UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO (UFRPE)

COORDENAÇÃO GERAL DE EDUCAÇÃO A DISTÂNCIA (EAD/UFRPE)

Programação II

Fernando Antonio Mota Trinta

VOLUME 2

Recife, 2010

Universidade Federal Rural de Pernambuco

Reitor: Prof. Valmar Corrêa de Andrade
Vice-Reitor: Prof. Reginaldo Barros
Pró-Reitor de Administração: Prof. Francisco Fernando Ramos Carvalho
Pró-Reitor de Extensão: Prof. Paulo Donizeti Siepierski
Pró-Reitor de Pesquisa e Pós-Graduação: Prof. Fernando José Freire
Pró-Reitor de Planejamento: Prof. Rinaldo Luiz Caraciolo Ferreira
Pró-Reitora de Ensino de Graduação: Profª. Maria José de Sena
Coordenação Geral de Ensino a Distância: Profª Marizete Silva Santos

Produção Gráfica e Editorial
Capa e Editoração: Rafael Lira, Italo Amorim e Gláucia Fagundes
Revisão Ortográfica: Marcelo Melo
Ilustrações: Allyson Vila Nova
Coordenação de Produção: Marizete Silva Santos

Sumário

Apresentação

Caro(a) cursista,

Seja bem-vindo(a) ao curso de **Programação II**. Este curso é composto por quatro volumes. O primeiro volume foi dedicado à teoria e aos principais conceitos relacionados ao paradigma de programação baseado em objetos. Foram apresentadas as vantagens do paradigma OO em relação a outros paradigmas de programação, além de uma introdução à linguagem de programação Java.

Este segundo volume será dedicado em sua totalidade à sintaxe Java, seu sistema de tipos, operadores lógicos, comandos de decisão e repetição. No terceiro volume, você aprenderá a utilizar os conceitos de orientação a objetos utilizando a sintaxe apresentada neste volume.

Por fim, no quarto e último volume serão abordados assuntos ainda mais avançados, como a ideia de herança, polimorfismo e tratamento de exceções.

Bons estudos!

Fernando Trinta
Autor

CONHECENDO O VOLUME 2

Neste segundo volume, você irá encontrar o Módulo 2 da disciplina **Programação II.** Para facilitar seus estudos, veja a organização deste módulo.

Módulo 2 – Introdução à Linguagem de Programação Java

Carga Horária do Módulo 2: 15 h

Objetivo do Módulo 2: Apresentar a sintaxe da linguagem Java, seu sistema de tipos e principais construções.

Conteúdo Programático do Módulo 2

- Instruções e Expressões. Declaração de Variáveis. Tipos Primitivos em Java. Comentários. Literais. Expressões e Operadores Matemáticos. Incremento/ Decremento;
- Operadores Relacionais. Operadores Lógicos. Operadores condicionais. if e switch. Blocos. Operadores de repetição: Laços for. Laços while...do. Laços do...while. Interrupção de Loops (breaks / rótulos);
- Vetores e Matrizes. Entrada de dados pelo teclado.

CAPÍTULO 1

O que vamos estudar neste capítulo?

Neste capítulo, vamos estudar os seguintes temas:

» Introdução à Linguagem de Programação Java.

Metas

Após o estudo deste capítulo, esperamos que você consiga:

» Conhecer os principais tipos de dados disponíveis em Java;

» Aprender sobre a diferença entre tipos primitivos e objetos, especialmente a classe String;

» Conhecer como se constroem expressões em Java;

» Conhecer os operadores matemáticos.

CAPÍTULO 1 – CONCEITOS BÁSICOS DA LINGUAGEM JAVA

Vamos conversar sobre o assunto?

Caro cursista, toda linguagem de programação tem por base a definição de instruções e a manipulação de variáveis que armazenam temporariamente dados. Estes dados são classificados de acordo com o tipo de informação que a variável pode armazenar, como números inteiros, caracteres ou booleanos. Vamos ver como Java utiliza esta ideia para definir variáveis em seus programas, assim como operadores matemáticos que possibilitam a realização de somas, subtrações, dentre outras operações. Além disso, vamos ver alguns conceitos importantes que auxiliam a construção de sistemas em Java, como comentários e precedência de operadores.

1.1 INTRODUÇÃO

Java é uma linguagem de programação que utiliza conceitos comuns a outras linguagens previamente definidas. Boa parte dessas construções pega emprestada a sintaxe do C e C++, o que de certo modo facilita a migração destes programadores para Java. Como toda linguagem de programação, Java precisa indicar as instruções em seus programas, permitir a escrita de comentários, dentre outras possibilidades. Instruções manipulam dados, retornam valores, calculam expressões. Dados por sua vez possuem tipos, etc. Vamos então neste capítulo, ver como estes conceitos são construídos em Java.

1.2 COMENTÁRIOS

Comentários são importantes em qualquer linguagem de programação. Java define três tipos de comentários. Uma primeira possibilidade é o uso das sequências `/*` e `*/` para limitar os comentários de um programa, como no exemplo da listagem abaixo.

```
1   /*
2     Este programa serve para mostrar uma mensagem na tela do meu computador.
4     A mensagem que aparece é Oi Mundo!!!
5   */
6   public class AloMundo {
7       public static void main(String args[]){
8               System.out.println("Oi Mundo!!!"); //Este comando imprime na tela
9               int x = 123L;
10      }
11  }
```

O trecho entre as linhas 1 e 5 é um comentário com múltiplas linhas. Comentários como este não podem ser aninhados, ou seja, não pode haver um comentário dentro de outro comentário.

Neste mesmo exemplo, temos outro tipo de comentário. Na linha 8, as barras duplas (//) podem ser utilizadas para um comentário de uma única linha. Com isso, o texto do início do comentário até o final é ignorado pelo processo de compilação.

O último tipo de comentário é uma particularidade de Java. Estes comentários são associados à documentação da classe, e são chamados de comentários Javadoc. Eles se iniciam com /** e terminam com */. Nós falaremos mais sobre este tipo de comentário no próximo módulo.

1.3 Instruções

Uma instrução realiza uma operação em Java e deve ser sempre finalizada com ponto-e-vírgula(;). As instruções podem ser colocadas em uma mesma linha, desde que preservada a separação com ponto-e-vírgula.

Em muitos casos, instruções são tratadas em conjunto, como um bloco de instruções. Neste caso, as instruções que fazem parte de um mesmo bloco de instruções devem ser envolvidas por chaves ({}). Nós já vimos isso em nosso primeiro exemplo de um programa Java, no capítulo 3 do primeiro volume deste curso. Na classe `AloMundo` (Figura 1), o método `main()` é um bloco de instruções, que neste caso, só possui uma instrução - `System.out.println("Oi Mundo!!!").`

```
public static void main(String args[]){
    System.out.println("Oi Mundo!!!");
}
```

Figura 1 - Método main() da classe AloMundo

1.4 Variáveis e Tipos de Dados

Como em toda linguagem de programação, um conceito comum é o de variável. Uma variável é um local de memória que armazena temporariamente um valor. Em Java, variáveis podem armazenar endereços de memória ou valores atômicos de tamanho fixo. Endereços de memória são utilizados como referências entre objetos e são inacessíveis aos programadores, pois Java não suporta aritmética de ponteiros. Em outras palavras, você não pode acessar diretamente uma posição de memória específica, mas apenas por meio dos objetos que você cria em seus programas. Já valores atômicos representam tipos de dados primitivos.

Em geral, uma variável possui um tipo, um nome e um valor, sendo que em Java, uma variável precisa ser declarada (e possivelmente inicializada) antes de ser utilizada no programa. Uma das boas características de Java é que ela é uma linguagem fortemente tipada. Em outras palavras, é obrigatório que se forneça o tipo da variável antes de sua utilização. Essa característica também indica que uma vez definido um tipo, uma variável só pode armazenar valores compatíveis com este tipo, a não ser que haja alguma forma de conversão.

A declaração de uma variável em Java segue a seguinte sintaxe:

```
<tipo>  <identificador>  [=  <valor>];
```

Onde o trecho entre colchetes "`[]`" é opcional e representa a atribuição de um valor à variável no momento de sua definição. Embora se utilize o sinal "=", isto não representa uma comparação de valores. A comparação utiliza-se o operador "==" e será visto adiante.

Em relação ao tipo, este representa um valor, uma coleção de valores ou uma coleção de outros tipos. No caso, tipos também podem ser classificados como:

- **básicos ou primitivos**: quando representam unidades indivisíveis de informação de tamanho fixo;
- **complexos**: quando representam informações que podem ser decompostas em tipos menores. Os tipos menores podem ser primitivos ou outros tipos complexos

Em outras palavras, na declaração de uma variável Java, o tipo de uma variável pode ser definido como:

- Um dos 8 tipos de dados básicos (também chamados tipos primitivos)
- Uma classe, ou seja, um tipo complexo;
- Um vetor, ou seja, uma coleção de outros tipos;

O uso de vetores será visto adiante em nossa disciplina, assim como detalharemos melhor o uso de classes e objetos. Por enquanto, nos concentraremos nos tipos básicos, e também abordaremos uma classe em especial, a classe String, devido à sua importância na linguagem.

1.4.1 Tipos Primitivos

Os oito tipos primitivos Java representam os tradicionais aqueles já conhecidos em outras linguagens de programação, e que lidam com a manipulação de números inteiros, de ponto flutuante, caracteres e valores lógicos. A tabela apresenta estes tipos e suas características.

Tabela 1 – Tipos Primitivos em Java

Tipo	Descrição	Tamanho	Intervalo de valores válidos
`boolean`	Utilizado para representar valores lógicos	8 bits(*)	Verdadeiro ou Falso (true ou false)
`byte`	Utilizado para representar valores numéricos inteiros	8 bits	-128 a 127
`char`	Utilizados para representar caracteres	16 bits	Caracteres UNICODE
`short`	Utilizado para representar valores numéricos inteiros	16 bits	-32.768 a 32.767
`int`	Utilizado para representar valores numéricos inteiros	32 bits	-2.147.483.648 a 2.147.483.647
`long`	Utilizado para representar valores numéricos inteiros	64 bits	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
`float`	Utilizado para representar valores numéricos de ponto flutuante	32 bits	+/-3.4E-38 a +/-3.4E+38
`double`	Utilizado para representar valores numéricos de ponto flutuante	64 bits	+/-1.7E-308 a +/-1.7E+308

Como apresentado na tabela, existem quatro tipos básicos em Java para trabalhar com números inteiros. Todos podem trabalhar com números positivos ou negativos. A escolha de qual tipo usar depende dos valores que a variável vai armazenar. Se for atribuído um valor maior do que o tipo da variável, ele é truncado.

Em relação ao nome de variáveis, estas devem obrigatoriamente iniciar com uma letra, um sublinhado (_) ou um cifrão ($). Após o primeiro caractere, podem ser utilizados qualquer letra ou número. Em geral, variáveis possuem nomes significativos que por vezes são compostos por mais de uma palavra. Por convenção (não é obrigatório, mas altamente recomendável), Java segue a padronização que a primeira palavra seja grafada com letras minúsculas, e as demais tenham apenas a primeira letra grafada em maiúsculo. São exemplos de definição de variáveis:

```
int numero = 10;
double _salario$ = 1234.56;
boolean temFilhos = true;
char letraPreferida = 'F';
```

Java faz diferenciação entre maiúsculas/minúsculas. No trecho de código abaixo, temos três variáveis distintas, inclusive de tipos distintos.

```
int exemplo;
float Exemplo;
boolean EXEMPLO;
```

Embora não seja obrigatório, é uma boa prática sempre atribuir valores às variáveis

antes de utilizá-las. Porém, caso isso não seja feito, as variáveis são inicializadas com valores padrão, de acordo com o seu tipo: 0 (zero) para variáveis numéricas, '\0' para caracteres e `false` para booleanos.

Em relação à definição de variáveis, é possível também agrupar a definição de variáveis de um mesmo tipo, podendo inclusive estipular valores iniciais a cada uma delas. Veja o exemplo:

```
int x, y, z = 0;

boolean pago, vendido = true;
```

1.4.2 Palavras Reservadas

Além de obedecer às regras definidas para identificadores, Java também impõe que certas palavras reservadas não possam ser utilizadas pelo programador para nome de variáveis ou classes. A lista destas palavras é apresentada a seguir.

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

É interessante notar que existem palavras que não são usadas pela linguagem, como `const` ou `goto`, mas mesmo assim são reservadas e que portanto, você não pode usá-las como identificadores em seus programas.

1.4.3 A Classe String

Como você pode ter observado, Java não apresenta um tipo básico para trabalhar com sequências de caracteres. Java trata estes dados com objetos. Embora nós só venhamos a trabalhar mais fortemente com o conceitos de objetos no próximo módulo, nós vamos falar aqui sobre uma classe muito utilizada em Java: a classe String. Esta classe é utilizada para criar objetos que representam dados textuais, comumente chamados de strings. Por exemplo, a listagem abaixo cria algumas variáveis string.

```
String nome = "Fernando";

String sobreNome = "Trinta";

String stringVazia = "";
```

Ao contrário de linguagens de como C ou C++, strings em Java não são um vetor de caracteres, embora seja possível realizar operações parecidas, como obter a quantidade de caracteres ou saber a posição de um caractere na string. Mas a principal diferença é o

fato de cada variável que representa uma string, na realidade ser uma referência para um objeto. E isto é bem diferente de guardar um valor primitivo. Vamos dar uma olhada nas principais diferenças.

Em Java, existem duas estruturas que armazenam os dados que são referenciados por variáveis: a *heap* e uma pilha de dados. A *heap* é uma estrutura em memória que serve exclusivamente para guardar os objetos que um programa Java cria e manipula. O acesso a cada objeto é feito por meio de uma referência, que mais diretamente é representada por um identificador, uma variável. Valores de tipos primitivos não são armazenados na *heap*, mas sim, na pilha, principalmente por questões de eficiência de acesso. Vamos tentar visualizar esta diferença através de um exemplo, em três momentos distintos de sua execução. Veja este trecho de código:

```
public class AloMundo {

    public static void main(String args[]){

        int numero = 123, x = 10;

        String nome = "fernando";

        // Primeira parada


        int numero2 = 123;

        String nome2 = "fernando";

        // Segunda parada


        nome = nome2;

        // Terceira parada

    }
}
```

Vamos agora tentar visualizar a memória da máquina virtual Java logo após a declaração do objeto nome, conforme Figura 2.

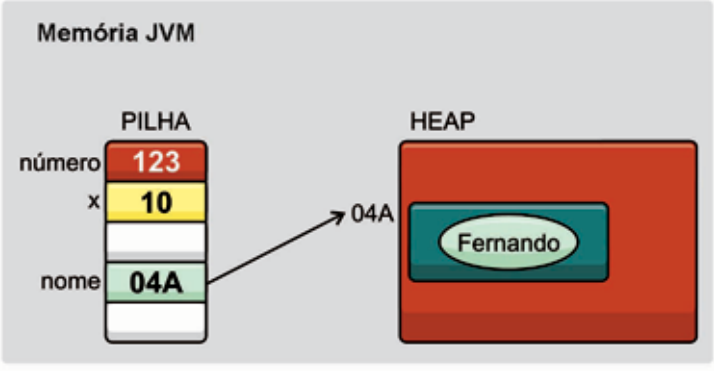

Figura 2 - Ilustração da memória usando a pilha e a heap. (Momento 1)

A pilha guarda os valores das variáveis e referências (posições de memória) para objetos que estão na *Heap*, como no caso, a posição `04A`. Por isso, diz-se que o identificador `nome` é uma referência para o objeto. Vamos ver como seria a disposição do nosso esquema de memória após a definição das variáveis `nome2` e `numero2`.

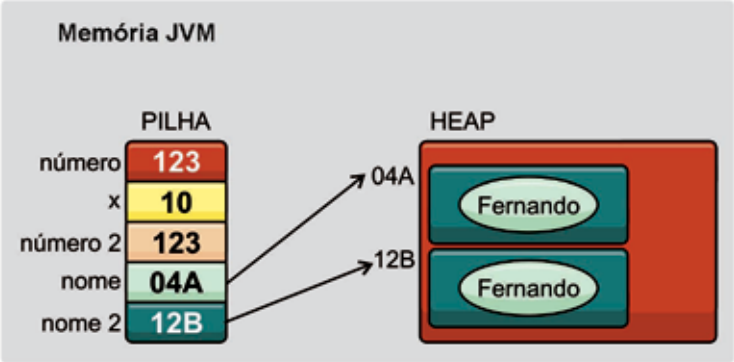

Figura 3 - Ilustração da memória usando a pilha e a heap. (Momento 2)

Note que, apesar de tanto o identificador nome quanto `nome2` guardarem a mesma string ("`fernando`"), são criados dois objetos distintos para cada um deles, em posições de memória diferentes. Veja o que acontece quando o programa realiza a operação `nome = nome2` (Figura 4).

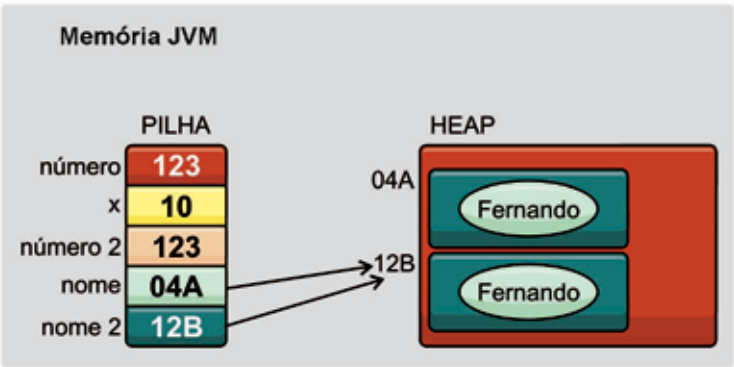

Figura 4 - Ilustração da memória usando a pilha e a heap. (Momento 3)

Neste caso, a semântica (ou significado) é temos agora dois identificadores apontando para um mesmo objeto. O objeto inicialmente "apontado" pelo identificador nome é "perdido". Isso acontece porque Java não permite acesso direto à uma posição de memória. Este objeto será mais tarde descartado e nós veremos este processo chamado de Coleta de Lixo – *Garbage Collector*. Por enquanto é importante que você aprenda que em Java, caso você queira guardar um dado alfanumérico, você fará uso de um objeto, e este objeto será do tipo `String`.

1.5 Literais

Literal é um termo comum em linguagens de programação que, essencialmente, significa "o que você escreve é o que você tem". Literais são utilizados para indicar valores simples que podem ser atribuídos às variáveis. Existem literais associados aos diferentes tipos em Java. Vamos dar uma olhada em cada um deles.

1.5.1 Literais de Número

Existem diversos literais para lidar com números inteiros. Por exemplo, 12 é um literal do tipo `int`. Todo número inteiro é por padronização do tipo `int`, mas pode

ser atribuído a variáveis dos tipos `byte` ou `short`, desde que seu valor seja pequeno o suficiente para se enquadrar no intervalo de valores destes tipos. Se um literal inteiro está fora do intervalo definido para um `int`, ele é automaticamente convertido para um `long`. Por exemplo, as seguintes declarações são possíveis:

```
int a = 1232323;

short b = 240;

byte c = 65;

long d = 9.223.372.036.854.775.809;
```

Porém, em certas situações, você precisará tratar números inteiros menores como do tipo long. Para tornar um número pequeno um `long`, você anexa um `L` ou `l` a este número. Por exemplo, `123L` é tratado com um long.

