E-book (Warm Up)

Primeiros passos com
a linguagem Java

v1.0

algaworks

Sumário

Capítulo 1

Introdução

Fala, mergulhador!

Se você ainda não está 100% confortável com a linguagem Java, neste e-book eu vou te ajudar a entender melhor os fundamentos da linguagem para que você consiga acompanhar o curso **Mergulho Spring REST (MSR)**, que vai acontecer de **16 a 23 de maio de 2022**.

Eu já estou considerando que você tem o JDK instalado, beleza?

Caso você não tenha o JDK na sua máquina, baixe o *workbook* de preparação de ambiente que já disponibilizei e siga o passo a passo.

Vamos lá?

1.1. Escrevendo seu primeiro código Java

Vamos criar o nosso primeiro programa, o famoso "Olá Mundo".

Este primeiro programa será importante para você aprender ou relembrar a estrutura básica que usaremos nos próximos exemplos, o processo de compilação e execução e também para resolver erros comuns de principiantes.

Neste e-book nós não usaremos qualquer IDE. A ideia é que você escreva os códigos sem ajuda de uma ferramenta. Você pode usar Bloco de Notas, Sublime Text ou qualquer outro editor de texto.

Para começar, crie uma pasta em qualquer local do seu computador, para armazenar os arquivos que você vai escrever para seguir este e-book.

Agora, abra o seu editor de texto e digite o código-fonte abaixo:

```
public class OlaMundo {

    public static void main(String[] args) {
        System.out.println("Olá mundo");
    }

}
```

Não se preocupe com o significado de cada coisa que escreveu. Por enquanto queremos apenas que funcione!

Você deve digitar o código **exatamente** como está no exemplo, inclusive letras maiúsculas e minúsculas, pois a linguagem Java é *case-sensitive*.

Quando terminar de digitar tudo, salve o arquivo dentro da pasta que você criou com o nome `OlaMundo.java`.

Preste atenção novamente às letras maiúsculas e minúsculas do nome do arquivo e também na extensão, que deve ser `.java`.

1.2. Compilando e executando

Agora vamos compilar nosso primeiro programa. O processo de compilação é o que transforma o código-fonte em *bytecode*, que é um tipo de codificação que só a JVM (Máquina Virtual Java) consegue interpretar.

Entre no prompt de comando do Windows ou terminal no Linux e macOS, acesse a pasta onde você salvou o arquivo `OlaMundo.java` e digite o comando:

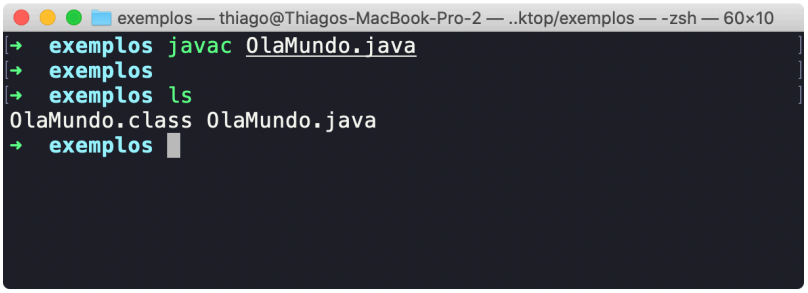
```
javac OlaMundo.java
```

O `javac` é o programa do JDK responsável por compilar um arquivo com código-fonte Java. Se funcionar, o programa ficará silencioso (não aparecerá nenhuma mensagem de sucesso).

Se alguma coisa der errado, você ficará sabendo, pois surgirão mensagens de erro no terminal.

Para confirmar se o arquivo foi compilado, você pode listar todos os arquivos da pasta usando o comando `dir` ou `ls`, dependendo do seu sistema operacional.

Se um novo arquivo chamado `OlaMundo.class` aparecer, é porque você teve sucesso ao compilar seu primeiro programa.

