar o relatório de erros na pilha de chamada de métodos. Vamos supor, usando o algoritmo do exemplo anterior, que existe um método chamado lerArquivo(), o qual é o quarto método dentro de uma sequência de métodos sendo chamados por um programa.

O programa executa o método1, que executa o método2, que executa o método3 que executa o lerArquivo(). Seria mais ou menos assim:

```
metodo1() {
    executa metodo2;
}
metodo2() {
    executa metodo3;
}
metodo3() {
    executa lerArquivo();
}
```

Suponha que metodo1 é o único método interessado nos erros que podem ocorrer dentro do lerArquivo(). Normalmente, a notificação de erro forçaria metodo2 e metodo3 a propagar os códigos de erro retornados por lerArquivo()na pilha de chamadas até que os códigos de erro finalmente alcancem metodo1, que é o único método que está interessado neles.

Ou seja, a linguagem Java guarda a chamada dos métodos na pilha de execução e caso o erro ocorra no lerArquivo(), o rastreamento do erro é feito até chegar no método que possua o tratamento de erro adequado. Neste caso, ficaria assim:

```
metodo1() {
    try {
        executa metodo2();
    }
    catch (exception e) {
        Trata o erro;
    }
}
```

```
Metodo2() throws exception {
    executa metodo3();
}

metodo3() throws exception {
    executa lerArquivo();
}
```

3. Agrupar e diferenciar tipos de erros

As exceções dentro de um programa são objetos. Logo, o agrupamento ou classificação das exceções é um resultado natural de uma hierarquia de classes. Portanto, você pode criar grupos de exceções e tratar exceções em uma forma geral, ou você pode usar o tipo de exceção específica para diferenciar exceções e tratar exceções em uma forma exata.

Por exemplo, dentro do programa de leitura de arquivo do nosso exemplo, você pode dividir os vários tipos de exceções que podem ocorrer e tratá-las separadamente de acordo com o tipo de erro.

Se for uma exceção de não ter encontrado o arquivo, você trataria a exceção FileNotFoundException. Por exemplo:

```
catch (FileNotFoundException e){ ... }
```

Se for algum problema de leitura do disco, você pode tratar a exceção como IOException. Exemplo:

```
catch (IOException e){ ... }
```

Ou se você preferir e quiser tratar exceções genéricas, após ter escrito o código de tratamento das exceções acima, você poderia usar a classe Exception, como no exemplo a seguir:

```
catch (Exception e) {...}
```

Porém, na maior parte dos casos, é interessante usar manipuladores de exceção para ser o mais específico possível. O motivo é que a primeira coisa que um manipulador deve fazer é determinar que tipo de exceção ocorreu antes que ele possa decidir sobre a melhor estratégia de recuperação do erro. Quando os erros específicos não são capturados, o manipulador deve tratar qualquer possibilidade. Manipuladores de exceção que muito genéricos podem tornar o código mais

propenso a erros por captura e manipulação de exceções que não foram antecipadas pelo programador e para o qual o manipulador não foi destinado.

ATIVIDADES

01.

O código a seguir está correto?

```
try {

}
finally {

}
```

02. Que tipos de exceção podem ser obtidos pelo tratador abaixo?

```
catch (Exception e) {

}
```

O que está errado usando este tipo de tratador de exceção?

03. Há algo errado com o tratador de exceção seguinte do jeito que está escrito? O código vai compilar?

```
try {

}
catch (Exception e) {

}
catch (ArithmeticException a) {

}
```

04. Associe cada situação na primeira lista com um item na segunda lista

a) a) int[] A;

b) A[0] = 0;

c) b) A JVM começa a executar seu programa, porém a JVM não consegue encontrar as classes da plataforma Java (as classes da plataforma Java ficam nos arquivos classes.zip ou rt.jar.)

d) c) Um programa está lendo um fluxo e alcança o fim do marcador de fluxo.

e) d) Antes de fechar o fluxo e depois de alcançar o fim do marcador de fluxo, um programa tenta ler o fluxo novamente.

1. Erro

2. Exceção verificada (checked)

3. Erro de compilação

4. Sem exceção

REFLEXÃO

O assunto de tratamento de exceções é visto em sala de aula, em livros como este, em vídeo-aulas, em treinamentos corporativos, enfim, é sempre comentado. Porém a gente vê pouco uso na prática a não ser quando são os casos de se usar um IDE que já detecta erros de compilação devido à falta do tratamento de exceções. Vários programas poderiam ser melhores só pelo fato de usar um bloco try-finally, por exemplo. Ao longo da sua vida profissional de programador e analista, você perceberá o quão importante é conhecer as boas práticas e usar as exceções nos programas que você cria. Dessa forma, várias mensagens de erro inapropriadas deixarão de ser mostradas nos aplicativos que usamos no nosso dia a dia.

REFERÊNCIAS BIBLIOGRÁFICAS

CORNELL, G.; HORSTMANN, C. **Core Java 2**: Recursos Avançados. São Paulo: Makron Books, 2001.

CORNELL, G.; HORSTMANN, C. S**. Core Java - Vol. 1 - Fundamentos**. 8. ed. São Paulo: Pearson Education, 2010.

DEITEL, H. M.; DEITEL, P. J. **Java**: como programar. 8. ed. Rio de Janeiro: Pearson, 2010.

ECKEL, B. **Thinking in Java**. 4. ed. New Jersey: Prentice Hall, 2006.

FLANAGAN, D. **Java**: O guia essencial. 5. ed. Rio de Janeiro: Bookman, 2006.

HUBBARD, J. R. **Programação com Java.** 2. ed. Rio de Janeiro: Bookman, 2006.

ORACLE CORPORATION. Advantages of Exceptions. **The Java™ Tutorials**, 2015. Disponível em: <https://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.html>. Acesso em: 01 jul. 2016.

SANTOS, F. G. D. **Linguagem de programação**. Rio de Janeiro: SESE, 2015.

SIERRA, K.; BATES, B. **Use a cabeça:** Java. 2. ed. Rio de Janeiro: Alta Books, 2007.

ZEROTURNAROUND. **Java Tools and Technologies Landscape for 2014**, 2014. Disponível em: <https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/6/>. Acesso em: 27 jul. 2016.

5

Coleções

Coleções

Já falamos, ao longo deste livro, o quanto a linguagem Java é poderosa e importante. Pode acreditar que muito do que você precisa em termos de recursos para programar aplicativos interessantes você encontrará nativamente na linguagem ou dentro da sua biblioteca de classes.

Um assunto que estudamos na faculdade, abordado em livros e treinamentos, é a manipulação de vetores e matrizes em Java. Sim, é um assunto importantíssimo para qualquer programador e dominar este assunto é sem dúvida fundamental. Mas você vai perceber, ao longo dos seus dias de trabalho como programador(a), que existem situações em que essas estruturas de dados deixam a desejar em tarefas simples. E é aí que aparece as coleções em Java, chamadas de Collections. Vamos apresentar as principais neste capítulo e dar a você uma base para poder explorar as outras por sua própria conta.