Literais inteiros, por padrão, são tratados na base decimal. Porém, também podem ser expressos na base octal ou hexadecimal. Um inteiro que inicie com 0(zero) indica que o número inteiro está na base octal – por exemplo, 087 ou 0333. Um inteiro que inicie com 0x ou 0X indica que este é um hexadecimal – por exemplo, 0xFF ou 0xAE12.

Já os literais de ponto flutuante são aqueles constituídos de uma parte inteira e outra decimal – por exemplo, 3.141516 ou 1.5. Por padrão, todo literal de ponto flutuante é do tipo double, independente da precisão ou de seu valor. Para tratar um literal de ponto flutuante como um float é necessário anexar a letra f (ou F) ao final do número – por exemplo, 1.45F. É possível ainda usar expoentes em literais de ponto flutuante, usando a letra e ou e seguida do expoente, como nos exemplos abaixo:

```
double num1 = 1.45e34;  // equivale a 1.45 x 10 elevado a 34

float num2 = 0.22e-2F  // equivale a 0.22 x 10 elevado a -2
```

1.5.2 Literais Booleanos

Literais para valores lógicos são bem simples e consistem nas palavras-chaves true e false para representar, respectivamente, os valores lógicos: verdadeiro e falso.

1.5.3 Literais de Caracteres

Consistem de um caracter simples. Para representá-los, basta escrever o caracter entre apóstrofos (aspas simples) – por exemplo, `'f'`, `'@'`, `'3'`. Estes caracteres são armazenados como caracteres Unicode de 16 bits. Literais de caracteres também podem representar caracteres especiais não imprimíveis, apresentados na Tabela 2.

Tabela 2 – Caracteres Unicode em Java

Escape	Significado
\n	Nova linha
\t	Tabulação
\b	Retroceder
\r	Retorno de carro
\f	Alimentar folha
\\	Barra Invertida
\´	Apóstrofo
\”	Aspas Duplas
\ddd	Octal
\xdd	Hexadecimal
\udddd	Caracter Unicode

O padrão Unicode permite a representação e manipulação consistente de dados textuais em qualquer sistema de escrita existente. Por exemplo, o código 22823 representa o caractere chinês 大.

1.5.4 Literais de Strings

Literais de strings constituem-se de uma sequência de caracteres entre aspas duplas. Esta cadeia de caracteres pode também conter constantes como nova linha, tabulação e outros caracteres Unicode. Vejam os exemplos na Figura 5.

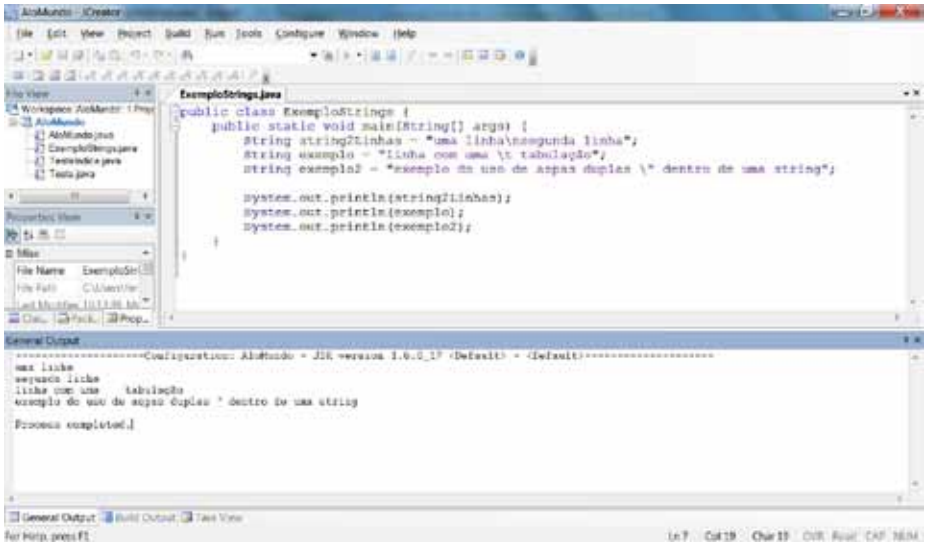

Figura 5 - Exemplo de literais de Strings

Além de uma sequência de caracteres, uma string tem como literal a palavra reservada `null`. Na realidade, a palavra `null` está associada a qualquer tipo de objetos e não exclusivamente àqueles do tipo `String`. Um objeto `String` com referência nula significa que apesar de uma variável ter sido criada, ela não aponta para nenhum objeto válido. Veja no exemplo do código da Figura 6.

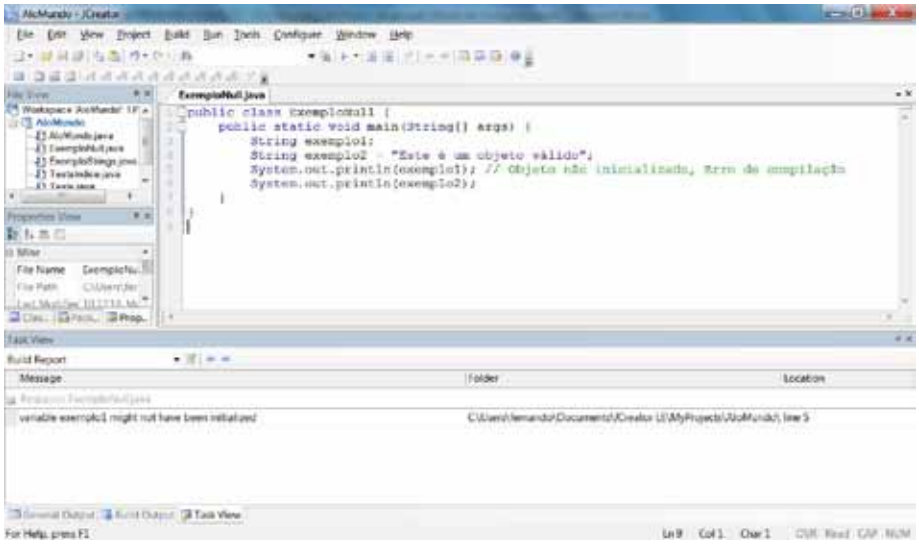

Figura 6 - Exemplo de uso de objetos não inicializados

Neste exemplo existem duas variáveis do tipo String, exemplo1 e exemplo2. Porém, apenas a segunda referencia um objeto válido. A segunda não "aponta" para ninguém. Desta forma, diz-se comumente em Java que exemplo1 é uma "referência nula" ou aponta para null. Java não permite que um objeto seja utilizado sem antes apontar para algum valor. Com isso, o programa da Figura 6 apresenta um erro de compilação na linha 5, pois o objeto exemplo1 não foi inicializado antes de seu uso.

Porém, é possível definir um identificador de um objeto e fazer com que o mesmo aponte para uma referência nula. Para isto, basta atribuir ao identificador o valor `null`. Veja como o programa `ExemploNull` (Figura 7) compila sem problemas modificando a definição do objeto `exemplo1`, e apresenta o literal `null` quando pedido que o valor do identificador seja apresentado na tela (linha 5).

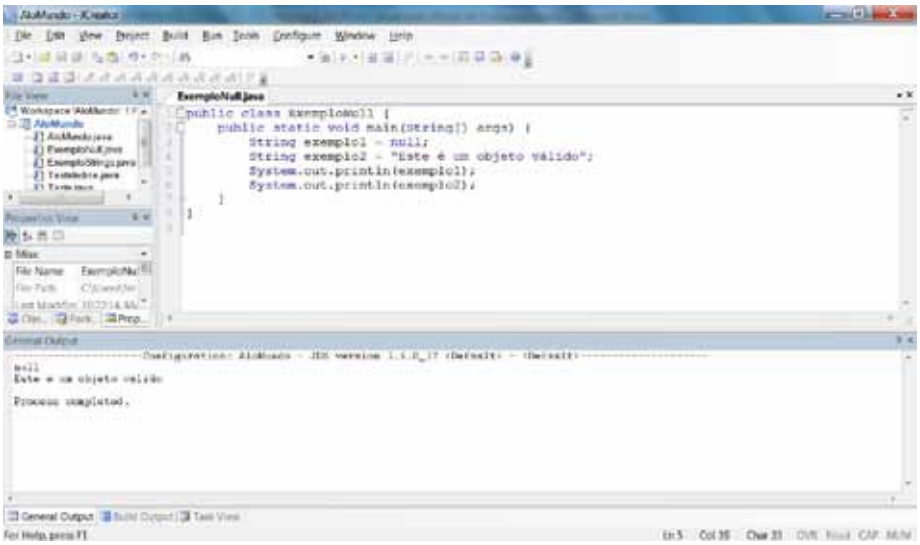

Figura 7 – Inicializando um objeto String com o valor nulo

1.6 Expressões e Operadores

Expressões representam uma forma de manipulação de dados a partir de instruções realizadas por determinados operadores. Em geral, expressões devolvem um valor associado

à sua realização.

A realização das expressões envolvem cálculos matemáticos, testes de igualdade ou grandeza, dentre outras operações. Pelo fato de devolver um valor, expressões podem ser associadas a variáveis.

As operações realizadas nas expressões são determinadas por meio de operadores. A maioria dos operadores presentes em Java são os mesmos encontrados em outras linguagens de programação, como adição, subtração, maior que, dentre outros. Estes operadores produzem um novo valor a partir de um ou mais argumentos.

A seguir vamos ver os principais operadores oferecidos por Java.

1.6.1 Operadores Aritméticos

Java oferece cinco operadores básicos para cálculos aritméticos, conforme tabela abaixo

Tabela 3 - Operadores Matemáticos em Java

Operador	Descrição	Exemplo	Resultado
+	Adição	3 + 5	8
-	Subtração	3 – 5	-2
*	Multiplicação	10 *4	40
/	Divisão	21 / 3	7
%	Módulo da divisão inteira	20 % 3	2

Cada um dos operadores listados na tabela recebe dois operandos, sendo que o operador subtração pode ser usado para negar um operando simples. Veja o exemplo:

```
int x = 10;
int y = -x; // y = -10;
```

Os operadores podem trabalhar com qualquer tipo numérico. Porém é necessário saber algumas propriedades destes operadores. Primeiro, na divisão entre inteiros, o resultado da operação será sempre um inteiro. Se houver fração decimal, esta será ignorada. Portanto, ao dividir 9 por 4, o resultado será 2. Para se obter o valor correto, deve-se trabalhar com tipos fracionários, como `double` ou `float`, como visto na Figura 8.

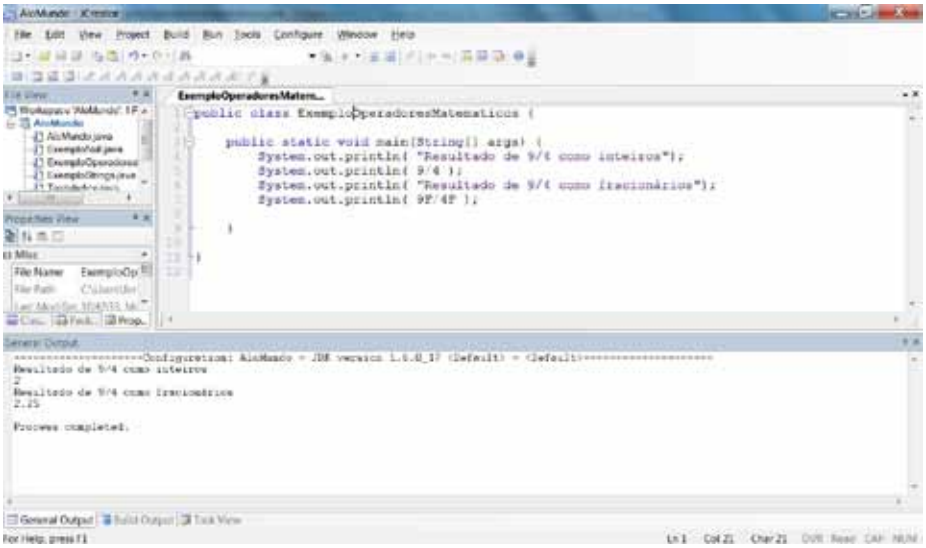

Figura 8 - Divisão por Inteiros e Divisão por Números Fracionários

1.6.1.1 Ainda sobre a Atribuição...

Ao realizar uma atribuição, o valor da direita é copiado para o valor da esquerda do operador "=". Com isso, variáveis podem ser usadas nos dois lados de uma atribuição. Por exemplo:

```
int x = 10; //copia para x o literal inteiro 10
int z = x; // copia o valor contido em x para z
```

As atribuições podem ser também agrupadas, como no exemplo abaixo:

```
int x, z;
x = z = 13;
```

Neste exemplo, o valor 13 é copiado para as variáveis x e z.

Em uma atribuição, a expressão do lado direito é sempre a primeira a ser avaliada. Com isso, é possível realizar em Java expressões como:

```
int x = 10;
x = x + 2; // x é somado ao valor constante 2 e depois atribuído à variável x
```

Na realidade, este tipo de operação é tão comum que Java utiliza operadores simplificados para estas operações, de forma semelhante à C e C++. A tabela mostra estas operações.

Tabela 4 - Operadores simplificados com atribuição

Expressão	Significado
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y

1.6.2 Incremento e Decremento

Java também oferece operadores utilizados para incrementar e decrementar o valor de uma variável de 1. Estes operadores são `++` e `--`. Por exemplo, `x++` equivale à expressão `x = x + 1`.

Na realidade, os operadores podem ser prefixados ou sufixados. Em outras palavras, os operadores podem aparecem antes ou depois da variável a ser modificada. Em operações simples, não há diferença no resultado destas operações. Mas quando o incremento está envolvido em uma atribuição, os resultados são distintos. Veja o exemplo a seguir.

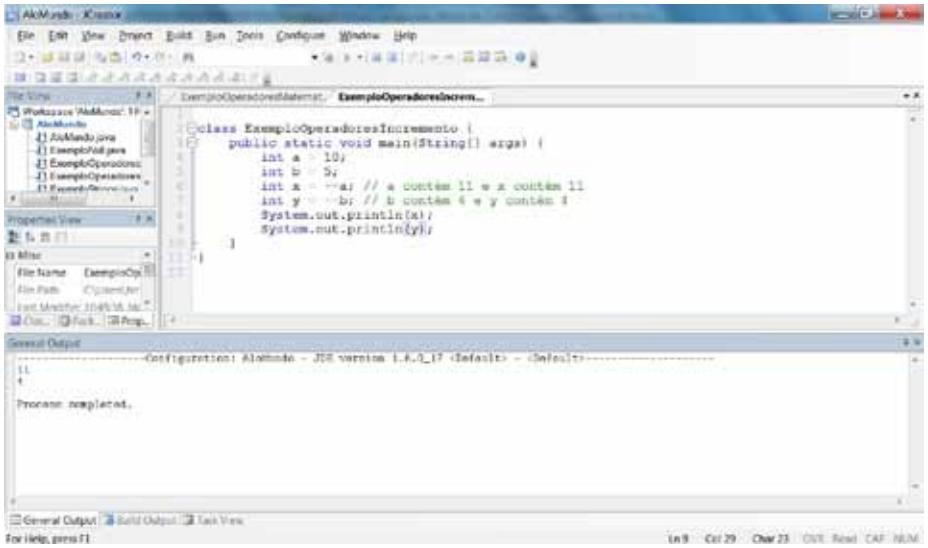

Figura 9 - Exemplo de uso de operadores de incremento pré-fixados

Neste exemplo, as operações de incremento e decremento são realizadas antes das atribuições. Mas vejamos o resultado do mesmo programa, modificando a posição dos operadores.

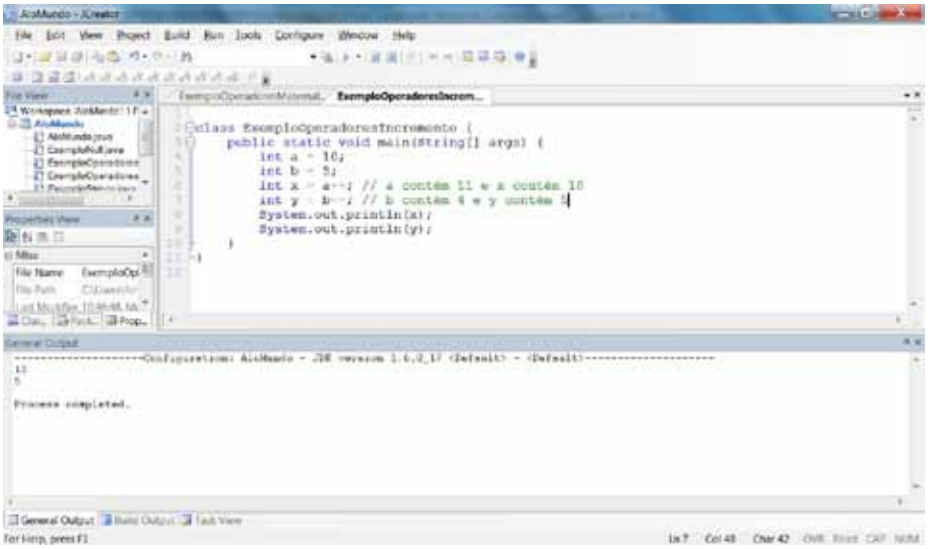

Figura 10 - Exemplo de uso de operadores de incremento pós-fixados

Neste último caso (Figura 10), as operações de incremento e decremento só são realizadas após a atribuição. Portanto, `x` e `y` recebem primeiro os valores de `a` e `b`, antes que estes realizem o incremento ou decremento destes valores.

1.6.3 Precedência de Operadores

A precedência entre operadores determina em que ordem as operações de uma expressão serão realizadas. Existe uma precedência implícita entre os operadores Java. Por exemplo, o operador de multiplicação deve ser executado antes de operações de adição. Por exemplo, para a expressão abaixo, o valor é calculado utilizando as regras implícitas de precedência:

```
int x = 2 + 2 * 3 - 9 / 3; // 2 + 6 - 3 = 5
```

Para mudar a precedência entre operadores, o programador pode utilizar parênteses para sobrepor a precedência implícita. Veja como o uso de parênteses altera o valor final da expressão anterior.

```
int x = (2 + 2) * (3 - 9) / 3; // 4*(- 6)/3 = - 8
```

A tabela abaixo mostra a ordem completa de precedência entre operadores em Java. Os operadores iniciais são os de maior precedência, ou seja, são executados primeiro.

Tabela 5 - Tabela de Precedência em Java

Operador	Descrição
`. [] () (tipo)`	Máxima precedência: separador, indexação, parâmetros, conversão de tipo
`+ - ~ ! ++ --`	Operador unário: positivo, negativo, negação (inversão bit a bit), não (lógico), incremento, decremento
`* / %`	Multiplicação, divisão e módulo (inteiros)
`+ -`	Adição, subtração
`<< >> >>>`	Translação (bit a bit) à esquerda, direita sinalizada, e direita não sinalizada (o bit de sinal será 0)
`< <= >= <`	Operador relacional: menor, menor ou igual, maior ou igual, maior
`== !=`	Igualdade: igual, diferente
`&`	Operador lógico e bit a bit
`^`	Ou exclusivo (xor) bit a bit
`|`	Operador lógico ou bit a bit
`&&`	Operador lógico e condicional
`||`	Operador lógico ou condicional
`?:`	Condicional: if-then-else compacto
`= op=`	Atribuição

Quando as expressões utilizam expressões de mesma precedência, estas são calculadas da esquerda para a direita. Por exemplo:

```
int y = 13 + 2 + 4 + 6; // Equivale a (((13 + 2) + 4) + 6)
```

1.7 Conversão de Tipos Primitivos

Java trabalha com a conversão automática de tipos primitivos sempre que isto for necessário. As conversões ocorrerão automaticamente quando houver garantia de não haver perda de informação. Essas situação ocorrem de:

- Tipos menores em tipos maiores;
- Tipos de menor precisão em tipos de maior precisão;
- Inteiros em ponto-flutuante.