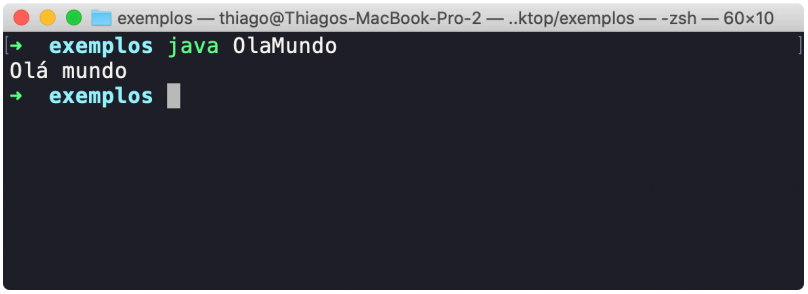


O arquivo com extensão `.class` é o *bytecode* gerado (executável). Se você for curioso, poderá tentar abri-lo usando um editor de texto. Mas como já disse, só a JVM consegue interpretá-lo.

Agora que você já tem o programa compilado, para executá-lo, digite o comando:

```
java OlaMundo
```

Preste atenção novamente para as letras maiúsculas e minúsculas.



Se funcionar, você deve ver a mensagem "Olá mundo" no seu terminal, como na imagem acima.

Apenas para ter certeza que você entendeu, o que acabamos de executar no

último passo foi o arquivo `OlaMundo.class`, mas veja que não precisamos especificar a extensão para executá-lo.

Você poderia ter apagado ou movido o arquivo `OlaMundo.java` para outro lugar e mesmo assim a execução aconteceria com sucesso, porque nesse momento apenas o *bytecode* é lido e executado.

Uma curiosidade: a partir do JDK 11 nós podemos executar um programa Java a partir do seu código-fonte, ou seja, sem precisar gerar um `.class` antes, dessa forma:

```
java OlaMundo.java
```

Note que agora estamos especificando o nome do arquivo de verdade, incluindo a extensão dele.

Mas essa é uma facilidade apenas para estudos e pequenos programas, como é o nosso caso aqui. De qualquer forma, achei importante você entender o processo de compilação e execução de forma separada.

1.3. Entendendo o que foi codificado

Agora vamos estudar um pouco sobre o que codificamos no primeiro programa.

A primeira linha do arquivo declarou um nome de classe.

O termo "classe" vem da orientação a objetos, que não é o foco deste e-book. Neste contexto, entenda como a declaração de um "programa", ou seja, criamos o programa `OlaMundo`.

A abertura e fechamento das chaves indicam um bloco de código. Tudo que estiver lá dentro pertence ao programa `OlaMundo`.

```
public class OlaMundo {
}
```

Dentro do bloco de código do programa, criamos um método chamado `main`. Um método é um bloco de código que só roda quando ele é chamado.

O método `main` é necessário para que nosso programa seja executado quando digitamos o comando `java OlaMundo`. Este método é o ponto de entrada do programa, por isso ele será executado automaticamente quando solicitarmos a execução do programa.

O nome do método não pode ser alterado, porque apesar de não pertencer à sintaxe da linguagem, é um padrão que significa um ponto inicial de um programa desenvolvido.

O método `main` recebe um argumento do tipo *array de strings*, declarado como `String[] args`.

Basicamente, temos acesso aos argumentos passados na execução do programa a partir de `args`.

O bloco de código delimitado pelas chaves deve possuir uma ou mais linhas com a programação do que o sistema deve fazer.

```
public static void main(String[] args) {
}
```

Em nosso exemplo, nosso programa apenas imprime "Olá mundo" na tela. Para fazer isso, usamos o método `System.out.println`.

Todo texto (string) em Java é delimitado por aspas duplas, e toda instrução (comando) deve terminar com um ponto e vírgula.