Além disso, vamos estudar outros assuntos tão importantes quanto as Collections: as classes "envelopadas" (wrapper) e os genéricos. Esses conceitos quando combinados formam estruturas de dados muito poderosas e úteis. Esperamos que você acompanhe e entenda os exemplos porque, certamente, farão diferença nos seus programas.

Bom estudo! Vamos lá!

OBJETIVOS

Ao final deste capítulo, você estará apto(a) a:

- Criar programas contendo coleções em Java;
- Usar as wrapped classes e aplicá-las nos seus projetos;
- Criar classes genéricas usando os conceitos dos Generics em Java.

Usar arrays em Java é muito importante. Sem dúvida são estruturas de dados que facilitam bastante o trabalho de qualquer programador em várias situações diferentes, mas você já parou para pensar que, embora úteis, eles dão um certo trabalho para manipulá-los? Por exemplo:

→ Não dá para redimensionar um array em Java;

→ Se você não souber um índice de um determinado valor, não dá para buscá-lo;

→ Não dá para saber quantas "casas" do array foram preenchidas, a não ser que façamos algum método para calcular isso;

→ Toda vez que eu quiser saber quantas posições estão preenchidas em um array, terei que percorrê-lo?

Percebeu? Esses são apenas alguns dos problemas que o programador encontra ao usar arrays. A linguagem Java resolve estes problemas rapidamente por meio de um conjunto de estruturas de dados já definidas chamadas Collections.

Não deixe de consultar a api da linguagem java para entender um pouco mais sobre as collections. O link é: <https://docs.oracle.com/javase/7/docs/technotes/guides/collections/overview.html>.

O framework das Collections em Java é formado ao redor de algumas interfaces padrão (lembre-se de que vimos sobre elas no Capítulo 3). Muitas destas interfaces podem ser usadas na forma como são apresentadas ou serem modificadas de acordo com a sua necessidade.

Portanto, um framework para Collections é uma arquitetura unificada para representar e manipular coleções. Esta arquitetura contém:

→ **Interfaces:** estes tipos de dados abstratos representam as coleções. As interfaces permitem que as coleções sejam manipuladas independentemente dos seus detalhes de representação. Já vimos que as interfaces podem formar hierarquias.

→ **Implementações (classes):** são as implementações concretas das interfaces das coleções. Na verdade, elas formam as estruturas de dados reusáveis.

→ **Algoritmos:** são os métodos que realizam a parte computacional, como, por exemplo, a procura, ordenação nos objetos da coleção. Os algoritmos são polimórficos, ou seja, os métodos podem ser usados em vários tipos de implementações diferentes da interface de coleção apropriada.

Vamos ver quais são as interfaces que podemos encontrar nas *Collections* em Java:

Interfaces

O *framework* das *Collections* define várias interfaces. Vamos dar uma geral em cada uma delas:

→ Collection: permite que você trabalhe com grupos de objetos. É a interface que está no topo da hierarquia.

List: estende a interface Collection. Uma instância de List armazena uma coleção a qual permite que elementos possam ser inseridos de acordo com uma ordem específica e permite também elementos duplicados.

→ Set: estende a interface Collection. Guarda conjuntos os quais devem conter elementos únicos;

→ SortedSet: estende a Set para lidar com conjuntos ordenados;

→ Map: é uma estrutura que mapeia chaves únicas a valores;

→ Map.Entry: descreve um elemento (ou seja, um par chave/valor) dentro de um Map. Na verdade, Map.Entry é uma classe interna (inner class) de Map;

→ SortedMap: estende Map de forma que as chaves são mantidas na ordem crescente.

→ Enumeration: é uma interface ligada que define os métodos pelos quais você pode enumerar (ou seja, obtido um por vez) os elementos dentro de uma coleção de objetos. Esta interface foi substituída pela Iterator.

Assim como existem as interfaces, temos as classes que compõem o framework da coleção em Java. As classes que implementam as interfaces de coleção têm nomes típicos no formato <estilo-da-implementação>Interface. Veja a tabela 5.1 para entender melhor a nomenclatura:

INTERFACE	SET	LIST	DEQUE	MAP
TABELA HASH	HashSet			HashMap
ARRAY REDIMENSIONÁVEL		ArrayList	ArrayDeque	
ÁRVORE BALANCEADA	TreeSet			TreeMap
LISTA LIGADA		LinkedList	LinkedList	
TABELA HASH COM LISTA LIGADA	LinkedHashSet			LinkedHashMap

Tabela 5.1 – Collections: Interfaces e suas implementações.

Tenho certeza que de você pode estar confuso(a) com tantos termos. Não se preocupe! Essas estruturas de dados (hash, árvore, lista ligada e outras) serão estudadas em uma disciplina específica na qual todas elas serão vistas com mais detalhes. Por hora, a página da wikipedia sobre estruturas de dados pode ser consultada para você ir se acostumando com elas: <https://pt.Wikipedia.org/wiki/estrutura_de_dados>.

A tabela 5.1 mostra, na primeira coluna, as principais estruturas de dados de acordo com a interface correspondente em Java e, nas colunas adjacentes, temos a classe que pode ser usada para implementar a estrutura que queremos. Então, veja só: se queremos usar uma lista de valores (um array), o qual seja redimensionável (entre outras características), podemos usar um ArrayList. Se queremos usar outra estrutura de dados chamada lista duplamente ligada (DEQUE), temos de procurar uma classe correspondente na tabela.

Veja a figura 5.1 para entender a hierarquia da interface Collection.

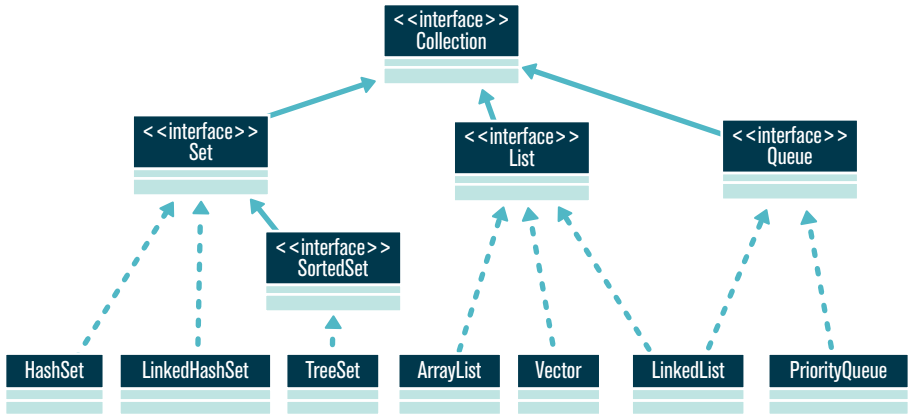


Figura 5.1 – Hierarquia do framework Collection.