Veja o exemplo a seguir (Figura 11), onde tivéssemos sete variáveis, uma para cada tipo primitivo.

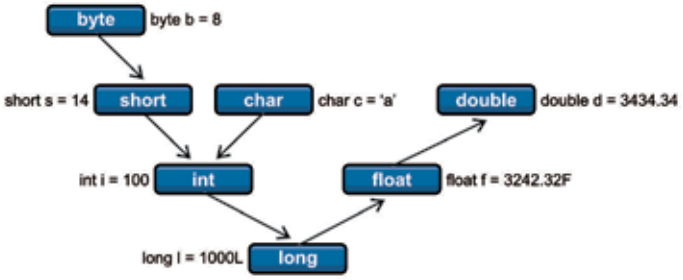

Figura 11 - Conversão automática em Java

De acordo com a Figura 11, seriam possíveis as seguintes conversões:

```
int s2 = b;     // byte sendo convertido para int, pois int > byte
int i2 = c;     // char sendo convertido para int, pois int > char
int i3 = s;     // byte sendo convertido para int, pois int > short
long l2 = i;    // int sendo convertido para long, pois long > int
float f2 = l;   // long sendo convertido para float, pois float > long
double d2 = f; // float sendo convertido para double, pois double > float
```

Ao se misturar valores de diferentes tipos primitivos, o tipo do resultado final será dependente dos tipos dos valores envolvidos. Nestes casos há uma conversão automática, chamada então de *promoção*. Na prática, a promoção de tipos ocorre para:

- o maior ou mais preciso tipo da expressão (até `double`)
- o tipo `int` para tipos menores que `int`;

Vamos explicar esta promoção por meio de um exemplo:

```
12L + 8 * 34.3
```

Neste caso, `12L` é um `long`, `8` é um `int` e `34.3` é um `double`. Desta forma o resultado geral desta expressão é do tipo `double`, pois este é o tipo mais preciso entre os envolvidos na expressão. Logo, se você tentar atribuir esta expressão a uma variável que não seja um `double`, você obterá um erro de compilação, como ilustrado na Figura 12.

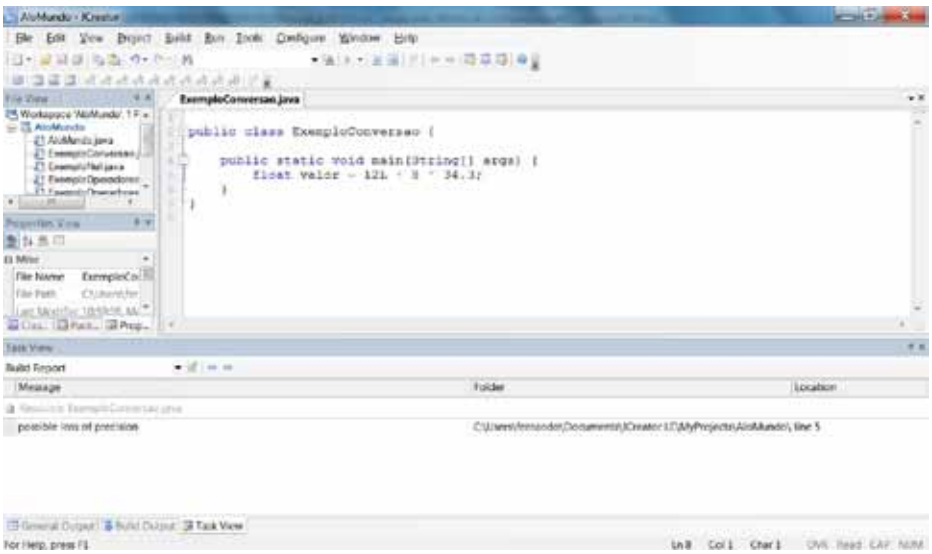

Figura 12 - Exemplo de uma conversão automática

1.7.1 Coerção entre Tipos Primitivos

A promoção entre tipos é uma tarefa realizada automaticamente pelo compilador Java para garantir a compatibilidade entre os valores de uma expressão. Porém, em muitos casos, pode-se querer fazer justamente o inverso. No caso, atribuir o valor de uma variável de um determinado grau de precisão para outro de menor precisão.

Isso é possível em Java por meio de uma operação denominada coerção ou *casting*. A ideia da coerção é deixar a cargo do programador a definição do melhor tipo de dado utilizado para guardar um valor. Veja o exemplo da Figura 13.

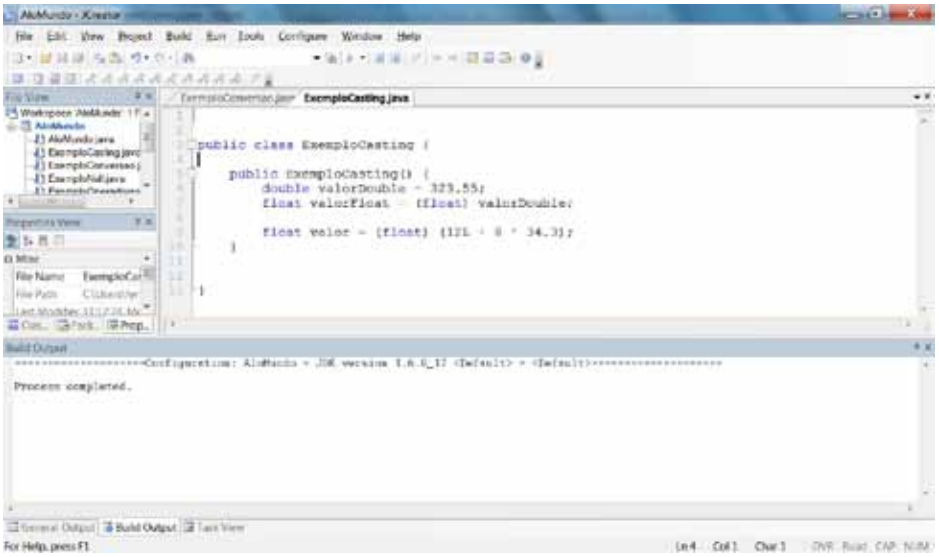

Figura 13 - Exemplo de coerção em Java

Para se fazer a coerção, deve-se explicitamente colocar entre parênteses o tipo para o qual se quer converter um valor ou uma variável. Na linha 7 da Figura 13, isto está sendo feito na conversão de um valor `double` (de maior precisão) para o `float`. Neste mesmo exemplo, a expressão da Figura 13 agora é possível, graças ao uso do *casting* para o tipo `float` (linha 9).

Interessante neste último caso a importância de se colocar os parênteses em toda a expressão. Se por acaso, a expressão fosse construída da seguinte forma

```
float valor = (float) 12L + 8 * 34.3;
```

a coerção estaria sendo feita apenas sobre o valor `12L`. Com isso, a expressão continua sendo avaliada como um todo para `double`, pois estaria envolvendo um `float` (`12`), um `int`(`8`) e um `double`(`34.3`).

Basicamente, a ideia de *casting* é deixar toda a responsabilidade de conversões entre tipos nas mãos dos programadores. Ao fazer uma coerção, é como se o programador dissesse: "Sei que vai haver perda de precisão nesta conversão, mas eu me responsabilizo".

1.7.2 Conversão entre Tipos Primitivos e Complexos com Classe Wrappers

Tipos primitivos e objetos são tratados de forma bem distinta em Java. Com isso, não há como se realizar conversões automáticas entre tipos primitivos e objetos. Porém, em certas situações em que haja necessidade em se converter dados de tipos primitivos em objetos e vice-versa. Para isso, a linguagem oferece um conjunto de classes, chamadas *Wrapper* (empacotadoras).

Estas classes permitem que tipos primitivos possam ser representados como objetos, bem como fazer conversões de tipos complexos para tipos primitivos. Em geral, estas classes fazem correspondência a cada tipo primitivo, como exemplo: `Float`, `Double`, `Byte`, etc. A maioria destas classes tem o mesmo nome de seu tipo primitivo correspondente, diferenciando-se pelo fato que começam com a letra maiúscula. As únicas exceções a esta regra são as classes `Character` e `Integer`, que correspondem respectivamente aos tipos primitivos `char` e `int`.

Por exemplo, para criar um objeto que guarda o valor de inteiro, deve-se usar a seguinte construção:

```
Integer valor = new Integer(100);      // o identificar valor é um objeto que
                                       // encapsula o valor de um inteiro
```

O uso da palavra reservada `new` é a forma padrão de se construir objetos de um determinado tipo e será visto com maiores detalhes no volume 3. Como são objetos, as classes *Wrapper* possuem métodos que permitem a realização de uma série de funcionalidades. Dentre estes métodos, encontram-se aqueles que realizam conversões entre tipos. Para cada tipo de classe *Wrapper*, existe um método `xxxValue()`, que retorna o valor encapsulado pelo objeto como um tipo primitivo. Por exemplo, veja o seguinte trecho de código.

```
Float v1 = new Float(10.5F);
Integer v2 = new Integer(4);
float v3 = v1.floatValue();
int v4 = v2.intValue();
```

Além destes métodos, cada classe *Wrapper* possui um método estático que faz a conversão de uma string em seu tipo primitivo correspondente. Estes métodos seguem um padrão de assinatura: `parseXxx(string)`. A chamada destes métodos não precisa da definição de um objeto, mas apenas a chamada a partir do próprio nome da classe. Veja

alguns exemplos de conversão de strings para tipos primitivos na Figura 14.

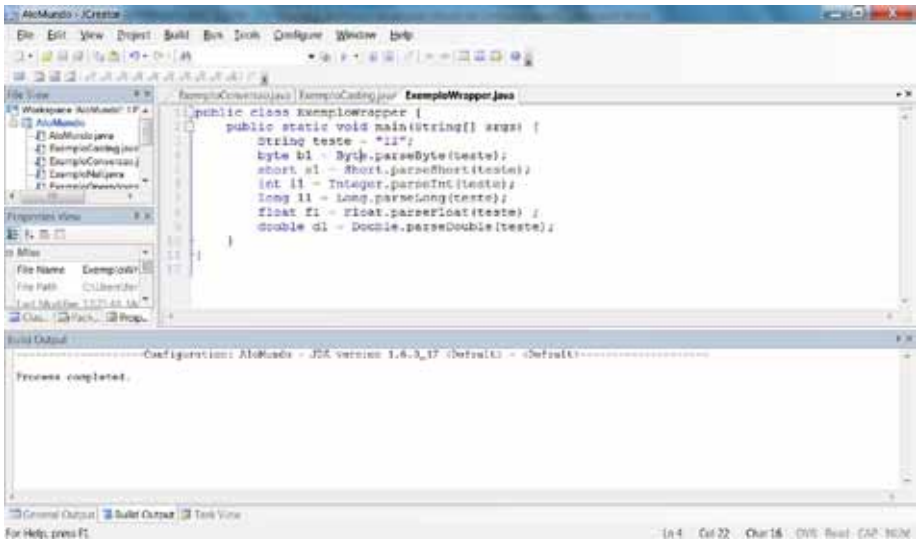

Figura 14 - Conversão de Strings para tipos primitivos

1.7.3 Um pouco mais sobre a classe String

O tratamento de variáveis que guardem valores alfanuméricos é algo extremamente comum em qualquer linguagem de programação. Porém, como em Java, strings são objetos, é interessante tratar o assunto com um pouco mais de atenção. Objetos do tipo `String` encapsulam os valores e protegem sua manipulação por meio de uma série de métodos para a realização de uma série de operações com estes valores.

Primeiramente, por serem objetos, strings também podem ser criados através de construtores, métodos especiais que servem para criar objetos e que serão ainda melhor explorados nos próximos volumes. Isso quer dizer que você pode criar um objeto através do uso da palavra reservada `new`, como no exemplo abaixo:

```
String palavra = new String("Uma palavra qualquer");
```

Esta notação é equivalente a se criar uma String utilizando a notação já apresentada previamente:

```
String palavra = "Uma palavra qualquer";
```

Como citado anteriormente, por obedecer ao conceito de objetos, a classe String oferece uma série de métodos para a manipulação de dados de uma string. Estes métodos podem ser chamados a partir de qualquer referência (identificador) que aponte para uma string. Vamos ver alguns destes métodos na Tabela 6.

Tabela 6 - Métodos da classe String

`char charAt(int index)`
Retorna o caracter localizado na posição especificada pelo argumento `index`, que indica a posição do caracter no array de caracteres representado.

`int compareTo(String outraString)`
Compara uma string com outra string especificada no argumento. Retorna um valor negativo se a string possuir um valor léxico menor que a string passada como argumento, `0` se ambas as strings tiverem o mesmo valor léxico e um valor positivo se a string tiver um valor léxico superior ao da string passada como argumento do método.
`int compareToIgnoreCase(String str)`
Funciona como o método `compareTo` ignorando o padrão *case sensitive* de Java, isto é, maiúsculas e minúsculas são indiferentes, apresentando o mesmo valor léxico.
`boolean equals(Object umObjeto)`
Retorna verdadeiro se a string possuir a mesma sequencia de caracteres do argumento `Object` passado para este método, argumento este que deve ser um objeto do tipo `String`. De outra maneira, se for passado um argumento que não seja do tipo `String` ou se não tiver a mesma sequência de símbolos da string, o método irá retornar `false`.
`boolean equalsIgnoreCase(String outraString)`
Funciona como o método `equals` ignorando o padrão *case sensitive* de Java, isto é, maiúsculas e minúsculas são indiferentes, apresentando o mesmo valor léxico.
`void getChars(int srcIni, int srcFim, char[] dst, int dstIni)`
Obtém os caracteres do array de caracteres da string compreendidos no intervalo definido pelos argumentos `srcIni` (na posição do primeiro caracter no array) e `srcFim` (na posição do último caracter no array) copiando estes caracteres para o array `dst` iniciando na posição `dstIni`.
`int length()`
Retorna o comprimento da string.
`String replace(char antigoChar, char novoChar)`
Retorna a string substituindo todas as ocorrências de `antigoChar` nesta string por `novoChar`.
`String substring(int beginIndice, int endIndice)`
Retorna a substring da string iniciando na posição especificada no argumento `beginIndice` até a posição especificada no argumento `endIndice`.
`char[] toCharArray()`
Retorna o array de caracteres da string.
`public String trim()`
Retorna a string suprimindo todos os seus espaços em branco nas extremidades da String.
`public String toLowerCase()`
Retorna uma nova string com todos os caracteres da string original em letra minúscula.
`public String toUpperCase()`
Retorna uma nova string com todos os caracteres da string original em letra maiúscula.

A Figura 15 apresenta uma classe com o uso de alguns dos métodos apresentados.

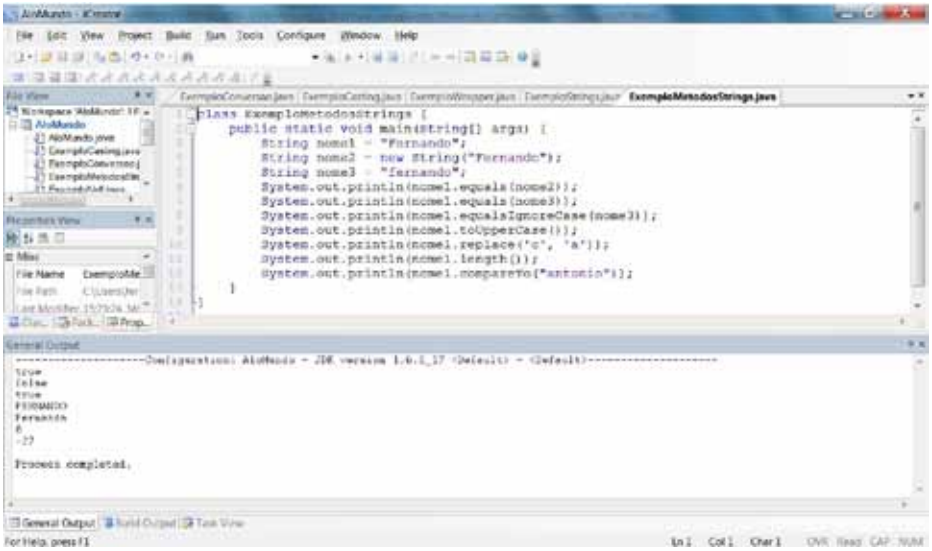

Figura 15 - Métodos da classe String

Particularmente, em relação aos métodos apresentados, o método `compareTo` é utilizado para ordenação entre strings. Veja que como a string armazenada no identificador nome1 começa com "`F`", ele é "maior" que a string `antonio` que começa com "`a`". Por isso, o resultado do método retornou negativo. O valor em si (`-27`) não é o importante, mas sim o fato que retornou um valor menor que 0.

Por fim, é importante dizer que o operador "+", utilizando para soma entre valores primitivos numéricos também é aplicável a objetos string. Neste caso, a "soma" entre duas ou mais strings indica que é criada uma nova string resultante da concatenação de cada valor individual. No caso de operações entre strings e outros tipos primitivos, os valores primitivos são transformados em string, antes de se efetuar a concatenação. Veja o exemplo na Figura 16.

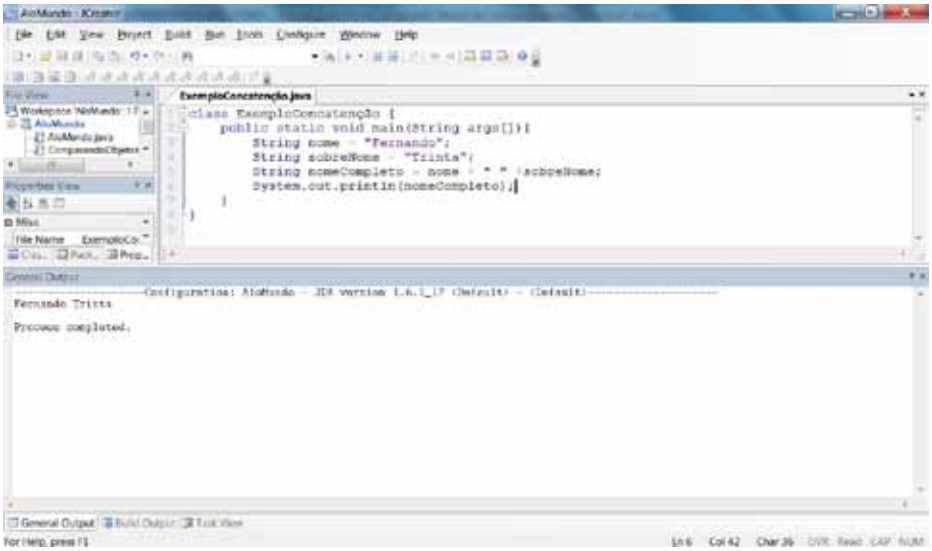

Figura 16 - Exemplo Concatenação

Exercícios

1. Responda: quais dos nomes abaixo são inválidos para identificadores em Java. Justifique cada resposta.
   a. string
   b. _valor
   c. MeuNOME
   d. 123deOliveira4
   e. meuMilhão$$$
   f. assert
2. Indique quais os possíveis tipos de dados para as seguintes variáveis. No que em alguns casos, pode haver mais de um tipo compatível. Justifique cada resposta.
   a. var1 = 10.0;
   b. var2 = 'e';
   c. var3 = 'e' + 10;
   d. var4 = 'e' + "10";
   e. var5 = 1;
3. Crie um programa em java que calcule a seguinte expressão matemática:

$$\frac{3 + 5 \times 6 - \frac{27}{3}}{4^2 - 2^3}$$

4. Escreva um programa em Java que a partir de uma variável inteira definida no início do programa escreve `true` se o número for par ou `false` se o número for ímpar. (dica: utilize o operador %).
5. Explique por as seguintes expressões retornam valores distintos.

```
int x = 10;
int y = 5;
int exp = ++x - ++y;
```

```
int x = 10;
int y = 5;
int exp = x++ - y++;
```

6. Qual o resultado da variável valor de acordo com a expressão abaixo, Justificando sua resposta.

```
int x = 10, y = 10;
boolean valor = ((x == y) == (false));
```

7. Dada a seguinte expressão, responda:

```
<Tipo> resultado = 7 + 34F + 0x34 + "94";
```

   Qual seria o tipo Java correto para ser colocado para a variável resultado? Uma vez substituído, qual seria o valor final da variável resultado? Justifique sua resposta e crie um programa que realize esta expressão.
8. Explique com suas palavras o que é o conceito de *casting*. Qual seu propósito?