Note também que o texto "Olá mundo" está entre parênteses, que indica o início e término de um parâmetro do método `println`.

```
System.out.println("Olá mundo");
```

O nome do arquivo precisou ser exatamente `OlaMundo.java`. Em Java, o nome do programa (classe) deve coincidir com o nome do arquivo, portanto, se a sua classe tivesse o nome `VouMergulhar`, o seu arquivo com o código-fonte deveria ter o nome `VouMergulhar.java`.

1.4. Erros comuns dos marinheiros de primeira viagem

Marinheiros de primeira viagem costumam cometer erros comuns, porque a linguagem Java é um pouco burocrática. Vejamos alguns erros que você pode cometer e como corrigi-los:

1. Ao compilar, aparece o erro:

```
OlaMundo.java:3: error: cannot find symbol
    public static void main(string[] args) {
                       ^
  symbol:   class string
  location: class OlaMundo
```

Você digitou `string` com a letra "s" em minúsculo. O correto é:

```
public static void main(String[] args)
```

2. Ao compilar, aparece o erro:

```
OlaMundo.java:4: error: package system does not exist
        system.out.println("Olá mundo");
```

Você digitou `system` com a letra "s" em minúsculo. O correto é:

```
System.out.println("Olá mundo");
```

3. Ao compilar, aparece o erro:

```
OlaMundo.java:4: error: ';' expected
        System.out.println("Olá mundo")
                                       ^
```

Você esqueceu de finalizar a instrução com um ponto e vírgula. Veja:

```
System.out.println("Olá mundo");
```

4. Ao compilar, aparece o erro (ou algo parecido):

```
OlaMundo.java:1: error: class OiMundo is public, should be
declared in a file named OiMundo.java
public class OiMundo {
       ^
```

Você mudou o nome da classe, mas não mudou o nome do arquivo adequadamente, ou esqueceu de colocar as iniciais do nome do programa em letras maiúsculas.

5. Ao executar, aparece o erro (ou algo parecido):

```
Error: Could not find or load main class olamundo
Caused by: java.lang.NoClassDefFoundError: OlaMundo (wrong name: olamundo)
```

Você digitou o nome do programa sem levar em consideração as letras maiúsculas e minúsculas. Digite o comando `java OlaMundo` com as iniciais do nome do programa usando letras maiúsculas.

1.5. Adicionando comentários

Comentários são textos que podem ser incluídos no código-fonte, normalmente para descrever como determinado programa ou bloco de código funciona.

Os comentários são ignorados pelo compilador, por isso eles não modificam o comportamento do programa. Em Java, você pode comentar blocos de códigos inteiros ou apenas uma linha.

Para comentar uma única linha, faça como no exemplo a seguir:

```
public class OlaMundo {

    public static void main(String[] args) {
        // imprime uma mensagem na saída padrão
        System.out.println("Olá mundo");
    }

}
```

Para comentar um bloco de código, use `/* */` para abrir e fechar. Veja um exemplo:

```
public class OlaMundo {

    public static void main(String[] args) {
        /*
```

```
        Esta linha será ignorada pelo compilador.
        System.out.println("Esta instrução será ignorada também");
        E esta linha também.
        */
        System.out.println("Olá mundo");
    }

}
```

1.6. Sequências de escape

Sequências de escape são combinações de caracteres iniciadas por \ (contra barra) e usadas para representar caracteres de controle, aspas, quebras de linha, etc.

Para ficar melhor entendido, vamos fazer um teste. O exemplo abaixo tenta imprimir a mensagem *Oi "Maria"* (com o nome "Maria" entre aspas duplas).

```
public class ExemploEscape {

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

}
```

Ao tentarmos compilar, temos um belo erro (não tão bonito assim):

```
ExemploEscape.java:4: error: ')' expected
        System.out.println("Oi "Maria"");
                                ^
ExemploEscape.java:4: error: ';' expected
        System.out.println("Oi "Maria"");
                                      ^
2 errors
```

O problema aqui é que o compilador Java não conseguiu entender o que significa a palavra "Maria".