Para sermos mais práticos e diretos, e para você entender melhor, usaremos uma classe que é bastante útil e muito usada em vários tipos de programas, como, por exemplo, aqueles que interagem com bancos de dados: a ArrayList.

ArrayList

Em primeiro lugar, gosto de dizer uma frase interessante: "uma coisa é uma coisa, outra coisa é outra coisa!". Ou seja, um array é um array. E um ArrayList é um ArrayList! Melhorando: um ArrayList não é um array! Antes de começar a explicar sobre ArrayList, já tenha isso em mente.

Outro detalhe: o ArrayList é uma implementação de List (lembre-se da tabela 5.1). Portanto, quando tratamos um ArrayList, estamos tratando de um List.

A ArrayList implementa todas as operações de lista opcionais, e permite todos os elementos, incluindo o null. Além de implementar a interface List, essa classe fornece métodos para manipular o tamanho do array que é usado internamente para armazenar a lista. Vamos à prática:

Para criar um ArrayList:

```
ArrayList minhalista = new ArrayList();
```

Como você pode perceber, não usamos os "[]" típicos da criação de um array. Ou seja, não definimos o seu tamanho. Logo, qual é a capacidade dela? A resposta é: o quanto de memória RAM separada para o seu programa permitir.

Outro detalhe diferente do *array* é que não especificamos o tipo da lista. Podemos inserir objetos nela e isso é diferente do array tradicional que você conhece. Ou seja, vamos trabalhar com instâncias da classe **Object**. É importante lembrar que toda Collection trabalha da maneira mais genérica possível.

Diferenças entre um array e ArrayList:

Embora parecidos, existem três diferenças fundamentais entre um array e um ArrayList:

1. Declaração:

Como já sabemos, um array é declarado da seguinte forma, por exemplo, um array de inteiros:

```
int[] vetor = new int[10]; // 10 posições
```

Um ArrayList:

```
List<int> vetor = new ArrayList<int>();
```

Logo, como podemos ver, no array já estabelecemos o seu tamanho. No ArrayList, o tamanho é dinâmico, ou seja, aumenta conforme o número de valores que são inseridos e removidos durante o seu uso.

2. Tamanho

Acabamos de falar sobre isso no final do tópico anterior. Usar um ArrayList pode ser mais vantajoso para poder economizar mais memória.

3. Índices

No caso de uma remoção em um array normal, o índice no qual o elemento foi removido ficará vazio, deixando uma "janela" no array. No ArrayList, após a remoção, a lista vai se ajeitar e "recuar", fazendo com que a lista fique sempre preenchida e contínua. Isso é uma grande vantagem para recuperar informações mais rapidamente na lista.

Também podemos abstrair a criação da lista a partir da interface List:

```
List minhalista = new ArrayList();
```

Para inserir nomes em uma lista:

```
minhalista.add("Fabiano");
minhalista.add("Vinicius");
```

Como podemos perceber, usamos o método add() para inserir elementos em um ArrayList. O método add() sempre insere os elementos no final. Existe outro método que permite inserir o item em qualquer lugar da lista.

Lembra da ContaBancaria que usamos como exemplo lá no Capítulo 3? Podemos criar uma lista de contas assim:

```
ContaBancaria cb1 = new ContaBancaria();
Cb1.depositar(500);

ContaBancaria cb2 = new ContaBancaria();
Cb1.depositar(50);

ContaBancaria cb3 = new ContaBancaria();
Cb1.depositar(550);

List listaContas = new ArrayList();

listaContas.add(cb1);
listaContas.add(cb2);
listaContas.add(cb3);
```

Veja o que podemos fazer com uma lista. O programa a seguir cria um ArrayList e faz várias operações que vão te ajudar a compreender melhor o que pode ser feito com este tipo de estrutura.

```
1   import java.util.*;
2
3   public class Teste {
4       public static void main(Stringargs[]) {
5           // criando a lista, chamamos de al (array list)
6           ArrayList al =new ArrayList();
7           System.out.println("Tamanho inicial: "+ al.size());
8
9           // inserindo elementos na lista
10          al.add("C");
11          al.add("A");
12          al.add("E");
13          al.add("B");
14          al.add("D");
15          al.add("F");
16          al.add(1, "A2"); //observe bem esta forma de inserir
17          System.out.println("Tamanho apos insercoes: "+al.size());
18
19          // mostrando a lista
20          System.out.println("Conteudo da lista: "+ al);
21
22          // Removendo elementos
23          al.remove("F");
24          al.remove(2);
25
26          if(al.contains("A")){
27              System.out.println("O elemento 'A' esta na lista");
28          }
29          else {
30                 System.out.println("O elemento 'A' NAO esta na lista");
31          }
32
```

```
33            System.out.println("Tamanho apos remocoes: "+ al.size());
34            System.out.println("Conteudo: "+ al);
35        }
36    }
```

O resultado da execução do programa é mostrado a seguir:

```
Tamanho inicial: 0
Tamanho apos insercoes: 7
Conteudo da lista: [C, A2, A,
O elemento 'A' NAO esta na li
Tamanho apos remocoes: 5
Conteudo: [C, A2, E, B, D]
```

O programa mostra alguns métodos importantes sobre o ArrayList:

→ **add():** como já falamos, este método serve para inserir elementos no final da lista (linhas 10 a 15) ou em uma determinada posição, passando esta como parâmetro (linha 16).

→ **size():** mostra o número de elementos da lista.

→ **remove():** este método remove um elemento da lista que é passado por parâmetro (linha 23) ou remove o item na posição desejada (linha 24).

→ **contains():** este método retorna true se o elemento passado por parâmetro está na lista ou false caso contrário.

Conhecemos os principais métodos da ArrayList. Vamos agora aplicar os conceitos que aprendemos na classe ContaBancaria. Veja a listagem do programa:

```
1    import java.util.*;
2    public class ContaCorrente {
3        private int numero;
4        private double saldo;
5        private static int totalContas;
6
7        public ContaCorrente(int n, double s){
8            numero = n;
9            saldo = s;
10           totalContas++;
11       }
12
```

```
13      public int getNumero(){ return numero; }
14      public double getSaldo(){ return saldo; }
15
16      public void depositar(doublevalor){
17          saldo += valor;
18      }
19
20      public void sacar(doublevalor){
21          if(saldo >= valor){
22              saldo -= valor;
23          } else {
24              System.out.println("Saldo insuficiente.\n");
25          }
26      }
27
28      public String toString(){
29          String retorno ="Conta "+getNumero()+" - Saldo: "+getSaldo();
30          return retorno;
31      }
32
33      public static void main(String[] args){
34          ContaCorrente.totalContas =0;
35
36          ContaCorrente conta1 =new ContaCorrente(1,100.0);
37          ContaCorrente conta2 =new ContaCorrente(2,200.0);
38          ContaCorrente conta3 =new ContaCorrente(3,300.0);
39
40          List contas =new ArrayList();
41          contas.add(conta1);
42          contas.add(conta2);
43          contas.add(conta3);
44
45          conta1.depositar(10);
46          conta2.depositar(22);
47          conta3.depositar(33);
48
```

```
49          for (int i=0; i<contas.size(); i++){
50              System.out.println(contas.get(i));
51          }
52      }
53  }
```

Vamos ao programa:

Nas linhas 2 a 31, temos o código que define a conta corrente. É uma classe bem simples contendo variáveis de instância e uma variável de classe.