9. Escreva um programa em Java que dadas três notas de um aluno armazenadas em variáveis definidas no programa, assim com o próprio nome do aluno, forneça como resultado a mensagem “A média do aluno XXXX foi MÉDIA”, onde XXXX deve ser substituído pelo nome do aluno e MÉDIA deve ser a média aritmética das três notas.
10. Escreva um programa em Java em que a partir de uma variável do tipo String definida no inicio do programa, sejam fornecidas as seguintes informações: quantidade de letras que formam a palavra, a letra inicial da variável, uma nova String a partir da variável original onde apenas a primeira letra esteja em letra maiúscula.

Vamos Revisar?

Você estudou, neste capítulo, como são definidos os tipos e expressões comuns em Java. Este é um assunto bem comum a qualquer linguagem de programação. A principal diferença está na forma como Java trata dados do tipo string. Strings são objetos e têm uma série de peculiaridades que serão melhor vistas no próximo volume. Vamos no próximo capítulo aprender continuar a aprender como Java implementa conceitos comuns como laços e estruturas de decisão.

CAPÍTULO 2

O que vamos estudar neste capítulo?

Neste capítulo, vamos estudar os seguintes temas:

- Operadores lógicos;
- Estruturas de decisão;
- Estruturas de repetição.

Metas

Após o estudo deste capítulo, esperamos que você consiga:

- Construir programas com suporte a tomada de decisões e que possam ter diferentes fluxos de execução;
- Construir programas com suporte a repetição de operações através do conceitos de laços.

CAPÍTULO 2 – OPERADORES E CONTROLES DE FLUXO

Vamos conversar sobre o assunto?

Vamos continuar neste capítulo a apresentar como Java apresenta conceitos clássicos de linguagens de programação. Para isto, você vai aprender sobre operadores relacionais, operadores lógicos, bem como estruturas de decisão e laços de repetição. Isso vai permitir que você construa programas ainda mais elaborados na linguagem.

2.1 INTRODUÇÃO

Em qualquer linguagem de programação, há necessidade de se realizar testes por meio de comparações. Estas comparações envolvem, em geral, variáveis e expressões. Java oferece operadores e construções para a realização de testes. Java também oferece operadores para estruturas de repetição. Neste capítulo, vamos dar uma olhada na sintaxe Java para estes conceitos.

2.2 OPERADORES RELACIONAIS

Os operadores relacionais estabelecem uma relação entre seus operandos. Nada mais óbvio que isso! Estas relações são descritas na tabela abaixo.

Tabela 7 – Operadores Relacionais em Java

==	Igualdade
!=	Diferença
>	Maior que
<	Menor que
>=	Maior ou igual que
<=	Menor ou igual que

Estes operadores tomam dois argumentos para cada comparação e sempre produzem um valor booleano, `true` ou `false`. Estes argumentos podem ser duas variáveis ou uma variável e uma constante.

Quando estes valores ou variáveis são de tipos primitivos, todos os operadores podem usados. Quando objetos são utilizados, apenas os operadores de igualdade (==) e diferença (!=) podem ser utilizados. Neste último caso, as comparações são feitas entre as referências destes objetos. Por isto, vejamos o exemplo da Figura 17.

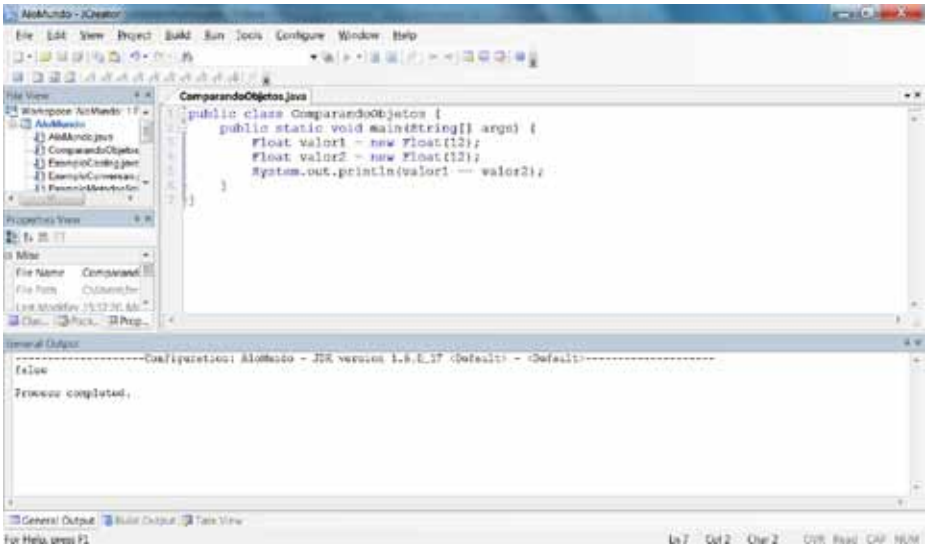

Figura 17 - Comparação entre Objetos

Os identificadores `valor1` e `valor2` são dois objetos do tipo `Float`. Embora estejam armazenando dados que representem o mesmo valor, utilizar o operador entre objetos significa perguntar se as duas variáveis apontam para o mesmo objeto, o que não é caso.

Agora veja o exemplo da Figura 18. Veja que estamos comparando strings, e que em Java, strings são tratados como objetos. Logo, uma pergunta que se pode fazer é por que a comparação entre os objetos valor1 e valor2 resultou em true. Já que são objetos, não foram criados dois objetos?!

A resposta a esta pergunta passa pelo projeto de Java, que para melhorar o desempenho no uso de uma construção tão comum, estabeleceu que strings criadas a partir da sintaxe `String var = "valor"`; são colocadas em um pool de strings, fazendo com que as referências sejam compartilhadas. O comportamento padrão de um objeto acontece quando se constrói uma string utilizando o operador `new`. Neste caso, é criado um novo objeto.

Por conta disso, uma boa prática de programação em Java é: sempre que quisermos comparar strings, devemos utilizar os métodos equals ou equalsIgnoreCase. Estes métodos sempre comparam o conteúdo da string, como pode ser visto no próprio exemplo da Figura 18.

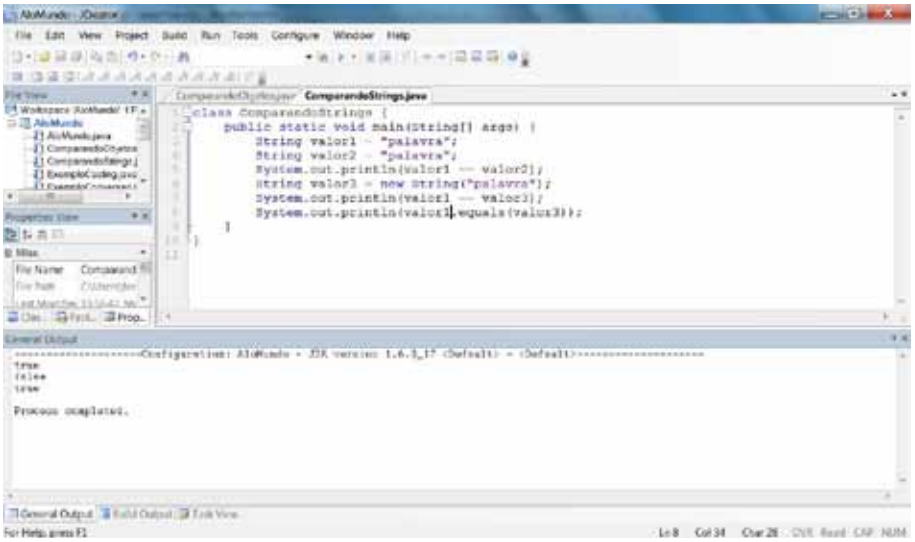

Figura 18 - Comparando Strings

2.3 Operadores Lógicos

Os operadores lógicos utilizam lógica proposicional para apoiar ou rejeitar proposições. Estes operadores complementam os operadores relacionados apresentados previamente. Em programação de computadores, operadores lógicos são usados para combinar os valores lógicos (`true/false`) e obter novos valores lógicos que determinam o fluxo de controle de um algoritmo ou programa. A Tabela 5 descreve os operadores lógicos em Java.

Tabela 8 - Operadores lógicos em Java

&&	E lógico	a && b	retorna true se a e b forem ambos true. Senão retorna false. Se a for false, b não é avaliada.
&	E lógico booleano	a & b	retorna true se a e b forem ambos true. Senão retorna false. Ambas expressões a e b são sempre avaliadas.
||	OU lógico	a || b	retorna true se a ou b for true. Senão retorna false. Se a for true, b não é avaliada.
|	OU lógico booleano inclusivo	a | b	retorna true se a ou b for true. Senão retorna false. Ambas expressões a e b são sempre avaliadas.
^	OU lógico booleano exclusivo	a ^ b	retorna true se a for true e b for false ou vice-versa. Senão retorna false
!	Não Lógico (negação lógica)	!a	retorna true se a for false. Senão retorna false

Como você pode perceber, todos os operadores lógicos recebem dois parâmetros, com exceção da negação. Estes argumentos precisam ser valores booleanos ou expressões que levem a um resultado booleano – por exemplo:

```
(x > 3) && (y == 4)
```

Os operadores || e `&&` são chamados de *short-circuit* (curto circuito). A ideia é que se o resultado de uma expressão já é suficiente para se saber o resultado da expressão maior na qual ela está contida, os demais testes são desnecessários. Veja este exemplo:

```
int x = 4;
int z = 8;
boolean z = (x > 4) || (z <9) && (x + z >= 3)
```

Neste caso, a primeira expressão (x > 4) já retorna `true`. Logo, poderíamos simplificar a expressão para

```
boolean z = true || <resto_das_expressões>
```

Como se trata de um OU lógico, basta que ter um operando igual a `true`, que já sabemos que o resultado da expressão como um todo será também `true`. Então para que perder tempo com mais testes?! Com isso, as expressões seguintes são ignoradas, e o

resultado já é fornecido.

2.4 Operadores de Deslocamento

Java fornece também operadores que permitem o deslocamento de bits em números inteiros. Estas operações correspondem a mover uma determinada quantidade de bits para a direita ou para a esquerda e são apresentados na Tabela 6.

Tabela 9 – Operadores de Deslocamento de ***bits***

<<	Deslocamento de bit à esquerda	Equivale a multiplicar o número por 2
>>	Deslocamento de bit à direita	Equivale à divisão inteira (truncada) do número por 2
>>>	Deslocamento de bit à direita sem considerar sinal	

Veja este exemplo:

```
int x = 255;    // em binário equivale a 11111111
int z = x << 1; // desloca-se um bit a esquerda e adiciona-se 0 a direita
                // z = 111111110, ou 510 em decimal
Int w = x >> 1  // desloca-se um bit à direita e adiciona-se 0 à esquerda
                // z = 01111111, ou 127 em decimal
```

Os operadores de deslocamento de bits operam sempre sobre inteiros (`int`) e inteiros longos (`long`). Tipos menores como `short` ou `byte` são convertidos para `int` antes da realização a operação de deslocamento.

E tem mais. Da mesma forma que os operadores aritméticos, os operadores podem sem combinados com atribuição em uma versão simplificada, no caso, <<=, >>= e >>>=. Por exemplo:

```
int x = 255;
x >>=2; // equivale à x = x >> 2;
```

2.5 Operadores de Execução

Visto os operadores que manipulam as variáveis e objetos, vamos falar agora sobre as estruturas utilizadas para o controle da execução de um programa em Java. Estas estruturas são similares aos encontrados em outras linguagens. São elas:

- **Estruturas de Seleção**: Também chamada de expressão condicional ou ainda construção condicional, esta estrutura realiza diferentes ações dependendo dos valores de uma condição que, em geral, podem assumir como valores verdadeiro ou falso;
- **Estruturas de Repetição**: Esta estrutura realiza e repete um conjunto de ações de acordo com o valor de uma determinada condição que, em geral, assume o valor

verdadeiro ou falso.

Vamos ver como programadores podem utilizar estas estruturas em Java.

2.5.1 Estruturas de Seleção

Java apresenta duas estruturas de seleção básicas: O `if-else` e o `switch`.

2.5.1.1 if – else

A estrutura `if-else` representa a ideia de que dependendo do valor de uma condição (variável ou expressão), dois caminhos distintos podem ser tomados na linha de execução de um programa. A sintaxe mais simples para a estrutura if-else é a seguinte:

```
if (expressão_booleana)
    instrução_simples;
```

Nesta forma, se a expressão de teste for verdadeira, a instrução imediatamente após ao teste será executada. Caso contrário, esta instrução é ignorada. Veja o exemplo:

```
// Assuma que as variáveis idade e sexo são definidas em algum ponto
// anterior a este trecho de código
if (idade > 18)
    System.out.println("Trata-se de um maior de idade");
if (sexo == 'F')
    System.out.println("É uma pessoa do sexo feminino");
```

Note que existem duas estruturas if associadas a dois testes distintos. Nestes dois exemplos, apenas a instrução imediatamente após o teste é realizada, caso a expressão seja avaliada como verdadeiro. Porém existem situações onde mais de uma instrução precisa ser realizada de acordo com a avaliação de uma expressão. Neste caso, deve-se fazer o uso de chaves ({ }), como no modelo abaixo.

```
if (expressão_booleana) {
    instrução1;
    instrução1;
    ...
    instruçãoN;
}
```

Veja o exemplo:

```
public class ExemploIF {
    public static void main(String[] args) {
        int idade = 15;
        char sexo = 'M';
        if ((idade > 18) && (sexo == 'F')) {
                System.out.println("Trata-se de um maior de idade");
                System.out.println("É uma pessoa do sexo feminino");
```

```
            }
    }
}
```

Neste caso, a expressão a ser avaliada é composta de duas subexpressões mais simples. Se o resultado for verdadeiro (true), todas as instruções contidas no bloco entre chaves serão executadas.

A estrutura mais completa para utilização desta estrutura de decisão `if` é aquela que é complementada pela cláusula `else`, e que indica o que deve ser executado no caso de expressão avaliada ser falsa (`false`). Veja o modelo abaixo:

```
if (expressão_booleana) {
    instruções para expressão_booleana == true
} else
    instruções para expressão_booleana == false
}
```

O exemplo abaixo ilustra a aplicação da estrutura completa de um `if`:

```
public class ExemploIF {
    public static void main(String[] args) {
        int idade = 15;
        if (idade > 18)
                System.out.println("Trata-se de um maior de idade");
        else
                System.out.println("Trata-se de um menor de idade");
    }
}
```

Da mesma forma como para os exemplos anteriores, pode-se utilizar chaves para marcar a execução de mais de uma instrução tanto para quando a condição de teste for verdadeira quanto falsa. Veja o exemplo na Figura 19.

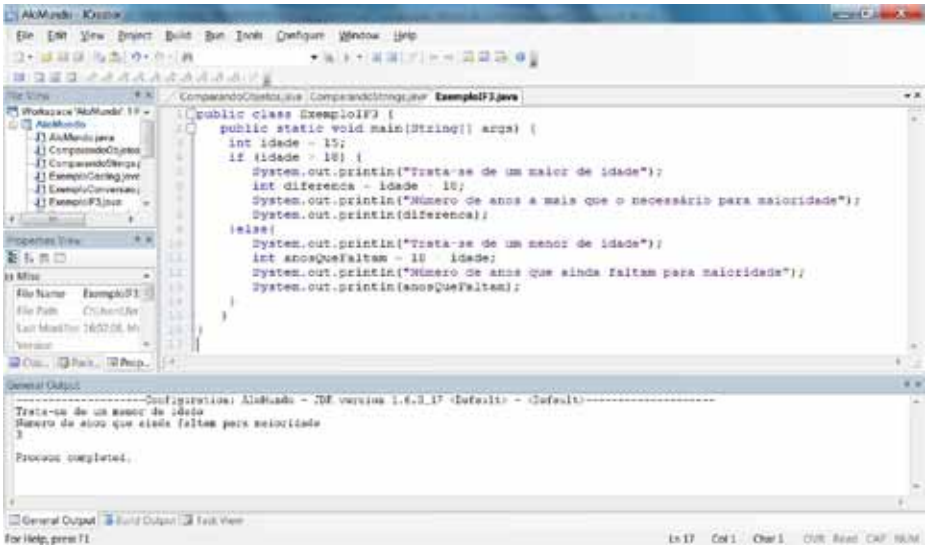

Figura 19 – Exemplo IF/ELSE

De acordo com a exigência apresentada para a solução de alguns problemas, pode haver a necessidade de se utilizar uma sequência de passos onde uma estrutura `if-else` pode conter outras estrutura de decisão. Quando isso ocorre, dizemos que houve um aninhamento de `if´s`. É possível encadear uma série de testes e um fluxo de execução bem complexo de acordo com a combinação destes testes. Neste caso, a sintaxe do comando `if` seria da seguinte forma:

```
if (expressão booleana) {
   instruções
   } else if (expressão booleana) {
      instruções
   } else {
      instruções
}
```

Neste caso, a partir do primeiro teste, os demais testes são realizados se a condição de seu teste anterior for falso (`false`). Veja o exemplo da Figura 20.