Se você prestar atenção, notará que a segunda aspa fechou a primeira, e a quarta fechou a terceira logo após o nome "Maria". Você percebeu que não existem aspas envolvendo o nome "Maria"?

E se quisermos dizer ao compilador que “Maria” é um texto, mas que as aspas que envolvem o nome também são textos? É aí que usamos uma sequência de escape.

```
System.out.println("Oi \"Maria\"");
```

A sequência `\"` diz ao compilador que a aspa deve ser considerada como um texto, e não como um delimitador de String. Agora nosso programa deve compilar e rodar normalmente.

Ao executar nosso programa, as aspas que envolvem o nome aparecem na saída padrão:

```
Oi "Maria"
```

Existem várias outras sequências de escape, sendo que as mais conhecidas são:

- `\n` para nova linha
- `\\` para uma barra invertida
- `\"` para aspas duplas

1.7. Palavras reservadas

As palavras reservadas são palavras-chave especiais, que têm significados para o compilador, que as usa para determinar o que seu código-fonte está tentando fazer.

Você não poderá usar as palavras reservadas como identificadores (nomes) de classes, métodos ou variáveis. As palavras-chave reservadas estão listadas a seguir:

abstract	boolean	break	byte	case	catch	char
class	const	continue	default	do	double	else
extends	final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long	native

new	package	private	protected	public	return	short
static	strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while	assert
enum						

Você não precisa memorizar agora todas essas palavras-chave, mas apenas ter ciência que elas existem para propósitos específicos e portanto você não pode usá-las para outros objetivos.

1.8. Convenções de código

Ao programar em Java, encorajo você a **sempre** usar as convenções de codificação da linguagem adotadas pelo mercado.

As vantagens em utilizar as convenções é que, além de você programar usando um padrão semelhante em todo o mundo, também facilita a manutenção no futuro por outros desenvolvedores e até mesmo por você, pois aumenta a legibilidade do código-fonte.

Você vai notar que todos os conteúdos da AlgaWorks, incluindo os conteúdos de aquecimento, aulas do **Mergulho Spring REST** e do **Especialista Spring REST** (nosso curso imersivo) sempre estarão usando convenções de código.

Infelizmente, nem todos os conteúdos que você vai encontrar na internet se preocupam com esse "detalhe" tão importante, por isso recomendo que você fique alerta para ser crítico em relação a isso e não aprender da "forma errada".

Um exemplo de uma convenção muito comum, praticamente uma regra, é que os nomes de suas classes devem ser escritos em *CamelCase*.

Isso significa que as palavras em uma frase que nomeiam sua classe devem ser iniciadas com letras maiúsculas e unidas sem espaços.

Por exemplo, se você tiver um programa responsável por gerar notas fiscais,

um nome válido seria `GeradorNotaFiscal`, mas não seria correto, de acordo com esta prática, usar os nomes `Geradornotafiscal`, `geradornotafiscal` ou `Gerador_Nota_Fiscal`.

Capítulo 2

Variáveis

Em linguagens de programação, variáveis são nomes simbólicos dados a informações alocadas na memória do computador.

As variáveis são definidas, atribuídas, acessadas e calculadas através do código-fonte do programa. Durante a execução de um programa, os conteúdos das variáveis podem mudar através de algum processamento.

Em Java, as variáveis devem ser declaradas com um tipo fixo para serem usadas. Isso quer dizer que, uma variável do tipo inteiro, por exemplo, não poderá ser alterada para um tipo real (decimal).