Entre as linhas 33 e 52, temos a definição do método principal, onde teremos a execução dos testes com o ArrayList criado na linha 40.

Usamos os métodos add() e size() e apresentamos o get() na linha 50. Este método retorna o objeto existente em determinada posição da lista. Como fizemos um loop que percorre a lista nas linhas 49 a 51, o objeto de cada posição será impresso. Como temos um método toString() na definição da ContaCorrente, o loop usará o toString() para imprimir as informações na tela conforme o resultado a seguir:

```
Conta 1 - Saldo: 110.0
Conta 2 - Saldo: 222.0
Conta 3 - Saldo: 333.0
```

Mas tem um detalhe: se o loop fosse substituído pelo código a seguir, nada iria ser mostrado na tela, pois o objeto da posição "i" (que é uma conta corrente) por si só não foi programado para mostrar informação alguma.

```
for (int i=0; i<contas.size(); i++){
    contas.get(i);
}
```

E se quiséssemos que o saldo seja mostrado em cada repetição do loop? Por exemplo, algo como:

```
contas.get(i).getSaldo();
```

Desta forma não será possível. E é aí que você tem que entender como o framework trabalha. O get vai retornar sempre uma instância da classe Object. Para que o saldo seja recuperado, é necessário fazer um cast (conversão) para ContaCorrente da seguinte forma:

```
for (int i = 0; i < contas.size(); i++) {
    ContaCorrente conta = (ContaCorrente) contas.get(i);
    System.out.println(conta.getSaldo());
}
```

Apesar de ser possíve, não se usa o "for" tradicional para percorrer este tipo de lista. Neste caso, é recomendado usar um objeto da classe iterator ou o "for extendido".

Os exemplos mostraram a mecânica principal de trabalho com os ArrayLists. A seguir vamos apresentar outros métodos que podem ajudá-lo na manipulação dos itens deste tipo de estrutura:

→ addAll(Collection c): insere todos os elementos da coleção c no fim da lista, na ordem que eles são retornados pelo iterador específico da coleção;

→ addAll(int i, Collection c): veja que é o mesmo método que o anterior, porém, neste caso, a coleção c será inserida na posição i;

→ clear(): remove todos os elementos da lista;

→ clone(): como o nome sugere, "clona" a lista (faz uma cópia);

→ ensureCapacity(int minCapacity): aumenta a capacidade da lista, se necessário, para garantir que ela consiga guardar ao menos o número de elementos passado no parâmetro;

→ indexOf(Object o): retorna o índice na lista da primeira ocorrência do elemento especificado no parâmetro, ou retorna -1 se a lista não possui o elemento;

→ lastIndexOf(Object o): retorna o índice na lista da última ocorrência do elemento especificado, ou -1 se a lista não possui o elemento;

→ removeRange(int inicio, int fim): remove todos os elementos da lista que estiverem entre os índices especificados nos parâmetros;

→ set(int indice, Object elemento): substitui o elemento na posição especificada no parâmetro pelo elemento do segundo parâmetro;

→ toArray(): retorna um array de todos os elementos na lista.

Você deve ter percebido que os métodos mostrados na relação são bem úteis e que, se usássemos arrays tradicionais, seria um pouco mais complicado implementá-los não é? Logo, para que reinventar a roda? Já existe muita coisa pronta! Use as coleções quando possível ao invés dos arrays e aproveite estes métodos. Existem outros métodos que vamos deixar por sua conta pesquisar e verificar como funcionam. Como sempre, lembre-se da API!

Veja um exemplo e, em seguida, o resultado da execução mostrando vários métodos citados anteriormente:

```
1    import java.util.*;
2
3    public class Teste {
4
5        publicstatic void main(Stringargs[]) {
6            // Criando uma lista vazia
7            ArrayList<String> lista =new ArrayList<String>();
8
9            // inserindo itens na lista
10           lista.add("Item1");
11           lista.add("Item2");
12           //insere "Item3" na terceira posição da lista
13           lista.add(2, "Item3");
14           lista.add("Item4");
15
16           // Mostra o conteudo da lista
17           System.out.println("Lista: "+ lista);
18
19           // Verificando o índice de um  item
20           int pos = lista.indexOf("Item2");
21           System.out.println("O indice do Item2: "+ pos);
22
23           // Verificando se uma lista está vazia
24           boolean check = lista.isEmpty();
25           System.out.println("A lista esta vazia? "+ check);
26
27           // Verificando o tamanho da lista
28           int tamanho = lista.size();
29           System.out.println("Tamanho da lista: "+ tamanho);
30
31           // Verificando se um elemento esta na lista
32           boolean elemento = lista.contains("Item5");
33           System.out.println("Verificando se o objeto Item5 esta na
lista: "+ elemento);
```

```
34
35          // Pegando o elemento de uma posição específica
36          String item = lista.get(0);
37          System.out.println("O item no indice 0: "+ item);
38
39          // Recuperando os elemento de uma arraylist
40          System.out.println("Lista: ");
41          for (String str : lista) {
42              System.out.println("Item: "+ str);
43          }
44
45          // Recolocando um elemento
46          lista.set(1, "NovoItem");
47          System.out.println("A arraylist apos a recolocacao: "+
lista);
48
49          // Removendo items
50          // removendo o item no indice 0
51          lista.remove(0);
52
53          // removendo a primeira ocorrencia do item "Item3"
54          lista.remove("Item3");
55
56          System.out.println("O  conteudo  final  da  arraylist:"+
lista);
57
58          // Convertendo ArrayList para Array
59          String[]   arrayNormal   =   lista.toArray(new   String
[lista.size()]);
60          System.out.println("O array criado apos a conversao da
arraylist: " + Arrays.toString(arrayNormal));
61      }
62  }
```

A execução é:

```
Lista: [Item1, Item2, Item3, Item4]
O indice do Item2: 1
A lista esta vazia? false
Tamanho da lista: 4
Verificando se o objeto Item5 esta na lista: false
O item no indice 0: Item1
Lista:
Item: Item1
Item: Item2
Item: Item3
Item: Item4
A arraylist apos a recolocacao: [Item1, NovoItem, Item3, Item4]
O conteudo final da arraylist: [NovoItem, Item4]
O array criado apos a conversao da arraylist: [NovoItem, Item4]
```

Classes Wrapper

Vimos no final do tópico anterior um recurso no qual o loop percorria uma lista, na verdade um ArrayList, e para cada posição era executado um método chamado get(), que retornava o objeto da posição atual do loop.