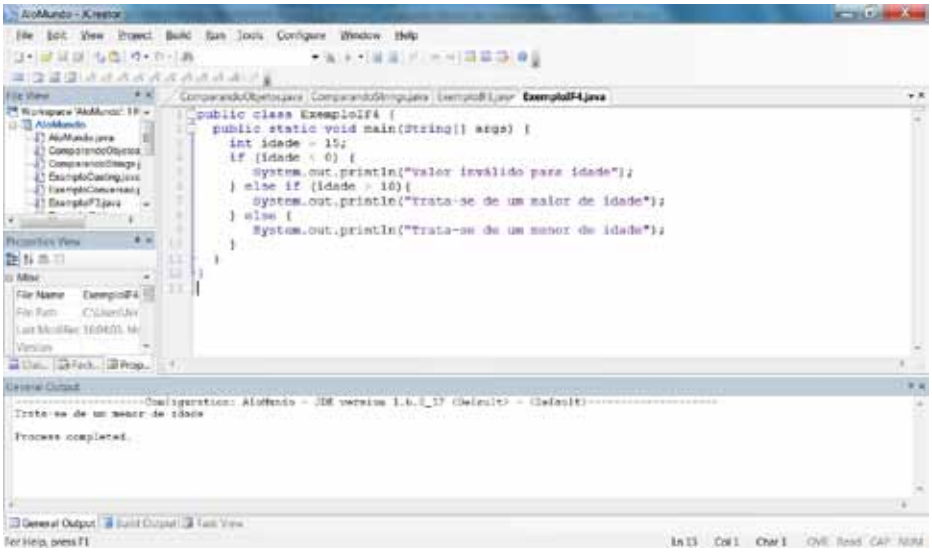

Figura 20 – Exemplo de IFs aninhados

Neste caso, percebe-se que os testes são efetuados sequencialmente até se encontrar uma condição verdadeira. Caso isso não exista, a última condição de else é executada.

2.5.1.2 switch – case

A estrutura `switch` é usada como alternativa ao aninhamento de `if´s`. Com ele o código se torna mais elegante e legível. Sua sintaxe é a seguinte:

```
switch(seletor_inteiro) {
    case valor_inteiro_1 :
        instruções;
        break;
    case valor_inteiro_2 :
        instruções;
```

```
        break;
        ...
    default:
        instruções;
}
```

Ao contrário do `if`, a estrutura `switch` recebe um valor ou expressão que possa ser convertida para um inteiro, chamado de *seletor*. Na estrutura, existe um conjunto de casos (`cases`), associados a possíveis valores do seletor. O conjunto de instruções a ser executada é escolhido de acordo com o valor de seleção. Se não houver nenhum valor válido, o trecho de instruções *default* é executado. Veja o exemplo da Figura 21.

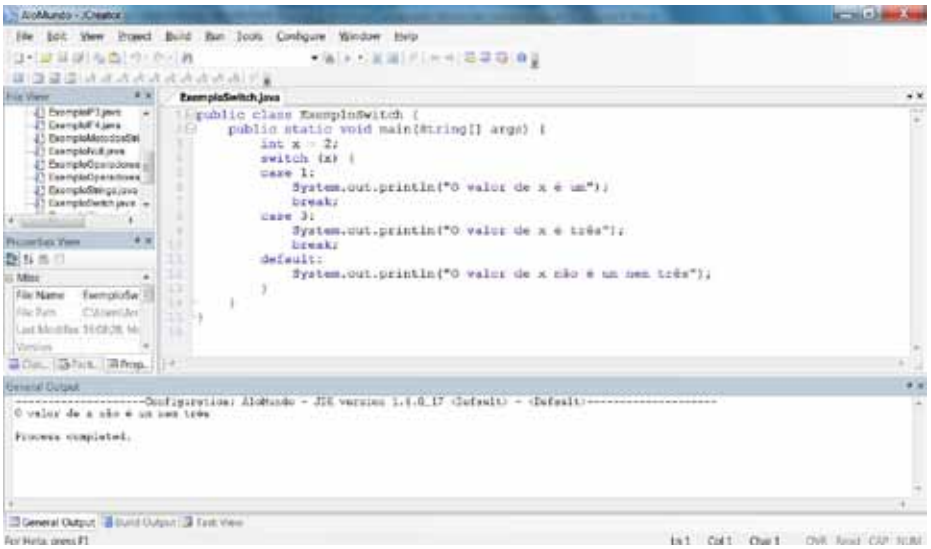

Figura 21 – Exemplo Switch

O valor utilizado para ser avaliado no `switch` pode também ser um `char`. Uma coisa importante é que ao contrário dos blocos que utilizam chaves para limitar um conjunto de instruções, cada opção (case) executa todas as instruções incluídas em sua definição, até encontrar a palavra reservada `break`, e que por sinal, não é obrigatória. Nestes casos, a linha de execução continua pelas instruções das opções (`case`) imediatamente em sequência na definição do `switch`. Veja o exemplo:

```
public class SwitchDemo {
    public static void main(String[] args) {
        int mes = 2;
        int ano = 2000;
        int numeroDias = 0;
        switch (mes) {
                case 1:
                case 3:
                case 5:
                case 7:
                case 8:
                case 10:
```

```
                case 12:
                        numeroDias = 31;
                        break;
                case 4:
                case 6:
                case 9:
                case 11:
                        numeroDias = 30;
                        break;
                case 2:
                        if ( ((ano % 4 == 0) && !(ano % 100 == 0))
                                        || (ano % 400 == 0) )
                                numeroDias = 29;
                        else
                                numeroDias = 28;
                                break;
                default:
                        System.out.println("Mês inválido.");
                        break;
                }
                System.out.println("Número de dias = " + numeroDias);
        }
}
```

O programa acima calcula o número de dias de um mês, de acordo com as entradas de numero de mês (1 a 12) e do ano, utilizado para verificar se o ano é ou não bissexto. No exemplo, note que quando a variável `mes` for igual a 3, o fluxo de execução do programa vai entrar no `case` específico para este valor. Neste `case`, não há nada para fazer, mas também não há um `break`. Então o fluxo de execução passa para a linha seguinte, que corresponde ao `case` para o valor 5, e assim por diante, até chegar ao `case` para o valor 12, onde finalmente é definido um valor para a variável `numeroDias` e é encontrado um `break`, que finaliza a execução do `switch`. A linha seguinte apresenta a mensagem do número de dias para aquele mês, naquele ano.

Embora não seja obrigatório, o uso do `break` é altamente recomendável, pois torna o código bem mais fácil de ser modificado e também menos suscetível a erros. Outra importante nota é que o `switch/case` não utiliza valores booleanos, logo não é possível utilizar comparadores no `case`, como por exemplo, `case x > 10`.

2.5.2 Estruturas de Repetição

As estruturas de repetição são utilizadas quando desejamos que um determinado conjunto de instruções ou comandos sejam executados um número definido ou indefinido de vezes, ou enquanto um determinado estado de coisas prevalecer ou até que seja alcançado. Estas estruturas são comumente chamadas de laços de repetição ou simplesmente laços. Em Java existem 3 formas possíveis de laços: o `do..while`, o `while..do` e o `for`.

Vamos ver como cada um deles funciona.

2.5.2.1 O do..while

A sintaxe do laço do..while é apresentada abaixo:

```
do{
   // instruções
}while(condição_booleana);
```

A semântica (ou funcionamento) desta estrutura indica que as instruções que estejam no bloco de instrução delimitado pelo `do/while` serão executadas continuamente, enquanto a condição booleana for verdadeira. Veja o exemplo da Figura 22.

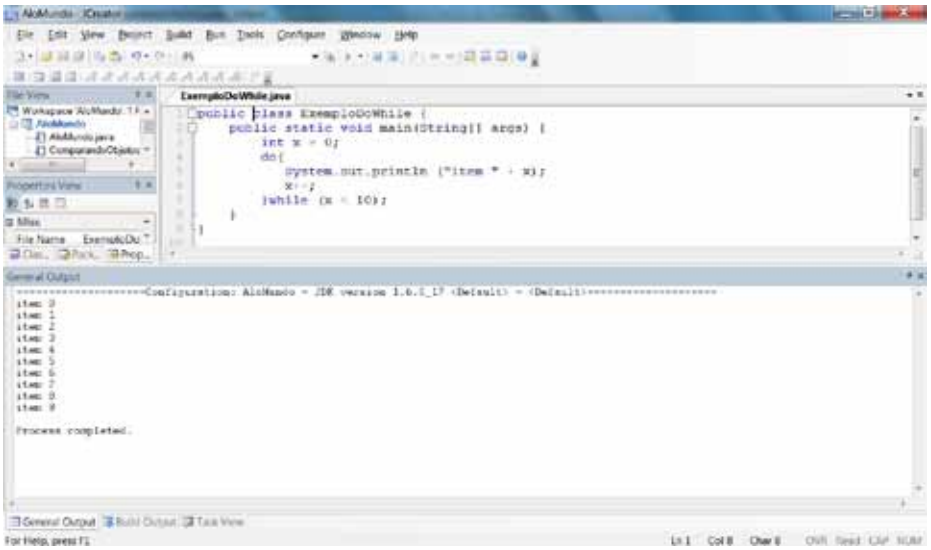

Figura 22 – Exemplo DO/While

Neste caso, o laço será executado dez vezes, pois em cada iteração do laço, o valor da variável `x` é incrementada de um. Enquanto ela for menor que 10, o programa fica executando as instruções do laço. Quando a condição for falsa, o programa segue seu fluxo de execução.

A estrutura `do..while`, assim como todas as outras estruturas de repetição utilizam valores booleanos ou expressões que retornem valores booleanas em sua condição de saída do laço.

É fácil perceber que na estrutura `do..while`, o teste de saída é realizado apenas ao fim da execução das instruções, o que garante a execução de pelo menos uma iteração do laço.

2.5.2.2 O while..do

O laço `while..do` é muito parecido com o laço anterior. Porém, a condição de saída é testada logo no início das instruções. Veja o exemplo da figura a seguir.

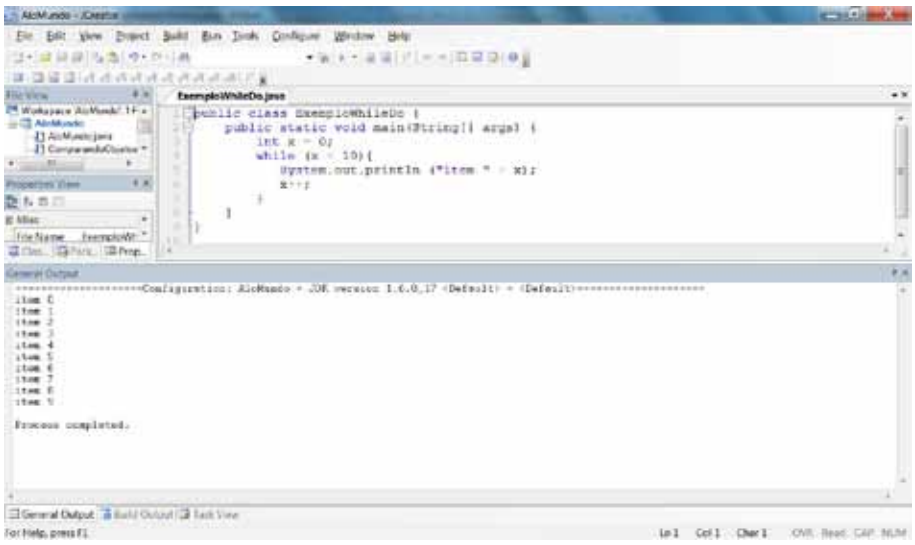

Figura 23 - Exemplo while/do

2.5.2.3 O `for`

O `for` representa uma forma alternativa de escrever as regras que determinam as condições para a repetição das instruções do laço. O laço `for` é representado da seguinte forma:

```
for (inicialização; condição de teste; incremento){
    instruções
}
```

Cada uma das três seções do laço tem uma função específica. Na parte da inicialização, são executadas ações durante a primeira iteração do laço. É muito comum no laço for a declaração de uma variável de controle (comumente chamada índice) que será utilizada para determinar a saída do laço. Por exemplo, pode-se declarar algo como `int i = 0`, sendo que a variável declarada tem seu escopo restrita ao bloco do laço.

A condição de teste é executada antes da execução de cada interação. Este condição deve retornar um valor booleano, e em geral envolve a variável de controle definida no bloco de inicialização. Se a condição de teste for verdadeira, as instruções do bloco `for` serão executadas. Caso contrário, o laço não será executado, e o fluxo de execução seguirá para a primeira instrução após a definição do laço.

A última parte do laço `for` é um trecho de código que é executado ao final de cada iteração. Normalmente, trata-se de um incremento utilizado para mudar o valor da variável de controle, de modo a aproximar o laço de seu término.

O código a seguir representa um laço que calcula a soma dos 10 primeiros números inteiros.

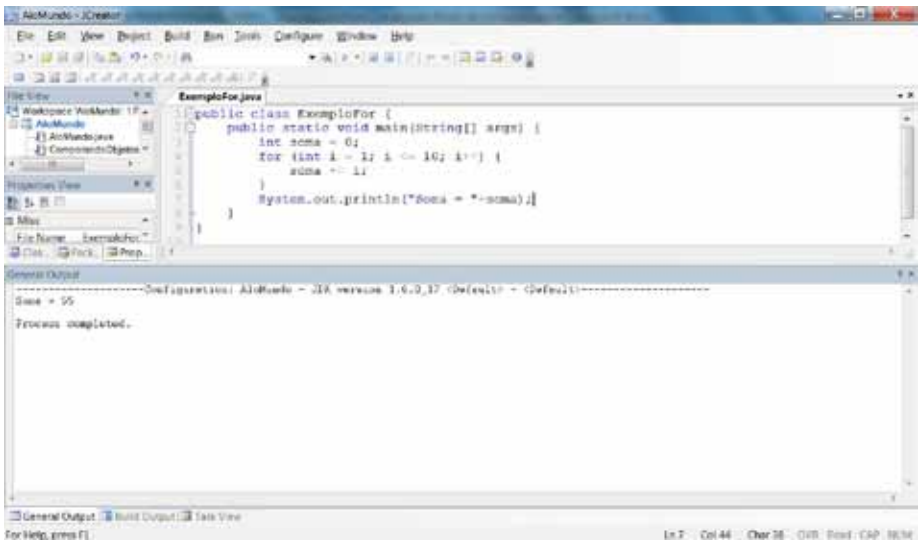

Figura 24 - Exemplo laço for

2.5.2.4 Interrupção de Laços

Para todos os tipos de laços apresentados, haverá uma condição de teste que determina o final do laço[1], e todas as instruções do laço serão executadas para cada iteração do laço. Porém, existem duas palavras reservadas que permitem alterar o funcionamento dos laços. São elas: o `break` e o `continue`.

A palavra reservada `break` para imediatamente a execução, e equivale ao fato da condição de saída do laço ser verdadeira. Veja o exemplo da Figura 25.

Neste exemplo, o laço não tem uma condição de saída definida, uma vez que é fixado o valor `true` na definição do laço. Trata-se de um laço infinito. Neste caso, a condição de saída é determinada pelo `if`, que indica que quando o valor da variável `x` for maior que `100`, o comando `break` deve ser executado. Esta execução fará com que o laço seja interrompido. Você pode argumentar que este laço poderia ser escrito tranquilamente com a construção tradicional do laço `while`. Isto é verdade. Porém, em certas situações, o `break` é muito útil, como no tratamento de vetores e matrizes, assim como para lidar com entrada do usuário.

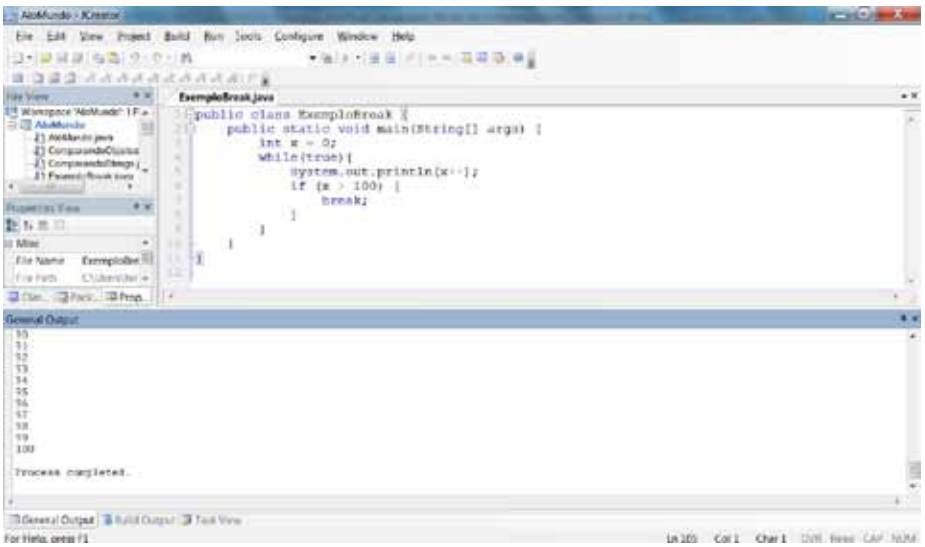

Figura 25 – Exemplo Break

Saiba Mais

Você sabia que você pode criar laços infinitos em java?! Se você utilizar a construção

```
for (;;)
```

```
System.out.
println("oi");
```

Não há uma condição de saída definida. Com isso, o programa ficará para sempre escrevendo a mesma mensagem.

Já a palavra-reservada `continue` faz com que a iteração atual seja interrompida, ignorando as instruções seguintes à chamada deste comando. Porém, em vez de abandonar o laço, a instrução `continue` força o reinício do laço a partir da próxima iteração. Nesta nova iteração, as condições de saída são novamente avaliadas, e para o laço `for`, a condição de incremento também é executada. Veja o exemplo da Figura 26.

Neste exemplo, tem-se um laço com uma variável de controle que varia de -10 até 10. Toda vez que o valor do índice de controle é par, o fluxo de execução do laço é retornado ao início do laço. Caso contrário, o fluxo de execução segue normalmente e o valor do índice de controle é apresentado na tela.

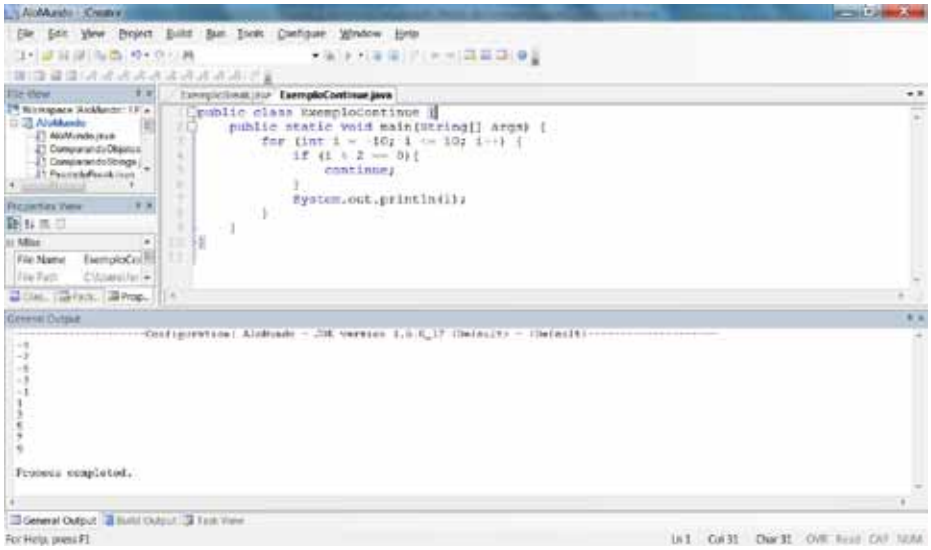

Figura 26 – Exemplo continue

2.5.2.5 Loops Aninhados e Rótulos

Laços podem ser sobrepostos para ser utilizados onde haja a necessidade de mais de uma variável de controle. Isto vale para qualquer tipo de laço. Veja o exemplo da Figura 27 que imprime a tabuada de 1 a 10.