2.1. Declarando variáveis

Para começar, vamos declarar uma variável chamada `quantidade`, do tipo `int`. O tipo int é capaz de armazenar apenas valores inteiros negativos ou positivos. Veja:

```
int quantidade; // declarando variável inteira
```

A variável acima foi apenas declarada, isso quer dizer que não atribuímos nenhum valor para ela. Para fazermos isso, usamos o operador = (igual) seguido por um número inteiro.

```
quantidade = 10; // atribuindo o valor 10
```

Agora a variável `quantidade` possui o valor `10`. Se precisarmos alterar o valor da variável `quantidade`, podemos atribuir um novo valor a ela. Vamos dizer que a variável deve possuir o valor `15`.

```
quantidade = 15; // atribuindo o valor 15
```

É muito comum precisarmos mostrar o valor de uma variável na tela do usuário. Para fazermos isso de uma forma bem simples, podemos usar `System.out.println`, passando como parâmetro o nome da variável.

```
// imprimindo o valor da variável
System.out.println(quantidade);
```

Veja como ficou este exemplo completo. Incluímos mais uma instrução de impressão da variável `quantidade` entre a atribuição do valor `10` e do valor `15` para podermos ver o valor da variável antes e depois de ser modificada.

```
public class ExemploVariaveis {

    public static void main(String[] args) {
        int quantidade;

        quantidade = 10;
        System.out.println(quantidade);

        quantidade = 15;
        System.out.println(quantidade);
    }

}
```

Se você quiser economizar uma linha de código, pode declarar e atribuir a variável na mesma linha, como no exemplo abaixo:

```
int quantidade = 10; // declarando e atribuindo
```

Veja como ficaria no exemplo completo:

```
public class ExemploVariaveis {

    public static void main(String[] args) {
        int quantidade = 10;
        System.out.println(quantidade);
```

```
        quantidade = 15;
        System.out.println(quantidade);
    }

}
```

2.2. Nomeando variáveis

As variáveis em Java podem conter letras, dígitos, _ (*underscore*) e $ (dólar), porém elas não podem ser iniciadas por um dígito e não podem ser palavras reservadas.

Veja alguns nomes de variáveis válidos:

```
int quantidade; // pode ser toda em letras minúsculas
int quantidade_alunos; // pode ter underscore
int QUANTIDADE; // pode ser toda em letras maiúsculas
int QuantidadeAlunos; // pode ter letras maiúsculas e minúsculas
int $quantidade; // pode iniciar com dólar
int _quantidade; // pode iniciar com underscore
int quantidade_alunos_nota_10; // pode ter dígitos
```

Agora alguns nomes de variáveis inválidos (que nem compila):

```
int 2alunos; // não pode iniciar com dígitos
int quantidade alunos; // não pode ter espaços
int new; // new é uma palavra reservada do Java
```

Apesar da linguagem suportar letras maiúsculas e minúsculas, *underscore*, dólar e dígitos nos nomes das variáveis, a convenção de código Java diz que elas devem ser nomeadas com a inicial em letra minúscula e as demais iniciais das outras palavras em letras maiúsculas. Veja alguns exemplos:

```
int quantidadeAlunos;
int quantidadeAlunosNota10;
int numeroDeAlunosAprovados;
int totalAlunosReprovados;
```

As declarações de variáveis abaixo estão corretas para o compilador, mas não estão de acordo com o padrão de código usado mundialmente, por isso, evite-as

a todo custo.

```
int quantidade_alunos;
int QuantidadeAlunosNota10;
int NUMERODEALUNOSAPROVADOS;
int Total_Alunos_Reprovados;
```

É uma boa prática escrever as palavras completas quando vamos declarar variáveis em Java. Abreviações devem ser usadas somente se forem muito conhecidas no domínio do negócio. Por exemplo, você deve **evitar**:

```
int qtAlu;
int quantAlunNt10;
int nAlunosAprov;
int totAlunosRep;
```

Essas regras não servem apenas para a linguagem Java, mas existe uma cultura muito forte entre os bons programadores Java que prezam pela clareza do código, e um nome de variável mal definido pode atrapalhar muito a legibilidade do código.