Porém, caso fosse necessário chamar um método da classe do objeto, não seria possível. Para resolver o problema, usamos um cast para a classe necessária e deu tudo certo.

A linguagem Java oferece um recurso chamado wrapper ou classes empacotadoras (wrapper classes), ou invólucro segundo alguns autores, que possuem métodos que fazem conversões em variáveis primitivas e também encapsulam tipos primitivos para serem usados como objetos, ou seja, "embrulham" o conteúdo em outro formato.

Na programação orientada a objetos, uma classe wrapper é uma classe que encapsula os tipos, de modo que esses tipos podem ser usados para criar instâncias e métodos de objetos em outra classe que precisa desses tipos. Assim, uma classe wrapper é uma classe que encapsula, oculta ou envolve os tipos de dados dos oito tipos de dados primitivos da linguagem (veja o Capítulo 1), de modo que estes

podem ser usados para criar objetos instanciados com métodos em outra classe ou em outras classes.

Vamos lembrar de outro exemplo para ajudar no seu entendimento. No Capítulo 1, usamos alguns métodos para converter tipos de dados ao fazer a leitura do teclado, como, por exemplo, no código abaixo:

```
1 import javax.swing.*;
2
3 public class Teste {
4     public static void main(String args[]) {
5     String  entrada  =JOptionPane.showInputDialog("Quantos  anos
voce tem?");
6     int anos = Integer.parseInt(entrada);
7     System.out.println ("Voce  tem  " + 365*anos+" dias de ida-
de aproximadamente.");
8     }
9 }
```

Olhe a linha 6. Aí está um exemplo de uma classe wrapper. Portanto, temos uma classe wrapper para cada tipo primitivo da linguagem identificado pelo mesmo nome do tipo que possui com a primeira letra maiúscula. Veja a figura 5.3 para entender como funciona a hierarquia das classes wrapper.

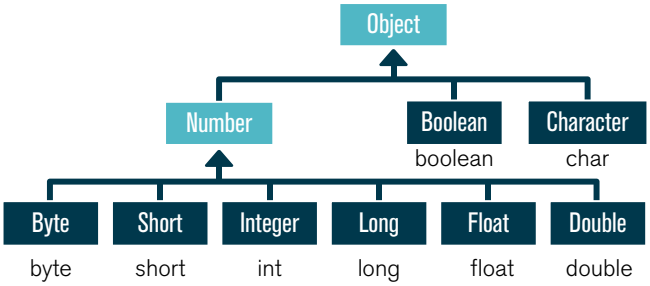


Figura 5.2 – Hierarquia das classes wrapper.

O uso das classes wrapper é bem abrangente mas no nosso caso percebemos sua aplicação quando lidamos com objetos de coleção (Collection) do Java.

Agora cabe outra reflexão a respeito de usar recursos naturais da linguagem ou da API. No tópico anterior, fizemos isso com os arrays e Collections. Ou seja, é melhor usar um array ou uma classe Collection? O mesmo raciocínio pode ser

aplicado aqui. No caso de um número inteiro, é melhor usar o tipo int ou a classe Integer, uma vez que esta, por ser uma classe, possui métodos específicos para trabalhar com números inteiros e facilitar a nossa vida de programador. A resposta é que cada programa tem uma necessidade diferente e o uso vai depender do caso. Apenas tenha em mente que estamos falando de tipos primitivos e objetos, ou seja, são estruturas e conceitos diferentes.

Temos de lembrar que as classes wrapper de tipos primitivos "apenas" encapsulam (ou embrulham) os tipos primitivos e os armazenam. Mas como já dissemos, se elas são classes, então possuem métodos e estes podem ser usados a nosso favor. Veja o caso da classe Integer e seus métodos no link: https://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html. Você vai encontrar métodos para converter Integer para outros formatos, obter o complemento de dois do número (o complemento de dois é usado na representação binária de um número), além de outros métodos úteis.

Vamos exemplificar com alguns pedaços de código para você entender melhor:

Observe com atenção:

```
String numero1 = "1234";
String numero2 = "123.45";
```

Olha o que podemos fazer:

```
Float fnum = new Float(numero2);
ou
Float fum = new Float("123.45");

    Integer inum = new Integer(numero1);
    ou
    Integer inum = new Integer("1234");
```

Para fazer a conversão de tipos primitivos para classes wrappers podemos usar os seguintes métodos:

→ **xxxValue():** usado quando precisa realizar uma conversão do valor de um objeto wrapper para um objeto primitivo. É claro que o "xxx" na frente do "Value" varia de acordo com o tipo, por exemplo doubleValue(), intValue() etc;

→ **parseXxx(String s):** já conhecemos este método, não é? É usado para converter um objeto String para um tipo primitivo e retorna um primitivo nomeado. O "xxx" deve ser substituído pelo tipo desejado;

→ **valueOf(String s):** usado para converter um objeto String para um objeto wrapper. O método retorna um objeto wrapper recém-criado do tipo que chamou o método;

→ **toString():** também já conhecido. Ele retorna a representação de um objeto em formato String (tipo primitivo encapsulado).

Vamos mostrar a aplicação destes métodos com alguns exemplos:

```
Criando um objeto wrapper
Integer dias = new Integer(365);

Convertendo para um tipo primitivo:
int dias_i = dias.intValue();

Conversão de uma String para o tipo primitivo
double valor = Double.parseDouble("123.45");
System.out.println("Valor = "+valor);

Exemplo do método valueOf com a classe String
Integer valor = new Integer(25);
String vString = String.valueOf(valor);

String valor = new String("123.45");
Double vDouble = Double.valueOf(valor);

System.out.println("Valor string: "+vString);
System.out.println("Valor double: "+vDouble);
```

Agora que aprendemos sobre as classes wrapper, vamos começar a relacionar os assuntos deste capítulo.

Na primeira parte do capítulo, estudamos sobre as coleções. Vimos que não podemos colocar um inteiro (int) ou qualquer outro tipo primitivo dentro de um ArrayList (ou outro membro das coleções em Java). As coleções só podem guardar objetos, então temos de dar um jeito de encapsular os tipos primitivos na sua classe wrapper correspondente (no caso do int, usamos a classe Integer).