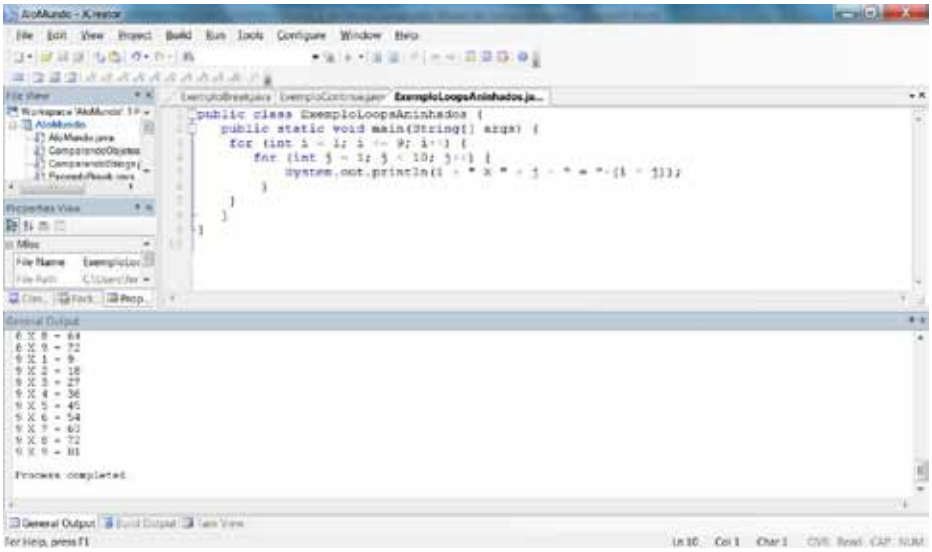

Figura 27 - Exemplo de Loops Aninhados para impressão da tabuada de 1 a 10

Pode haver infinitos níveis de aninhamento em laços de repetição. Estes níveis são importantes também quando do uso dos comandos `break` e `continue`. Quando usamos estes comandos em laços aninhados, apenas o laço mais interno é afetado. Veja este exemplo na Figura 28.

Neste caso, existem dois laços aninhados. Um mais externo utilizando a notação `for` e outro mais interno com a notação de `while`..`do`. Dentro do laço mais interno, existe um teste (linha 8) que em caso positivo, chama a instrução break. A execução desta instrução ocasiona a saída apenas do laço mais interno. Neste caso, o fluxo de execução retorna ao laço mais externo, que por sua vez, também inicia novamente a execução do laço `while`. Isso vai acontecer até a condição de saída do laço mais externo ser considerada verdadeira.

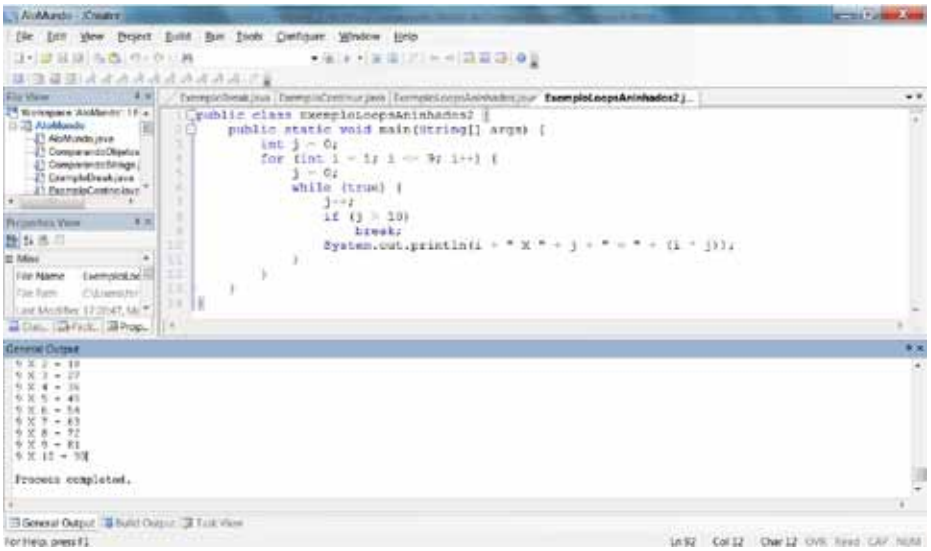

Figura 28 - Exemplo de Loops aninhados e o uso de rótulos

Porém, em certas situações, pode-se precisar que os comandos `break` ou `continue` afetem um laço específico. Neste caso, pode-se utilizar rótulos que marquem os laços, e que podem ser utilizados pelos comandos de interrupção para indicar que estes devem afetar um laço específico, e não o laço corrente.

Para usar um rótulo, deve-se adicionar um identificador antes da definição do laço. Entre o identificador e o laço utiliza-se dois pontos (:) como separador. Desta forma, tanto o `break` quanto o `continue` podem adicionar este identificador na sua chamada. Veja o exemplo da Figura 29.

Quando a condição de teste da linha 7 for verdadeira, haverá a execução da instrução `break`. Porém neste caso, há uma declaração explícita indicando que se quer interromper o laço marcado com o rótulo "`externo`". Isso implicitamente encerra todos os laços que estejam contidos dentro deste laço rotulado.

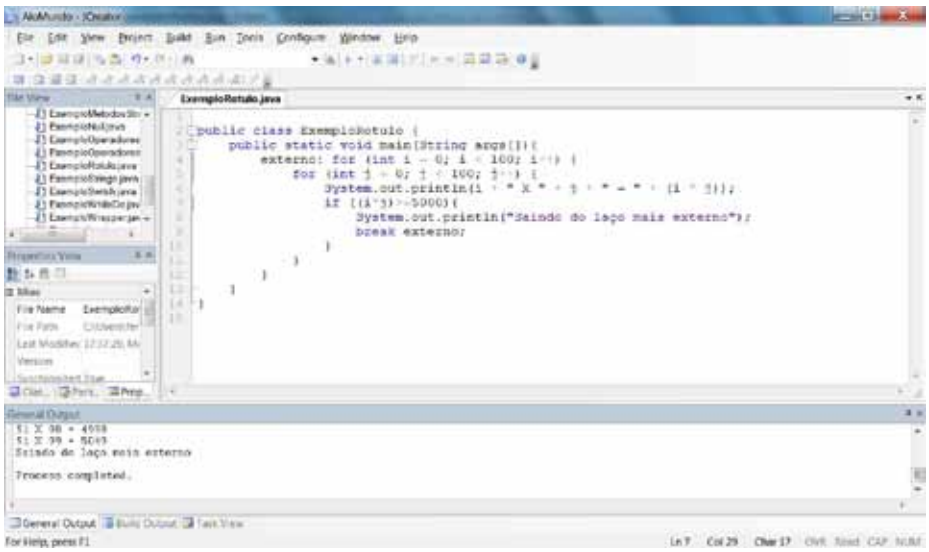

Figura 29 - Exemplo de laços com rótulos

Exercícios

1. Escreva um programa em Java que a partir de variáveis definidas no início do programa e que representem duas notas e o número de faltas do aluno, apresente uma mensagem de acordo com as seguintes regras.

   a) Se o número de faltas for maior que 15, a mensagem será: “Aluno reprovado por faltas”;

   b) Se o número de faltas for menor que 15, e a média das duas notas for maior ou igual a 7, a mensagem será: “Aluno aprovado”;

   c) Se o número de faltas for menor que 15, e a média das duas notas for menor que 7 e maior ou igual a 4, a mensagem será: “Aluno de recuperação”;

   d) Se o número de faltas for menor que 15, e a média das duas notas for menor que 4, a mensagem será: “Aluno reprovado”;

2. Escreva um programa em Java que a partir de variáveis definidas no início do programa e que representem o dia (int) e mês (int) escrevam a data representada, por extenso. Ex: Se dia = 11 e mês = 6, a saída será igual a “onze de junho”. Utilize a estrutura `switch/case`.

3. Reescreva o código abaixo utilizando o laço for.

```
int numero = 10;
   while (numero < 15) {
   System.out.println(“*”);
   numero ++;
}
```

4. A conversão de um valor registrando uma temperatura pode ser apresentada utilizando graus Celsius ou Fahrenheit. A conversão entre estas escalas é dada pela fórmula:

$$\frac{Celsius}{5} \rightarrow \frac{Fahrenheit - 32}{9}$$

Escreva um programa em java que apresente os valores de temperatura entre -10 e 40 graus Celsius convertidos em Fahrenheit.

5. Escreva um programa em Java que dada a variável x definida no início do programa, calcule o fatorial deste número.
6. A série de fibonacci é uma conhecida sucessão de números que começam 0 e 1, e que os demais termos são resultados da soma de seus dois antecessores, como ilustrado abaixo:

   0,1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...

   Escreva um programa em java que apresente os 50 primeiros termos desta série.
7. Um número é considero primo se ele só divisível por si próprio e por um. Escreva um programa em Java que a partir de um valor definido em uma variável no início do programa, apresenta todos os números primos de 1 até o número escolhido. Dica: Use laços aninhados.

Vamos Revisar?

Neste capítulo, continuamos a ver como Java implementa uma série de elementos básicos encontrados na maioria das linguagens de programação modernas. As estruturas de repetição e decisão, bem como os operadores vistos até aqui serão parte fundamental para os objetos e classes que serão criadas mais adiante no curso. Por enquanto, você deve fazer bastante exercícios para se familiarizar com estes conceitos.

CAPÍTULO 3

O que vamos estudar neste capítulo?

Neste capítulo, vamos estudar os seguintes temas:

» Vetores e Matrizes;

» Entrada de dados pelo teclado.

Metas

Após o estudo deste capítulo, esperamos que você consiga:

» Utilizar o conceito de vetores em Java para agrupar dados em um conjunto;

» Aprender como consentir que programas simples em Java permitam uma interação do usuário através da entrada de dados pelo teclado.

CAPÍTULO 3 – VETORES, MATRIZES E CAPTURA DE DADOS PELO TECLADO

Vamos conversar sobre o assunto?

Este capítulo encerra a explanação de como conceitos conhecidos em outras linguagens de programação são implementados em Java. Vamos ver nele como agrupar conjuntos de informações em uma única estrutura de dados, os vetores. Para finalizar, vamos enriquecer a interatividade de nossos programas permitindo que o usuário possa interagir por meio da entrada de dados através do teclado.

3.1 INTRODUÇÃO

Em praticamente toda linguagem de programação, há a necessidade de se guardar e tratar elementos como um único conjunto. Em linguagens de programação esta ideia é geralmente tratada através do conceito de vetores e matrizes (também chamados de *arrays*).

Um vetor ou uma matriz representa uma lista de itens, onde cada item guarda um valor individual. Você pode pensar em um vetor como um conjunto de variáveis individuais. Porém, este conjunto permite que cada valor individual possa ser acessado através de um índice que identifica cada variável no conjunto.

Java, porém, contém uma peculiaridade em relação à implementação de vetores. Em Java, vetores são objetos. Isso indica que estes possuem atributos e métodos que podem ser chamados por outros objetos, além de permitir que diferentes identificadores apontem para o mesmo vetor (da mesma forma como Strings).

Vetores e Matrizes podem conter qualquer tipo de dado (quer sejam tipos primitivos ou objetos), porém sem misturar estes tipos. Isso quer dizer que um vetor de inteiros armazena apenas inteiros, enquanto um vetor de strings guarda apenas strings.

3.2 PASSOS PARA CRIAÇÃO DE VETORES

Para definir um vetor ou uma matriz em Java são necessários três passos:

- Declaração de um identificador para referenciar o vetor;
- Definição de um novo objeto para guardar os valores do vetor;
- Atribuir valores a cada elemento do vetor.

Vamos ver como cada uma destas etapas é realizada.

3.2.1. Declaração de um identificador para o vetor

A declaração de um identificador para um vetor é bem parecida com a declaração de uma variável comum. É necessário definir o tipo dos valores do vetor e um identificador para o vetor. Porém, a diferença é que o nome da variável deve ser acompanhado por colchetes vazios (`[]`). São exemplos da definição de vetores em Java:

```
int numerosNaturais[];

float notasAlunos[];

String nomesAlunos[];
```

Na realidade, a definição de um vetor também pode utilizar a notação onde os colchetes são colocados após o tipo do vetor. Logo, as definições anteriores são equivalentes às apresentadas a seguir:

```
int[] numerosNaturais;

float[] notasAlunos;

String[] nomesAlunos;
```

3.2.2. Definição de um novo objeto para guardar os valores do vetor

Para se criar um objeto que represente uma matriz deve utilizar a seguinte notação:

```
<identificador> = new <Tipo>[quantidade de elementos];
```

O identificador pode ser um previamente criado ou pode-se criar este identificador na mesma linha que se estabelece a quantidade de elementos do vetor. Por exemplo, imaginando que fôssemos utilizar um dos vetores exemplificados na seção anterior, poderíamos ter as seguintes declarações de vetores:

```
int[] numerosNaturais = new int[10];  //Criação de um vetor de 10 inteiros

float[] notasAlunos = new float[20];    //Criação de um vetor de 10 números reais

int x = 30;

String[] nomesAlunos = new String[x]; //Criação de um vetor de 30 objetos do
                                      //tipo String
```

É importante ressaltar que ao se criar um vetor, todos os elementos são automaticamente inicializados para os valores padrão do tipo de elementos do vetor. São estes: `0` (zero) para números, `'\0'` para caracteres, `false` para booleanos e `null` para objetos.

3.2.3. Atribuir valores a cada elemento do vetor

Para se atribuir valores e consequentemente alterar os valores de um vetor é necessário saber como acessar individualmente cada elemento do conjunto. Em Java, cada elemento do vetor possui um índice de acesso, e a notação para acesso através deste índice é `vetor[índice]`.

Por exemplo, para acessar o elemento de índice 5 do vetor numerosNaturais, teríamos a seguinte notação: `numerosNaturais[5];`

É importante alertar que utilizar o índice 5, não significa acessar o quinto elemento do vetor. Assim como em C e C++, índices de vetores em Java começam com 0(zero). Desta forma, o vetor `numerosNaturais` possui 10 elementos, acessados pelos índices 0 a 9.

Na realidade, a tentativa de acesso a um índice inválido de um vetor é um dos erros mais comuns de programação para iniciantes em Java. Veja o trecho de código representado pela classe `TestaIndice` (Figura 30).

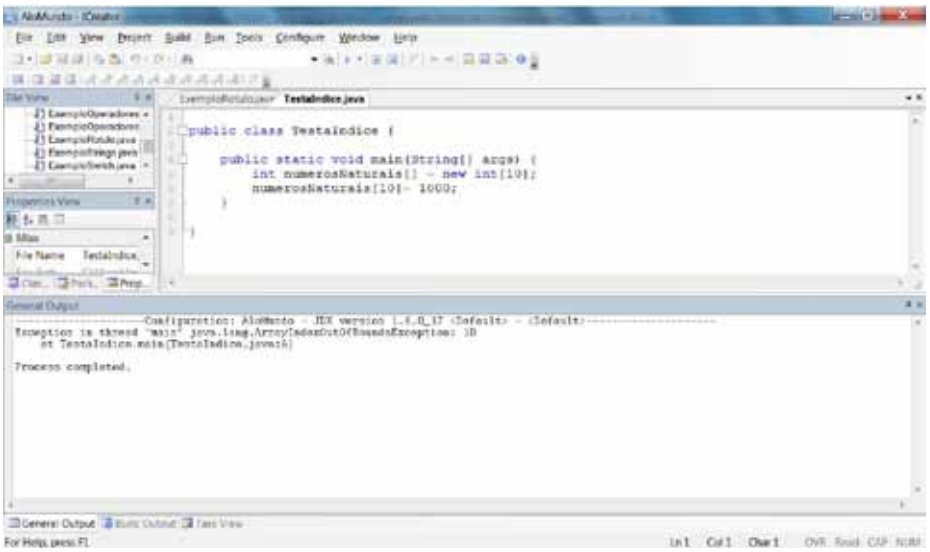

Figura 30 - Estouro no acesso a um índice de um vetor

A execução desta classe ocasionará um erro em sua execução, pois há uma tentativa de se ler um índice que não existe no vetor `numerosNaturais`. Os índices válidos para este vetor vão de 0 até 9. Como visto na Figura 48, a execução deste programa deverá apresentar uma mensagem de erro.

Erros em Java são apresentados como exceções. Neste caso a exceção `ArrayIndexOutOfBoundsException` indica justamente a tentativa de se acessar um índice fora dos limites de um determinado vetor. Estes limites refletem a fronteira dos índices válidos para o vetor. Para o vetor `numerosNaturais`, seriam valores menores ou iguais a 0 (zero) e maiores que 9 (nove).

Tendo esta regra na cabeça, o acesso a cada índice do vetor pode ser utilizado para se atribuir novos valores a cada elemento do vetor, como no exemplo da Figura 31.

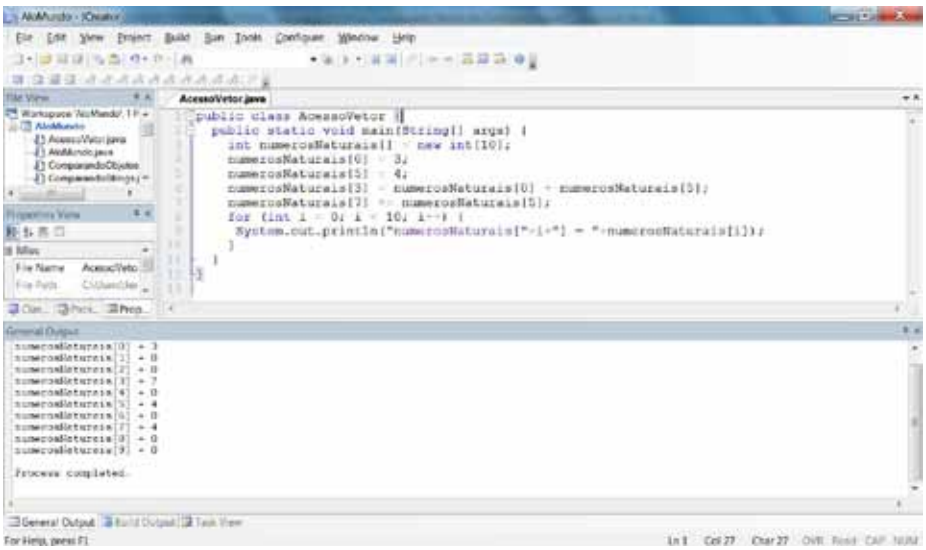

Figura 31 – Acessando dados de um vetor

3.2.4. Criando e atribuindo valores a matrizes em um único passo

Os passos apresentados previamente não são a única forma de se criar vetores em Java. Na realidade, é possível inclusive realizar a criação e atribuição de valores a cada elemento do vetor em uma única etapa. Para isto, em vez de utilizar o `new` para criar o vetor, os elementos devem ser associados diretamente no identificador do vetor, delimitando-os por chaves (`{}`) e separando-os por vírgula. Veja o exemplo da Figura 32, onde o vetor `numerosNaturais` é criado e tem os valores de seus valores definidos em um único passo.