2.3. Tipos primitivos

Nos exemplos anteriores, vimos como usar variáveis para armazenar apenas valores inteiros. Agora vamos estudar como criar variáveis para armazenar valores do tipo ponto-flutuante (com casas decimais). Por exemplo, vamos declarar uma variável e atribuir um valor com o preço de um produto:

```
double precoProduto = 9.43;
```

O tipo `double` é capaz de armazenar valores reais, que possuem casas decimais, mas também pode armazenar números inteiros.

Outro tipo primitivo bastante usado é o `boolean`. O tipo booleano pode armazenar os valores verdadeiro ou falso (`true` ou `false`). Por exemplo:

```
boolean alunoMatriculado = true; // recebe valor "verdadeiro"
boolean clienteBloqueado = false; // recebe valor "falso"
```

Os valores literais `true` e `false` são palavras reservadas do Java. Não é válido

atribuir números a uma variável booleana. Não existe nenhuma referência entre um valor booleano e os números `0` e `1` ou qualquer outro número, como em outras linguagens que talvez você já conheça, por isso, o código abaixo é inválido e não compila:

```
boolean alunoMatriculado = 1; // não compila
boolean clienteBloqueado = 0; // não compila
```

Depois de aprender como usar tipos inteiros, reais e booleanos, vamos conhecer mais sobre outros tipos primitivos do Java.

Os tipos de dados básicos (tipos primitivos) da linguagem Java são: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` e `double`.

A tabela abaixo mostra os tipos primitivos e a capacidade de armazenamento de cada um.

Tipo	Tamanho (bits)	Menor valor	Maior valor
`boolean`	1	`false`	`true`
`char`	16	`0`	$2^{16} - 1$
`byte`	8	-2^{7}	$2^{7} - 1$
`short`	16	-2^{15}	$2^{15} - 1$
`int`	32	-2^{31}	$2^{31} - 1$
`long`	64	-2^{63}	$2^{63} - 1$
`float`	32	-	-
`double`	64	-	-

Como pode ver na tabela dos tipos primitivos do Java, o tipo `boolean` ocupa apenas 1 bit para armazenar um valor booleano, ou seja, este tipo precisa apenas de uma da menor unidade de armazenamento na computação.

Uma variável do tipo `char` ocupa 16 bits, ou seja, 2 bytes (cada byte tem 8

bits) para armazenar um valor que representa um caractere. Veja abaixo alguns exemplos de declaração e atribuição de variáveis do tipo `char`:

```
char turmaAluno1 = 'A';
char tipoCliente = '2';
char simbolo = '@';
System.out.println(turmaAluno1);
System.out.println(tipoCliente);
System.out.println(simbolo);
```

O tipo `char` não pode ser usado para armazenar um texto. Na verdade, este tipo é capaz de armazenar apenas um caractere.

Os tipos `byte` e `short` são tipos numéricos inteiros com uma capacidade de armazenamento menor que o tipo `int`.

Para saber exatamente o menor e maior número que cada tipo suporta, basta fazer o cálculo da potência, conforme apresentado na tabela. Por exemplo, o menor valor do tipo `byte` é -2^7, ou seja, `-128`, e o maior é $2^7 - 1$, ou seja, `127`.

O tipo `long` é um tipo numérico inteiro (longo) com capacidade de armazenamento superior ao tipo `int`. Para se ter uma ideia, o tipo `long` é capaz de armazenar o valor máximo igual a `9223372036854775807`, enquanto o `int` suporta até o número `2147483647`.

No exemplo abaixo, declaramos e atribuímos uma variável para armazenar o número total de habitantes da cidade de Uberlândia/MG. Esta variável poderia ser do tipo `int`, mas já que estamos falando de `long`, exageramos um pouco no exemplo e declaramos como um inteiro longo.

```
long populacaoUberlandia = 699097;
System.out.println(populacaoUberlandia);
```

O código acima funciona, não temos nenhum problema com ele. Agora veja o código abaixo:

```
long populacaoMundial = 7000000000; // não compila
System.out.println(populacaoMundial);
```

O último exemplo não compila! Pode parecer estranho, mas isso acontece porque o número `7000000000` não é compatível com o tipo `int`.