Quando removemos um objeto de uma coleção, estamos recuperando um Integer, não é? E para obter o seu int correspondente, vamos usar o método intValue() que vimos neste tópico. Este mecanismo tem um nome na orientação a objetos em geral chamado box e unbox:

→ **box:** quando colocamos o tipo primitivo na sua classe wrapper correspondente.

→ **unbox:** no processo contrário, quando retiramos o valor da classe e obtemos o valor primitivo.

Na verdade, este trabalho de fazer o box e o unbox é meio chato e custoso, além de "poluir" o seu código. Será que não existe uma forma mais prática de automatizar isso? Uma das formas de fazer esta automação necessária é usar outro conceito muito útil em Java chamado Generics (tipos genéricos).

Generics

Agora voltaremos às coleções. Não falamos muito da classe List, pois usamos uma implementação específica chamada ArrayList. Mas, se você entendeu o conceito das listas, você deve ter entendido que podemos inserir qualquer objeto nelas, certo? Logo, podemos misturar objetos em uma lista, assim:

```
Criamos uma lista:
List minhaLista = new ArrayList();
Criamos dois objetos de classes diferentes:
Integer valorInteiro = new Integer("123");
Double valorDouble = new Double("123.45");

Criamos até uma ContaCorrente:
ContaCorrente conta1 = new ContaCorrente(1,100.00);
```

Vamos misturar tudo na minhaLista:

```
minhaLista.add("Uma string qualquer");
minhaLista.add(valorInteiro);
minhaLista.add(valorDouble);
minhaLista.add(conta1);
```

Mas isso é possível? Claro que é! Lembre-se: são objetos!

Se fosse possível desenhar esta lista, o resultado seria como mostrado na figura 5.4. Veja que a lista é referenciada por uma "variável" chamada minhaLista e que cada elemento da lista é ligado a outro. O último elemento da lista (conta1) não é ligado a nenhum elemento e, logo, não possui um próximo item. Não se preocupe com a representação gráfica, por hora perceba que a lista conterá objetos de classes diferentes.

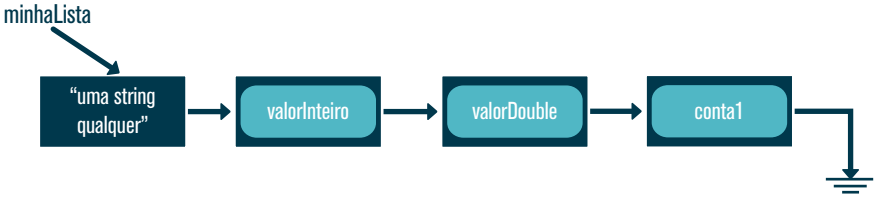


Figura 5.3 – ArrayList minhaLista.

Temos um problema. E se precisarmos recuperar os valores desta lista? Vimos o método get() e aprendemos que usamos o cast para recuperar os valores. Só que o problema agora é que temos uma lista com objetos de tipos diferentes, e isso não vai dar certo. Outro detalhe: não é nem um pouco recomendável, nem útil, que uma lista misture tipos. Para poder "fechar" a lista e permitir apenas o uso de um determinado tipo, usamos os Generics.

Vamos resolver este problema no exemplo a seguir, com as classes das contas bancárias. Uma agência é um conjunto de contas, não é? Logo, podemos implementar uma agência como uma lista de contas. Vamos lá:

```
List<ContaCorrente> contas = new ArrayList<>();
contas.add(conta1);
contas.add(conta2);
contas.add(conta3);
```

Temos algo diferente na criação da lista. Usamos o operador diamante "<>" para restringir a criação da lista contas para somente objetos do tipo ContaCorrente. Estamos aqui fazendo o uso dos Generics.

Quando usamos esta forma não precisamos mais fazer o cast no loop que vimos anteriormente. Sendo assim, é possível percorrer a lista da seguinte forma:

```
for(int i = 0; i < contas.size(); i++) {
    ContaCorrente conta = contas.get(i);
    System.out.println(conta.getSaldo());
}
```

Voltando ao exemplo das agências, vamos supor que precisamos listar todas as contas de uma agência. Teríamos algo parecido com isso:

```
class Agencia {
    public ArrayList<ContaCorrente> listaContas() {
        ArrayList<ContaCorrente> contas = new ArrayList<>();

        // implementação do método

        return contas;
    }
}
```

O código está correto sintática e semanticamente, mas há um detalhe: ele só vai retornar a lista na forma de um ArrayList. Conforme você for ficando mais experiente e o seu programa necessitar, pode ser que o retorno seja em outro formato, como LinkedList, por exemplo. Aí não dará certo. Neste caso, e passa a ser uma recomendação, é melhor usar a interface mais genérica possível, neste caso, List. Assim:

```
class Agencia {
    public List<ContaCorrente> listaContas() {
        ArrayList<ContaCorrente> contas = new ArrayList<>();

        // implementação do método

        return contas;
    }
}
```

O exemplo anterior contém o uso da interface List no retorno do método. Da mesma forma que fizemos no retorno, é uma boa ideia fazer na assinatura do método e em todos os lugares possíveis. Veja o exemplo da passagem de parâmetro:

```
class Agencia {
    public void verificaContas(List<ContaCorrente> contas) {
        // implementação do método
    }
}
```

É importante também declarar atributos como List ao invés de fixar em uma implementação específica. Deste modo vamos obter um baixo acoplamento e, assim, podemos trocar a implementação, pois estamos usando uma interface.

Métodos genéricos

É possível escrever uma única declaração de método genérico que pode ser chamado com argumentos de tipos diferentes. Isso é bem legal e útil! Com base nos tipos dos argumentos passados para o método genérico, o compilador trata cada chamada de método de forma adequada. Veja como fazer isso:

1. Todas as declarações de métodos genéricos usam o operador diamante que precede o do método de tipo de retorno (<E> no próximo exemplo).

2. Cada seção do tipo de parâmetro contém um ou mais parâmetros de tipo separados por vírgulas. Um tipo de parâmetro, também conhecido como uma variável de tipo, é um identificador que especifica um nome de tipo genérico.

3. Os parâmetros de tipo podem ser usados para declarar o tipo de retorno e atuam como espaços reservados para os tipos dos argumentos passados para o método genérico, que são conhecidos como argumentos de tipo reais.

4. O corpo de um método genérico é declarado como o de qualquer outro método. Importante: os parâmetros de tipo podem representar somente tipos de referência, ou seja, objetos e não tipos primitivos (como int, double e char).

Vamos entender melhor o que definimos com um exemplo usando Generics.