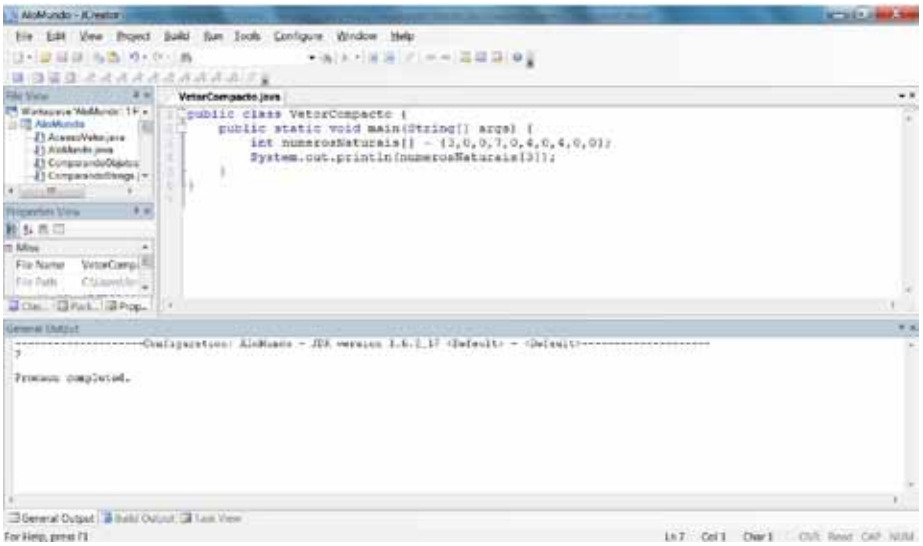

Figura 32 - Forma compacta de definição de um vetor

É importante ressaltar que assim como a maioria das linguagens, o tipo de um vetor restringe o tipo de dado que cada um de seus elementos pode assumir. Em outras palavras, não se pode ter um vetor que guarde elementos do tipo `String` e ao mesmo tempo armazene dados do tipo inteiro.

3.3 O ATRIBUTO LENGTH

Embora vetores sejam utilizados de maneira semelhante a várias linguagens de programação, existem algumas características específicas destas estruturas em Java. Na realidade, a grande diferença é que um vetor em java é um objeto, possuindo a ideia de referência e oferecendo métodos e atributos para o desenvolvedor.

Um destes atributos é o `length`. Este atributo retorna o tamanho de elementos que um vetor pode armazenar. Veja o exemplo do trecho de código na Figura 33.

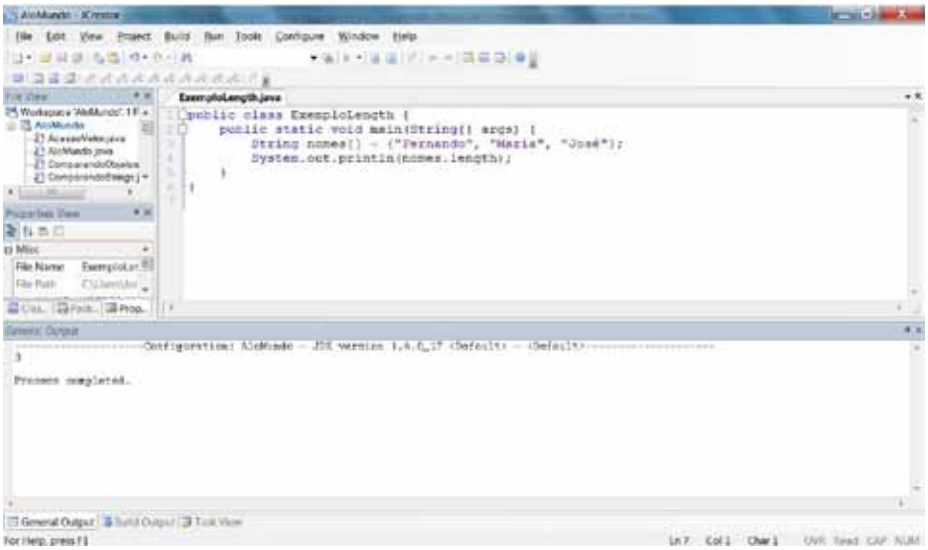

Figura 33 – Exemplo de acesso ao atributo length

Neste exemplo, ao se criar o vetor `nomes` e inserir valores strings para seus componentes, também se está definindo o tamanho máximo deste vetor e consequentemente o valor para o atributo `length`. O mesmo valeria para a definição convencional de vetores, como no exemplo:

```
float numerosPrimos[] = new float[1200];

// neste caso, numerosPrimos.length = 1200
```

O uso do atributo `length` é muito recomendado, principalmente para evitar efeitos colaterais na manutenção de programas. Imagine a seguinte a seguinte situação ilustrada no programa da Figura 34.

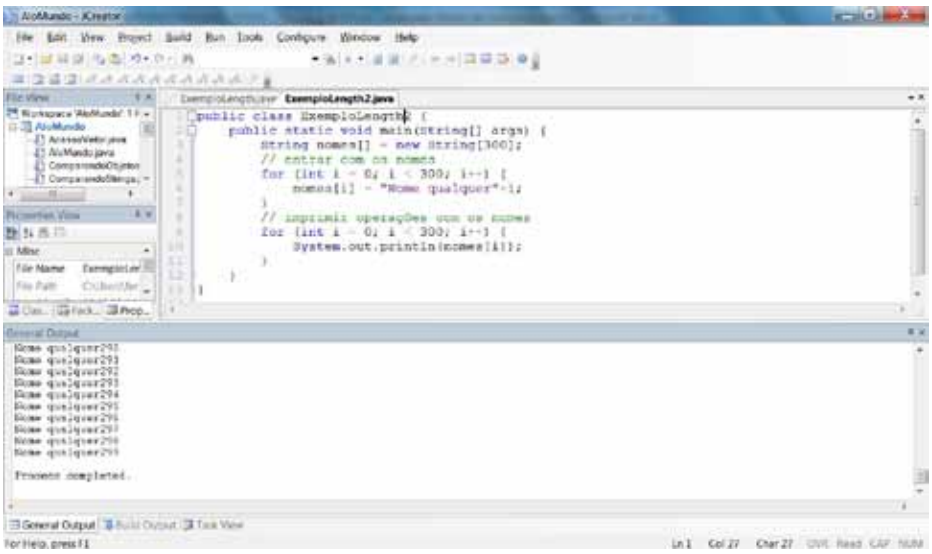

Figura 34 – Classe ExemploLength2 (versão 1)

Neste programa, o vetor nomes é definido como um vetor de 300 strings. Em dois pontos, são utilizados laços que percorrem todos os elementos do vetor para (i) atribuir valores e (ii) realizar operações com os valores de cada elemento do vetor. A princípio, nada de anormal. Porém, imagine que um dia, você resolva aumentar a quantidade de elementos do vetor. Você teria que sair mudando no código toda referência feita ao valor 300. Isso pode ser uma potencial fonte de problema. Pior ainda seria se em vez de aumentar, fosse necessário diminuir a quantidade de elementos do vetor. Se você esquecer algum laço com o valor antigo, você vai ter o erro `ArrayIndexOutOfBoundsException`, pois você tentará ler valores além do índice modificado. Agora veja esta nova versão da classe (Figura 35).

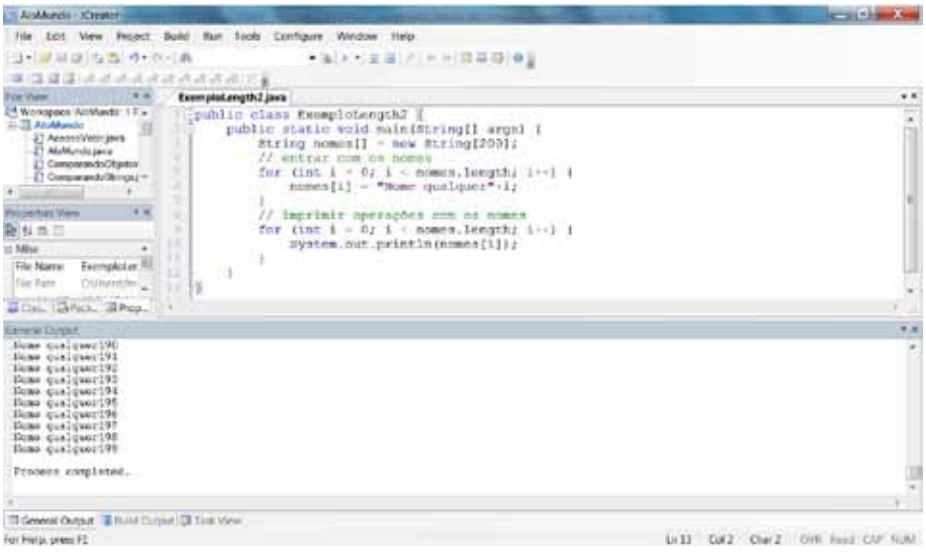

Figura 35 - Classe ExemploLength2 (versão 2)

Utilizando o atributo `length`, os laços serão sempre limitados pelo tamanho do vetor, facilitando a manutenção e diminuindo a propensão a erros.

Vale ressaltar que como em outras linguagens, vetores tem seu tamanho definido no momento de sua criação, independente de como esta criação seja realizada. Este tamanho é imutável. Logo, não é válida a tentativa de se atribuir um valor ao atributo `length` de um vetor, como `nomes.length = 310;`

Se por acaso necessita-se armazenar mais valores que a atual capacidade de um vetor, a solução é criar um novo vetor com maior capacidade, e copiar cada elemento para este novo vetor.

3.3.1. Matrizes (Vetores Multidimensionais)

Os exemplos apresentados anteriormente mostram apenas vetores que possuem uma única dimensão. Porém, em certas situações é interessante que os vetores possuam mais de uma dimensão. São os chamados vetores multidimensionais ou simplesmente matrizes. Imagine, por exemplo, se quiséssemos representar um acervo de livros por meio de um vetor de dados. Para este exemplo, imagine que os livros são dispostos em uma estante com 10 prateleiras, onde cada prateleira comporta 20 livros. Neste caso, vamos utilizar uma matriz de 10 linhas para representar as prateleiras, com 20 colunas por linha.

Para se definir uma matriz, o procedimento é semelhante à definição de um vetor unidimensional. Deve-se escolher o identificador e definir o tipo de dado que será guardado na matriz. A única diferença é que se deve usar mais de um par de colchetes (`[]`) para indicar o número de dimensões da matriz. Por exemplo, vamos utilizar strings para guardar o nome dos livros da estante. Então, a definição da matriz de strings é feita da seguinte forma:

```
String[][] estante;
```

Depois é necessário definir as dimensões da Matriz. Isto é feito de modo semelhante a um vetor, utilizando-se a palavra reservada e indicando a quantidade de elementos para cada dimensão. Para nosso exemplo teríamos:

```
estante = new String[10][20];
```

Note que da mesma forma como para um vetor, estes dois passos poderiam ser realizados em uma única etapa:

```
String[][] estante = new String[10][20];
```

A partir daí, deve-se então atribuir valores para cada elemento da matriz. O acesso a cada elemento é feito através de índices para cada dimensão da matriz. Por exemplo, se quisermos acessar o primeiro livro da primeira prateleira da estante, é equivalente a acessar o elemento da primeira linha e da primeira coluna. Como os índices em Java começam com zero, então seria equivalente a acessar o elemento de índices (zero,zero). Este mesmo raciocínio vale para as demais posições da estante. Veja alguns exemplos exemplo:

```
estante[0][0] = "Harry Potter";                // 1ªPrateleira 1, 1º livro
estante[0][1] = "Bíblia Sagrada";              // 1ªPrateleira 1, 2º livro
estante[3][2] = "Linguagem de Programação II"; // 4ªPrateleira 1, 3º livro
estante[5][9] = "Divina Comédia";              // 6ªPrateleira 1, 10º livro
```

O programa da Figura 36 mostra estes conceitos reunidos.

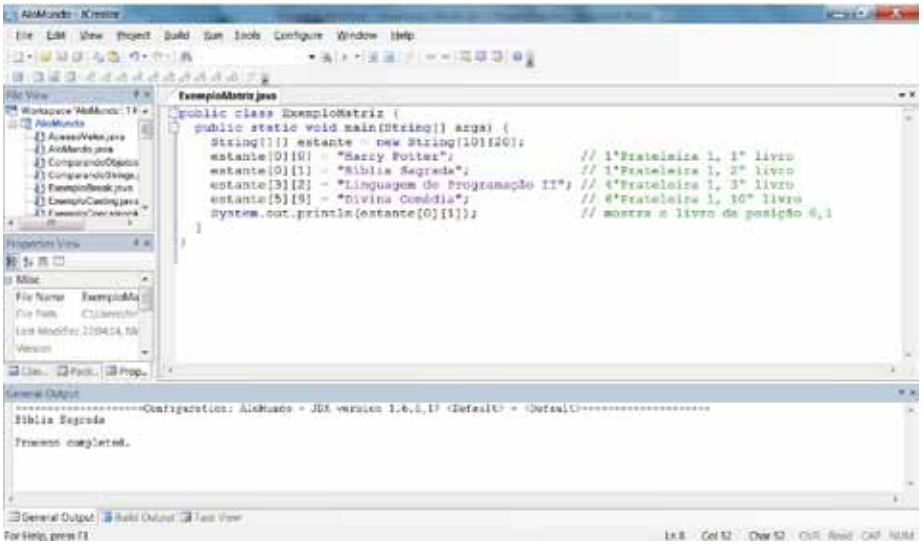

Figura 36 – Exemplo de um vetor multidimensional

Importante ressaltar que da mesma forma como vetores unidimensionais, é possível definir os valores de uma matriz em único passo. Neste caso, você deve pensar na matriz como um "vetor de vetores", Neste caso, existe um aninhamento entre os vetores na definição da matriz. Veja como seria a definição de uma matriz usando uma abordagem mais compacta (Figura 37).

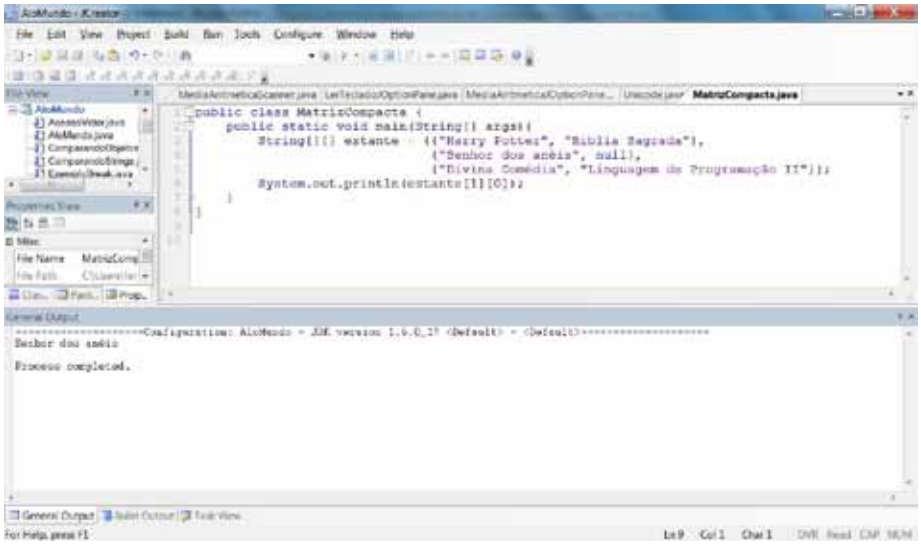

Figura 37 -Definição de Matriz (forma compacta)

Perceba que um dos elementos de um vetor é definido como `null`. Isto é totalmente possível e se assemelha a ideia que o vetor foi definido, e nenhum foi até então atribuído para aquela posição.

3.4 Entrada de Dados pelo Teclado

Até agora, os programas apresentados não tem permitido nenhuma interação do usuário com o programa. Os valores atribuídos a variáveis são definidos estaticamente na criação do programa. No caso de se mudar algum valor, esta mudança tem que ser feita diretamente no programa. Depois este deve ser recompilado para posteriormente ser

executado. De certo, é entediante criar programas onde não seja permitido, por exemplo, entrar com dados pelo teclado para verificar a execução de um mesmo programa com diferentes valores.

Uma abordagem mais prática seria permitir que estes valores fossem definidos a partir da entrada do usuário pelo teclado. Vamos ver nesta seção como tal ação é possível em Java.

Para falar a verdade, ler dados a partir do teclado em Java não é algo tão simples quanto em outras linguagens de programação. Java privilegia interfaces gráficas baseadas em janelas, com metáforas como caixas de texto, botões, etc. Com isso, a leitura a partir de uma interface textual, como a partir de uma janela DOS não é algo tão trivial.

Porém, para o propósito de aprendizado que pretendemos nesta disciplina, vamos ver formas que permitirão que você interaja com os programas que você crie a partir de agora, para tornar nossos programas mais interessantes e interativos.

Alguns conceitos que serão apresentados nesta seção serão apresentados um tanto quanto superficialmente. Porém, isto será feito propositadamente, uma vez que tais conceitos são assuntos futuros neste curso e que serão aprofundados posteriormente em um momento adequado.

Existem três abordagens para leitura de dados a partir do teclado para programas baseados em interfaces textuais (DOS). Todas elas utilizam classes fornecidas pelo JDK. São elas:

- Uso da classe `BufferedReader`;
- Uso da classe `Scanner`;
- Uso da classe `JOptionPane`;

Vamos a seguir falar sobre cada uma destas abordagens.

3.4.1. Lendo a partir da classe BufferedReader

Java oferece uma série de classes para os mais diferentes propósitos. Estas classes são agrupadas de acordo com tais propósitos. Estes agrupamentos são chamados de pacotes. Um destes pacotes trata justamente de questões de entrada e saída de dados, como a leitura de dados de um arquivo ou de uma conexão de rede. Dentro deste pacote existem três classes que podem ser utilizadas para capturar a entrada de dados pelo teclado. São estas:

- `java.io.BufferedReader`
- `java.io.InputStreamReader`
- `java.io.IOException`

Vamos ver como estas classes são utilizadas através de um exemplo, listado no programa `LerTecladoBufferedReader` (Figura 38).

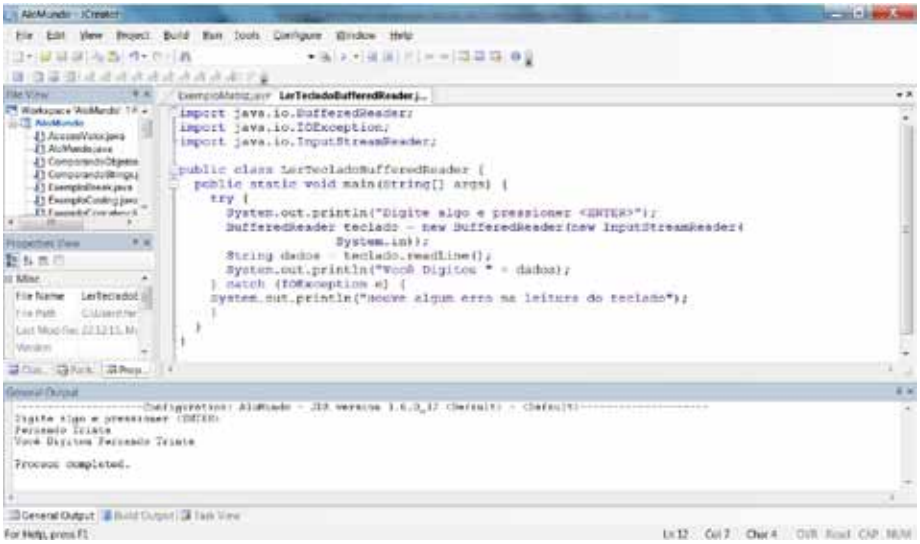

Figura 38 – Leitura do teclado usando a classe BufferedReader

Este programa pede que o usuário digite alguma coisa pelo teclado e depois pressione a tecla ENTER, indicando a entrada de um dado. Este dado é depois apresentado pelo programa.