Espere… mas não estávamos usando um tipo `int`, e sim um `long`. Certo?

Sim e não, certo e errado! Quando atribuímos um valor literal (digitado manualmente no código-fonte) a uma variável do tipo `long`, o valor literal é por natureza do tipo `int`, ou seja, os literais numéricos inteiros são do tipo `int` por padrão, a não ser se usarmos um pequeno truque. Veja como é fácil:

```
long populacaoMundial = 7000000000L; // compila!
System.out.println(populacaoMundial);
```

A única diferença do último exemplo para o que não compila é a letra `L` após o número `7000000000`. Ao incluir a letra `L` ao final de um número literal, indicamos ao compilador que queremos que o número seja interpretado como um tipo `long`, e não um `int`.

Os tipos `float` e `double` são os únicos tipos ponto-flutuante que são capazes de armazenar números reais (com casas decimais). Enquanto o tipo `float` possui 32 bits de precisão, o `double` possui 64 bits.

Já vimos como declarar variáveis do tipo `double`, agora tentaremos fazer o mesmo com o tipo `float`.

```
float saldoConta = 1030.43; // não compila
System.out.println(saldoConta);
```

O código acima não compila porque todo literal decimal em Java é por padrão do tipo `double` (independente do valor). Por isso, precisamos mais uma vez tirar uma "carta na manga"!

Para fazer o código compilar e rodar como desejamos, precisamos apenas incluir a letra `F` após o número `1030.43`.

```
float saldoConta = 1030.43F; // compila!
System.out.println(saldoConta);
```

A letra `F` diz ao compilador que queremos que o número seja entendido como um valor do tipo `float`, e não `double`.

Os tipos `float` e `double` não devem ser usados para armazenar valores que devem ter muita precisão, como valores monetários. Para isso, é melhor usar um o tipo

`BigDecimal`, que não é um tipo primitivo, mas uma classe Java.

2.4. Classes wrapper de tipos primitivos

Muitas vezes, precisamos enxergar números, caracteres ou valores booleanos como objetos também, para trabalhar mais orientado a objetos. Por isso existem as classes wrapper, que "embrulham" tipos primitivos para que eles possam ser tratados como objetos e fornecem alguns métodos utilitários.

Cada tipo primitivo existente na linguagem Java tem sua classe wrapper correspondente. Os nomes das classes wrappers coincidem com o nome do tipo primitivo, começando com letra maiúscula, exceto para o tipo `char` e `int`, que recebem o nome completo (e não abreviado) para os seus tipos wrapper.

Vamos ver as classes wrappers existentes e suas correspondências para os tipos primitivos:

- `Boolean` → `boolean`
- `Character` → `char`
- `Byte` → `byte`
- `Short` → `short`
- `Integer` → `int`
- `Long` → `long`
- `Float` → `float`
- `Double` → `double`

É interessante usar classes wrappers quando queremos enxergar números, booleanos ou caracteres como objetos.

Os tipos primitivos não aceitam valores nulos e também não são suportados por algumas APIs, como a de Collections. Quando esse é o caso, usamos as classes wrapper.

2.5. Usando classes wrappers

Para instanciar um objeto de uma classe wrapper nós podemos usar o método `valueOf` da classe correspondente.