O exemplo é muito interessante porque mostra como podemos imprimir um array de tipos diferentes usando um único método genérico. Veja o destaque no código na linha 5 para saber como fazemos isso.

```
1   import javax.swing.*;
2
3   public class Teste {
4       // criando um metodo genérico mostraArray
5       public static <E>void mostraArray(E[] arrayEntrada) {
6           // Mostra os elementos do array
7           for(E elemento : arrayEntrada) {
8               System.out.printf("%s ", elemento);
9           }
10          System.out.println();
11      }
12
13      public static void main(Stringargs[]) {
14          // Cria um arrays de Integer, Double e Character
15          Integer[] intArray = { 1, 2, 3, 4, 5 };
16          Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
17          Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
18
19          System.out.println("integerArray:");
20          mostraArray(intArray);   // passando um array de Integer
21
22          System.out.println("\ndoubleArray:");
23          mostraArray(doubleArray);  // passando um array de Double
24
25          System.out.println("\ncharacterArray:");
26          mostraArray(charArray);   // passando um array de Char
27      }
28  }
29
```

Veja que nas linhas 7 a 9 usamos o for extendido, que é a forma mais adequada, junto com o Iterator, para percorrer as coleções.

```
O resultado da execução é mostrado a seguir:
integerArray:
1 2 3 4 5

doubleArray:
1.1 2.2 3.3 4.4

characterArray:
H E L L O
```

Classes parametrizadas

Podemos declarar uma classe como classe genérica da mesma forma que criamos uma classe normal, exceto que o nome da classe é seguido por uma seção tipo de parâmetro.

Tal como acontece com métodos genéricos, a seção tipo de parâmetro de uma classe genérica pode ter um ou mais parâmetros de tipo separados por vírgulas. Essas classes são conhecidas como classes parametrizadas ou tipos parametrizados porque eles aceitam um ou mais parâmetros.

Veja o exemplo a seguir. Observe o parâmetro <T> na declaração da classe (linha 3). E depois veja como usamos os tipos para passar os parâmetros para a classe (linhas 15 e 16).

```
1   import javax.swing.*;
2
3   public class Teste<T> {
4       private T t;
5
6       public void set(Tt) {
7           this.t = t;
8       }
9
10      public T get() {
11          return t;
12      }
13
```

```
14        public static void main(String[] args) {
15            Teste<Integer> integerBox = new Teste<Integer>();
16            Teste<String> stringBox = new Teste<String>();
17
18            integerBox.set(new Integer(10));
19            stringBox.set(new String("Hello World"));
20
21            System.out.printf("Valor        inteiro        :%d\n\n",
integerBox.get());
22            System.out.printf("Valor String :%s\n", stringBox.get());
23        }
24    }
```

Existem muitas aplicações para os Generics em Java. Na verdade, é uma boa prática usá-los. Quando você estiver estudando Estrutura de Dados você vai usar bastante este conceito e sempre que ler ou ouvir falar do assunto chamado "Padrões de Projeto" ou "Design Patterns" tenha a certeza de que terá algum relacionamento com isso.

ATIVIDADES

Vamos resolver alguns testes sobre os assuntos deste capítulo. Os dois primeiros testes são questões reais encontradas em concursos públicos. É interessante saber que essas provas também caem bastante conteúdo técnico específico.

01. (Ano: 2015 - Banca: FGV - Órgão: TCE-SE - Prova: Analista de Tecnologia da Informação-Desenvolvimento)

Um programador Java precisa utilizar em seu aplicativo uma tabela dinâmica de inteiros, cujo tamanho pode aumentar ao longo da execução. Para isso, ele decide importar a classe java.util.ArrayList e a declaração da referência à tabela deverá ser:

a) ArrayList<int> tabela;
b) ArrayList<Integer> tabela;
c) ArrayList<int>[] tabela;
d) ArrayList<int> tabela[];
e) ArrayList<Integer> tabela[].

02. Ano: 2015 Banca: FGV Órgão: PGE-RO Prova: Analista da Procuradoria - Analista de Sistemas (Desenvolvimento)

Um programador Java precisa utilizar, em seu código, um arranjo dinâmico de números inteiros. A declaração correta para esse arranjo é:

a) ArrayList<int> arranjo;
b) ArrayList<Int> arranjo;
c) ArrayList<Integer> arranjo[];
d) ArrayList<Int> arranjo[];
e) ArrayList<Integer> arranjo.

03. Escreva um método genérico para trocar as posições de dois elementos de um array.

REFLEXÃO

Este foi apenas o último capítulo de um livro preparado cuidadosamente para você. Porém, ele representa o início, ou a continuação, do seu aprendizado na área de programação de computadores. Não se prenda a este material e nas aulas. Pesquise, e muito. Você tem uma grande biblioteca, que é a internet, para consultar, conversar, participar de grupos e agregar conhecimento de várias fontes como textos, vídeos, vídeo aulas e outros. Aproveite! Não se prenda somente na linguagem Java. Usamos o Java aqui como ferramenta, e não como finalidade. Os mecanismos que você estudou aqui são válidos para outras linguagens com algumas variações é claro, mas lembre-se: é como aprender outro idioma. Selecione as suas fontes de aprendizado. Se você está estudando Java, vá direto ao local onde ela é documentada (no site da Oracle) e de lá selecione outros locais confiáveis. No mais, só desejamos sucesso e felicidades a você!

REFERÊNCIAS BIBLIOGRÁFICAS

CORNELL, G.; HORSTMANN, C. **Core Java 2:** Recursos Avançados. São Paulo: Makron Books, 2001.

CORNELL, G.; HORSTMANN, C. S. **Core Java - Vol. 1 - Fundamentos**. 8. ed. São Paulo: Pearson Education, 2010.

DEITEL, H. M.; DEITEL, P. J. **Java**: como programar. 8. ed. Rio de Janeiro: Pearson, 2010.

ECKEL, B. **Thinking in Java**. 4. ed. New Jersey: Prentice Hall, 2006.

FLANAGAN, D. **Java**: O guia essencial. 5. ed. Rio de Janeiro: Bookman, 2006.

HUBBARD, J. R. **Programação com Java**. 2. ed. Rio de Janeiro: Bookman, 2006.

SANTOS, F. G. D. **Linguagem de programação**. Rio de Janeiro: SESE, 2015.

SIERRA, K.; BATES, B. **Use a cabeça**: Java. 2. ed. Rio de Janeiro: Alta Books, 2007.

ZEROTURNAROUND. **Java Tools and Technologies Landscape for 2014**, 2014. Disponível em: <https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/6/>. Acesso em: 27 jul. 2016.