Como citado anteriormente, as três classes utilizadas por este programa fazem parte de um pacote de classes relacionadas com a entrada de dados. Este pacote é referenciado dentro da linguagem Java como o pacote `java.io`. Se um programa precisa usar uma classe de algum pacote fornecido pela linguagem, isto deve explicitamente ser declarado no início do programa. É justamente isso que as três primeiras linhas estão fazendo:

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

Elas devem ser obrigatoriamente postas no início da definição de sua classe/programa, e não importa a ordem entre elas. Fazendo isto, você está habilitando o uso de qualquer uma dessas classes em qualquer parte de seu programa.

Com isso, dentro do método `main`, é necessário criar um objeto que utilize estas classes. É justamente o que a linha abaixo faz:

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

Esta linha cria um objeto com o identificado `teclado` e do tipo `BufferedReader` que servirá para capturar caracteres obtidos a partir da entrada padrão `(System.in)` do sistema. A entrada padrão do sistema é o teclado, mas você poderia usar aqui uma referência para um arquivo de texto, por exemplo.

Objetos do tipo `BufferedReader` permitem que sejam utilizados vários métodos para leitura de dados, mas para nós basta um deles: o `readLine`. Este método retorna uma string com os dados obtidos a partir de uma entrada, e é justamente o que acontece na linha:

```
String dados = teclado.readLine();
```

Os dados digitados pelo usuário são retornados pelo método e jogados na variável `dados`. Esta variável é depois utilizada para apresentar uma mensagem na tela.

Por fim, note que o método `main` possui praticamente todo seu corpo instruções englobado dentro de um bloco que começa com a palavra-reservada `try` e termina com outra palavra-reservada, `catch`. Estas palavras estão relacionadas com a forma de tratamento de erros em Java. Na linguagem, possíveis problemas na execução de um trecho de programa são chamados de exceções. Ler dados a partir de uma fonte de informações pode ocasionar erros. Por exemplo, se o usuário não digitar nada e pressionar ENTER, para o programa listado, isto é visto como uma exceção, pois não vai haver dados de retorno para o método `readLine`. Exceções são o último tópico de nosso curso. Para nossas pretensões de ler dados a partir do teclado, entenda apenas o seguinte: quando houver a possibilidade de um erro em um trecho de programa é necessário usar um bloco `try/catch` e dizer o que se deve fazer caso o erro ocorra. É justamente o que nosso programa está fazendo. Caso ocorra uma exceção de entrada ou saída (`IOException`), este erro é capturado e uma rotina de tratamento deste erro é executada. No caso do nosso exemplo, é simplesmente apresentada uma mensagem de aviso ao usuário.

Vale ressaltar que apesar do método retornar apenas valores string, isto não impede que se construa programas que simulem a entrada de valores números, booleanos, dentre outros. Para isto, devemos usar das classes *Wrapper*, aprendidas no primeiro capítulo. A classe (Figura 39) abaixo calcular a média aritmética de dois números fracionários.

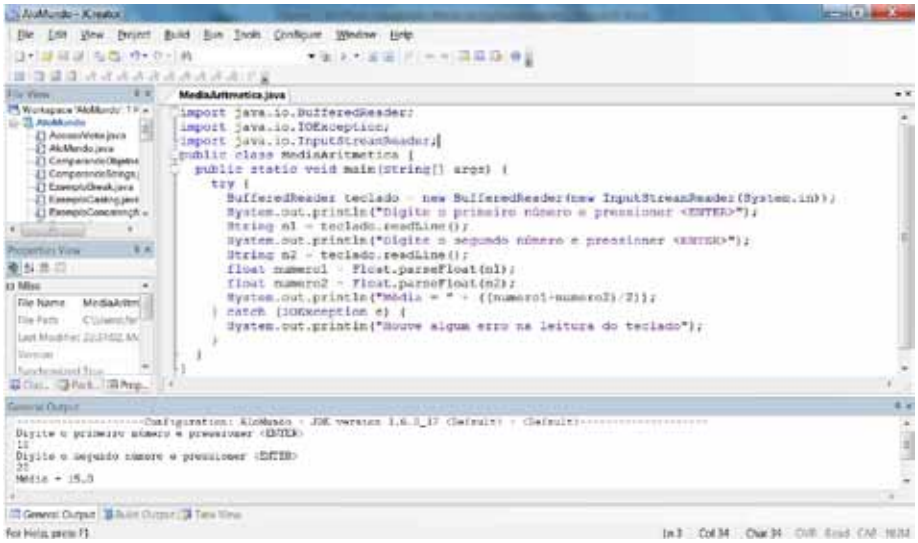

Figura 39 - Usando a classe BufferedReader para calculo da média aritmética de dois números

3.4.2. Lendo a partir da classe Scanner

Como você pode ter percebido, a leitura a partir do teclado não é trivial. A partir da versão 5.0 da plataforma Java, uma nova forma para permitir a obtenção de dados a partir de uma fonte de informações foi apresentada. Esta proposta utiliza a classe Scanner, que faz parte do pacote de classes utilitárias da plataforma Java (pacote `java.util`) e portanto para ser utilizada deve ser importada.

De modo semelhante à abordagem anterior, vamos ver um exemplo de um programa que utiliza a classe Scanner para ler dados de um teclado na Figura 40.

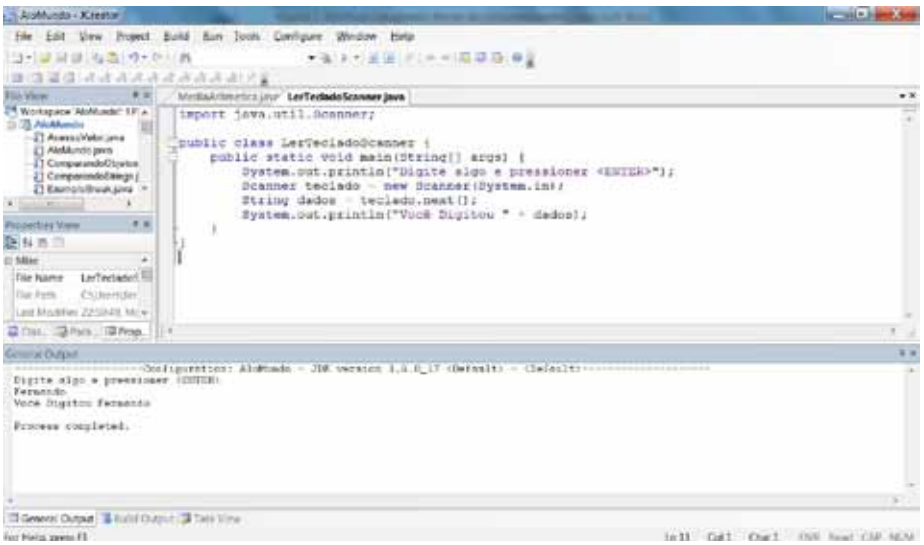

Figura 40 – Leitura do teclado usando a classe Scanner

Assim como para a classe `BufferedReader`, é obrigatório que a classe `Scanner` seja importada. Isso é feito com a primeira linha do programa:

import java.util.Scanner;Depois, um objeto do tipo `Scanner` é declarado. A definição deste objeto recebe como parâmetro a entrada padrão do sistema `System.in`.

```
Scanner teclado = new Scanner(System.in);
```

A classe `Scanner` oferece um conjunto de métodos para leitura de diferentes tipos de dados, que são apresentados na Tabela 10. Você pode utilizar qualquer um destes métodos em seus programas.

Tabela 10 - Métodos da classe Scanner para leitura de dados

Método	Finalidade
next()	Aguarda a entrada em formato String
nextInt()	Aguarda a entrada em formato Inteiro e utiliza o tipo de retorno int
nextByte()	Aguarda a entrada em formato Inteiro e utiliza o tipo de retorno byte
nextLong()	Aguarda a entrada em formato Inteiro Longo e utiliza o tipo de retorno long
nextFloat()	Aguarda a entrada em formato Fracionário e utiliza o tipo de retorno float
nextDouble()	Aguarda a entrada em formato Fracionário e utiliza o tipo de retorno double

Veja na Figura 41, a versão do programa de calculo da média aritmética utilizando a classe Scanner.

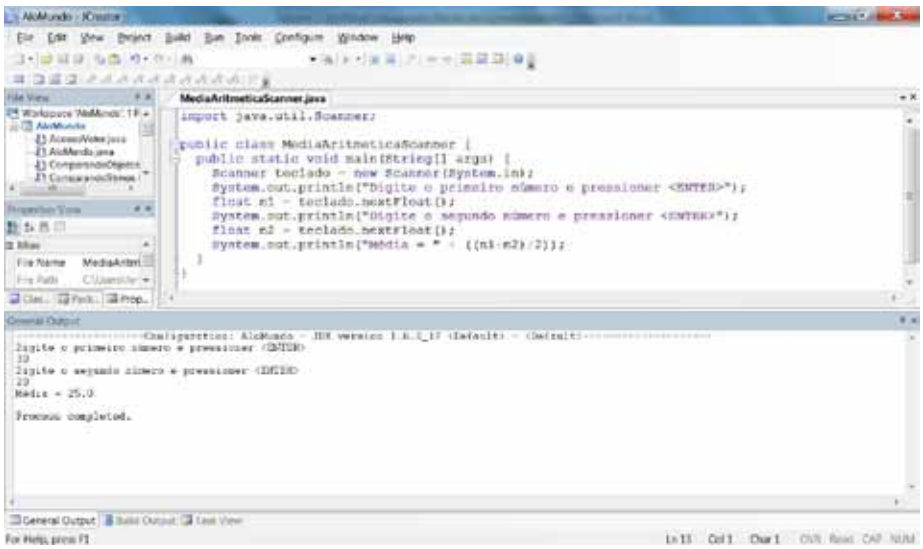

Figura 41 - Usando a classe Scanner para calculo da média aritmética de dois números

Esta abordagem reduz o número de linhas necessárias, a necessidade de conversões a partir de uma string e a necessidade do tratamento de exceções. Por tais razões, é uma abordagem mais recomendada.

3.4.3. Lendo a partir da classe JOptionPane

Uma última possibilidade para ler dados do teclado é o uso de caixas de diálogo pré-definidas em Java. A classe `JOptionPane` faz parte de um pacote com várias janelas de interação e alertas, e que permite que o usuário forneça dados. A Figura 42 apresenta um exemplo da exibição de um caixa de diálogo criada com a classe `JOptionPane`.

Figura 42- Exemplo de uma caixa de diálogo

Vamos ver então como seria um programa para entrada de dados utilizando esta classe (Figura 43).

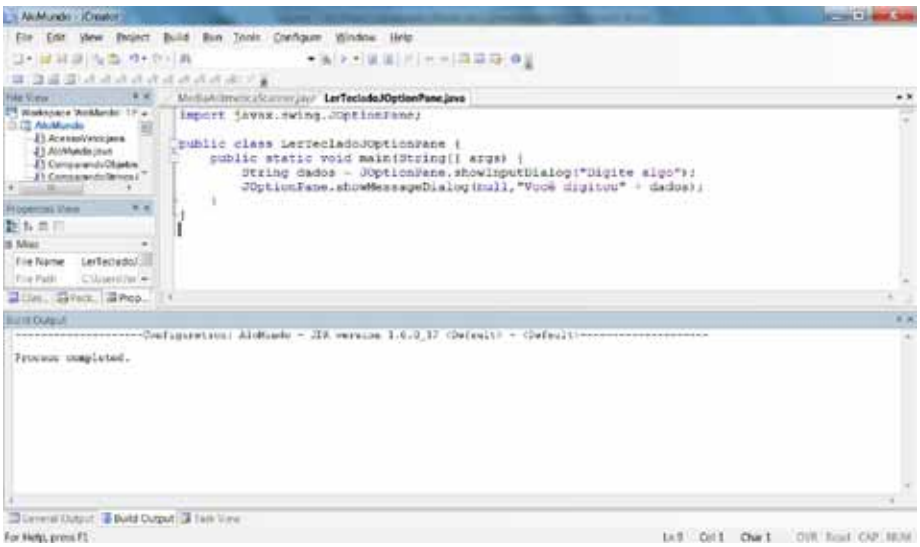

Figura 43 - Leitura do teclado usando a classe JOptionPane

Da mesma forma como as outras abordagens, o primeiro passo é a importação da classe. Isso é feito através da linha

```
String dados = JOptionPane.showInputDialog("Digite algo");
```

Depois, para se criar uma caixa de diálogo, é necessário utilizar o método *showInputDialog* da classe `JOptionPane`. Um aspecto importante é que este método é estático à classe. Em outras palavras, não é necessário criar um objeto para se chamar esta operação, ela vale para qualquer objeto do tipo `JOptionPane`. Por isso, este método também pode (na realidade, é recomendado) ser chamado a partir do próprio nome da classe. Este método recebe como parâmetro uma string com a mensagem a ser apresentada na caixa de diálogo, e devolve como parâmetro, a string com os dados digitados pelo usuário. Se o usuário optar por usar o botão cancelar, o método retorna como resultado `null`.

A classe `JOptionPane` também fornece outro método que permite criar uma caixa de diálogo com uma mensagem específica. Este método é o *showMessageDialog*. Este métodos tem dois parâmetros: a janela ao qual a caixa de diálogo está vinculada e a mensagem a ser apresentada. No nosso caso, não há janela de vínculo e portanto é passado como parâmetro `null`. A Figura 44 mostra um exemplo de uma caixa de diálogo apresentando uma mensagem.

Figura 44 - Apresentando uma mensagem em uma caixa de diálogo

Da mesma forma como a classe `BufferedReader`, o método *showInputDialog* devolve apenas strings, sendo necessária a conversão através de classes *Wrappers* para outros tipos de dados, como na versão abaixo (Figura 45) para o cálculo da média aritmética de dois números fracionários.

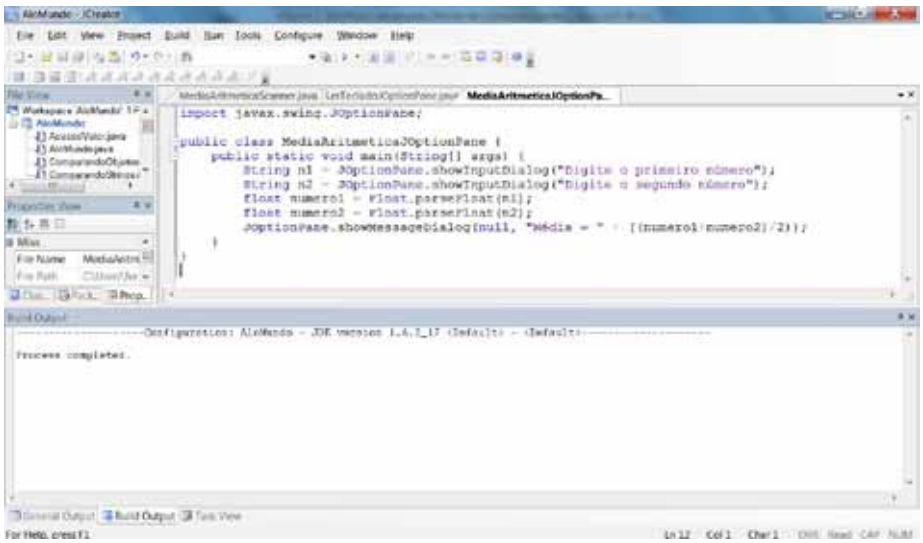

Figura 45 – Usando a classe JOptionPane para calculo da média aritmética de dois números

A Figura 46 apresenta as caixas de diálogo durante a execução do programa.

Figura 46 - Caixas de Diálogo durante a execução da classe MediaAritmeticaJOptionPane

Exercícios

1. Implemente um programa em Java que peça um número como entrada. Fornecido este número, deve ser definido um vetor de inteiros e o usuário deve fornecer valores para cada elemento do vetor. O programa deve fornecer: a soma de seus elementos, o maior elemento e a média aritmética de seus elementos.
2. Implemente um programa em Java que peça um número como entrada. Fornecido este número, deve ser definido dois vetores: um de strings e outro de notas, com o número de elementos igual ao definido pelo número fornecido pelo usuário. Crie um laço em permite que o usuário entre com dados para cada elemento de cada um dos vetores. Depois, o programa deve perguntar o nome de um aluno. De acordo com o nome fornecido deve ser apresentada a nota deste aluno. Esta consulta deve se repetir até que se forneça a palavra “FIM” como nome do aluno.
3. Crie um programa em Java o qual permita que o usuário entre com o nome de um aluno, e suas três notas. Para cada aluno, deve ser apresentada a média aritmética

de suas notas. Este procedimento deve ser repetido até ser fornecido como nome do aluno do aluno a palavra “FIM”. Após este procedimento, deve ser informado, respectivamente o nome e a média dos alunos com a melhor e a pior média dentre os valores informados.

4. Modifique o programa anterior, perguntando primeiramente o número de alunos que vão ser cadastrados. Após o cadastramento, imprima a lista dos alunos e suas médias em (i) ordem crescente de suas médias e depois (ii) em ordem alfabética.

5. Faça um programa que leia dois vetores de inteiros. Para cada vetor pergunte o número de elementos e depois entre com cada elemento do vetor. Somente após estas entradas, devem ser apresentados quais são os elementos que fazem parte dos dois vetores (vetor interseção).

6. Implemente um programa em Java que realize a soma de duas matrizes de inteiros que possuam o mesmo número de linhas e colunas. O programa deve perguntar qual o número de linhas e colunas das matrizes, e depois permitir que o usuário entre com os valores de cada elemento da matriz. Depois deve ser apresentado o resultado da matriz soma destas matrizes.

7. Considere a seguinte matriz de distâncias (KM) entre capitais.

	Recife	Fortaleza	São Luis	Natal	Teresina
1.Recife	-	800	1600	300	1100
2.Fortaleza	800	-	1100	550	630
3.São Luis	1600	1100	-	1600	450
4.Natal	300	550	1600	-	1170
5.Teresina	1100	630	450	1170	-

Crie um programa que permita que, a partir destes dados, o usuário forneça um trajeto. Exemplo: 1 , 3, 5 entende-se por saída de Recife, escala em São Luís e chegada em Teresina. E que, a partir destas entradas, se forneça a distância percorrida no trajeto.

Vamos Revisar?

Neste último capítulo, você aprendeu a trabalhar com estruturas mais complexas para agrupar dados e tratá-los como um conjunto, chamados comumente de vetores. Você também aprendeu a entrar com dados a partir do teclado, o que lhe permite a criação de programas mais interativos. No próximo volume, você vai utilizar estes conceitos para finalmente criar seus objetos e classes.

CONHEÇA OS AUTORES

Fernando Trinta

Sou professor de ciência de computação, formado pela Universidade Federal do Maranhão. Tenho Mestrado e Doutorado em Ciência da Computação pelo Centro de Informática da Universidade Federal de Pernambuco, com ênfase na área de Sistemas Distribuídos. Durante minha pós-graduação, estive envolvido com os temas de objetos distribuídos e educação à distância no Mestrado, e jogos digitais, *middleware* e computação ubíqua no Doutorado. Trabalhei no desenvolvimento de sistemas em várias empresas, privadas e públicas. Atualmente faço parte do corpo discente do Mestrado em Informática Aplicada da Universidade de Fortaleza (UNIFOR), no Ceará. Além da informática, gosto muito de esportes em geral e cinema. Mas nos últimos anos, duas novas paixões tomaram conta do meu mundo: Ian e Ananda.