GABARITO

Capítulo 1

01.

```
    import javax.swing.JOptionPane;
    public class Exercicio {
        public static void main(String[] args){
            String entrada, saida;
            int numeroConta, saldo, totalItens, totalCreditos, limiteCredito,
novoSaldo;
            entrada = JOptionPane.showInputDialog("Digite o numero da conta");
            numeroConta = Integer.parseInt(entrada);

            entrada = JOptionPane.showInputDialog("Digite o saldo");
            saldo = Integer.parseInt(entrada);

            entrada = JOptionPane.showInputDialog("Digite o total de itens");
            totalItens = Integer.parseInt(entrada);

            entrada = JOptionPane.showInputDialog("Digite o total de creditos");
            totalCreditos = Integer.parseInt(entrada);

            entrada = JOptionPane.showInputDialog("Digite o limite");
            limiteCredito = Integer.parseInt(entrada);

            novoSaldo = saldo + totalItens - totalCreditos;

            if (novoSaldo>limiteCredito){
                JOptionPane.showMessageDialog(null, "O cliente da conta " +
numeroConta +" excedeu o limite!");
```

```
            }
            else {
                JOptionPane.showMessageDialog(null, "O cliente da conta "+nu-
meroConta+" NAO excedeu o limite!");
            }
        }
    }
```

02.

a)
```
Segunda string
Terceira string
```

b)
```
public class Exercicio {
public static void main(String[] args){
    intumNumero = 3;
    if (umNumero>= 0)
    if (umNumero == 0)
    System.out.println("Primeira string");
    else System.out.println("Segunda string");
    System.out.println("Terceira string");
}
}
```

c)
```
public class Exercicio {
    public static void main(String[] args){
        intumNumero = 3;
        if (umNumero>= 0)
        if (umNumero == 0)
            System.out.println("Primeira string");
        else
            System.out.println("Segunda string");
        System.out.println("Terceira string");
    }
}
```

d)
```
public class Exercicio {
    public static void main(String[] args){
        int umNumero = 3;
        if (umNumero>= 0){
```

```
                if (umNumero == 0)
                    System.out.println("Primeira string");
            }
            else {
                System.out.println("Segunda string");
            }
            System.out.println("Terceira string");
        }
    }
```

Capítulo 2

01. As variáveis são: horas, minutos, segundos.

02. Não há variáveis de classe na classe Hora.

03. As variáveis de instância são: nome, atraso, tempoTrabalhado, tempoAtraso.

04. A variável de classe é: totalFuncionarios

05. O tipo de associação é composição.

06. Sugestão:

```
public class controleHorario {
    public static void main(String[] args) {
        Hora horaChegada = new Hora(8, 0, 1);
        Hora horaSaida = newHora(9, 30, 0);
        Funcionario ze = new Funcionario("Ze", horaChegada, horaSaida);

        System.out.println("Hora de chegada: " + horaChegada);
        System.out.println("Hora de saída: " + horaSaida);
        System.out.printf("Horas        trabalhadas:        %.1f\n",ze.
getHorasTrabalhadas());
    }
}
```

Capítulo 3

01. De uma maneira bem simples, a herança pode ser definida como o processo no qual uma classe adquire as propriedades (métodos e campos) de outra classe. Com o uso da herança, as informações são gerenciadas de uma maneira hierárquica.

02. Usamos o super quando um método substitui um dos métodos da sua superclasse e, assim, podemos invocar o método substituído.

03. Polimorfismo é a capacidade de um objeto assumir muitas formas. O uso mais comum de polimorfismo na orientação a objetos ocorre quando uma referência da superclasse pai é usada para se referir a um objeto da classe filho.

04. Uma classe abstrata é um tipo de classe que não podem ser instanciada, mas podem ser superclasse de outras.

05. Sugestão de implementação:

```
public class Ingresso {
    private double valorIngresso;

    public Ingresso(double valor) {
        this.valorIngresso = valor;
    }

    public double getValorIngresso() {
        return valorIngresso;
    }

    public void setValor Ingresso(doublevalor) {
        this.valorIngresso = valor;
    }

    public void imprimeValor(){
        System.out.println("Valor do ingresso R$"+valorIngresso);
    }
}
```

```
    public class VIP extends Ingresso {
        private doublevalor Adicional;

        public VIP(doublevalor, doublevalor Adicional) {
            super(valor);
            this.valorAdicional = valorAdicional;
        }

        public void imprimeValor(){
            doublevalorVIP          =          this.valorAdicional          +
super.getValorIngresso();
            System.out.println("Valor VIP R$"+ valorVIP);
        }
    }

    public class IngressoNormalextends Ingresso {
        private double valorNormal;

        public IngressoNormal(double valor){
            super(valor);
            this.valorNormal = valor;
        }

        public void imprimeValor(){
            System.out.println("Ingresso normal R$"+valorNormal);
        }
    }

    public class CamaroteSuperior extends VIP {
        privatedoubletaxaSuperior;
        privateStringlocalizacaoIngresso;
```

```
        public CamaroteSuperior(doub levalor, double valorAdicional, double
taxaSuperior, String local) {
            super(valor, valorAdicional);
            this.taxaSuperior = taxaSuperior;
            this.localizacaoIngresso = local;
        }

        public void setLocalizacao (String localizacao){
            localizacaoIngresso = localizacao;
        }

        public String getLocalizacao() {
            return"Localização: " + localizacaoIngresso;
        }

        public String camaroteSuperior() {
            return"Valor  Camarote  superior:  R$"  +  super.getValorIngres-
so()+this.taxaSuperior;
        }
    }

    public class CamaroteInferior extends VIP {
        private String localizacaoIngresso;

        public Camarote Inferior(double valor, doublevalorAdicional, String
local) {
            super(valor, valorAdicional);
            this.localizacaoIngresso = local;
        }

        public void setLocalizacao(String localizacao){
            localizacaoIngresso = localizacao;
        }
```

```
    public String getLocalizacao() {
        return"Localização do Ingresso: " + localizacaoIngresso;
    }
}
```

Capítulo 4

01.

Sim, o código está correto e é bem útil. Um comando try não tem, obrigatoriamente, de ter um catch se ele possui um bloco finally.

02.

Esse tratador captura exceções do tipo Exception; portanto, ele pega qualquer tipo de exceção.

O problema é que usar este tipo de tratador faz com que seu programa fique prejudicado, pois ele perde a capacidade de tratar exceções específicas, como resultado, o programa pode ser forçado a determinar o tipo de exceção antes que possa decidir sobre a melhor estratégia de recuperação do erro.

03.

O primeiro tratador captura exceções do tipo Exception; portanto, ele pega qualquer exceção, incluindo ArithmeticException. O segundo tratador nunca será usado.

Este código não irá compilar.

04.

a) 3 (erro de compilação). O array não é inicializado e não compilará.
b) 1 (erro).
c) 4 (semexceção). Quando um fluxo é lido, é esperado que ele tenha um marcador de fluxo. Você pode usar exceções para tratar comportamentos inesperados no seu programa.
d) 2 (Exceção verificada (checked).

Capítulo 5

01. B

02. E

03. Sugestão:

```
public final class Teste {
    public static <T> void troca(T[] a, int i, int j) {
        T temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

ANOTAÇÕES

ANOTAÇÕES

ANOTAÇÕES

ANOTAÇÕES