As classes wrappers implementam dois métodos `valueOf`: um que recebe o tipo primitivo como argumento e outro que recebe uma `String`, exceto a classe `Character`, que só recebe o tipo primitivo. Veja exemplos de uso:

```
// instanciando um wrapper que representa
// um número primitivo do tipo long
Long idadeEmMilisegundos = Long.valueOf(933120000000L);

// instanciando um wrapper que transforma
// a String "15" em tipo primitivo
// e embrulha em um objeto wrapper
Integer diasParaPagamento = Integer.valueOf("15");

// wrapper que representa um double
Double precoPassagem = Double.valueOf(200.10);

// wrapper que representa um float
Float distanciaPercorrida = Float.valueOf("100.6");

// representa o valor true
Boolean temPendencias = Boolean.valueOf(true);

// representa o valor false
// (qualquer texto diferente de true representa false)
Boolean atrasado = Boolean.valueOf("Yes");
```

Os métodos que instanciam objetos de classes wrappers de tipos numéricos e que recebem uma `String` como parâmetro, lançam exceções (erros) se a `String` fornecida não puder ser transformada em um tipo primitivo apropriado. Vamos ver alguns exemplos inválidos:

```
// não faz transformação de um número por extenso
Integer idade = Integer.valueOf("trinta e um");

// os decimais devem ser separados por ponto, e não vírgula
Double precoTotal = Double.valueOf("140,30");
```

Quando não for possível efetuar uma conversão de uma `String` para um tipo

específico (`Integer`, por exemplo), você receberá uma exceção, como pode ver abaixo:

```
Exception in thread "main" java.lang.NumberFormatException:
    For input string: "trinta e um"
    at java.base/java.lang.NumberFormatException.forInputString(
        NumberFormatException.java:65)
    at java.base/java.lang.Integer.parseInt(Integer.java:652)
    at java.base/java.lang.Integer.valueOf(Integer.java:983)
```

2.6. Autoboxing e autounboxing

Os recursos chamados de *autoboxing* e *autounboxing* permitem que tipos primitivos sejam usados como objetos de classes wrapper e vice-versa.

Esses recursos fazem o embrulho de tipos primitivos para objetos e o desembrulho de objetos para tipos primitivos, automaticamente, sempre que for necessário.

Isso quer dizer que o código abaixo é totalmente válido, mesmo misturando tipos primitivos com classes wrapper:

```
Integer quantidade = 10;
double valorUnitario = 45.35;

Double total = quantidade * valorUnitario;
System.out.println(total);
```

Capítulo 3

Strings

Para imprimir um texto na saída usando Java, tudo que precisamos fazer é colocá-lo entre aspas duplas dentro de um `System.out.println`:

```
System.out.println("Fala, mergulhadores!");
```

Se precisarmos juntar um texto com o valor de uma variável, podemos concatená-los. Na linguagem Java, usamos o símbolo + (mais) para fazer isso.

```
int x = 10;
int y = 5;
int soma = x + y; // adição
System.out.println("Resultado: " + soma); // concatenação
```

Quando usamos o + em textos, ele é entendido pelo compilador como concatenação, e não adição.

No exemplo acima, poderíamos eliminar a variável `soma` e efetuar o cálculo diretamente no `println`. Veja abaixo uma tentativa de fazer isso:

```
int x = 10;
int y = 5;
System.out.println("Resultado: " + x + y);
```

A execução do último exemplo exibirá na tela "Resultado: 105", pois o valor da variável `x` é concatenado com o valor da variável `y`. Isso acontece porque o compilador verifica que existe um texto sendo concatenado com a variável `x`, e faz o mesmo para a variável `y`.

Agora veja outro exemplo:

```
int x = 10;
int y = 5;
System.out.println(x + y + " foi o resultado");
```

Você apostaria que o código acima imprime na tela "15 foi o resultado"?

Apesar de parecer estranho, é isso que acontece. Neste caso, as variáveis `x` e `y` são somadas, e não concatenadas. Isso acontece porque o compilador só começa a concatenar a partir do momento que encontra um texto.

E se quisermos efetuar o cálculo de `x` com `y` depois que um texto foi digitado, como na primeira situação? Basta incluirmos a operação entre parênteses para dizer ao compilador que a operação tem precedência.

```
int x = 10;
int y = 5;
System.out.println("Resultado: " + (x + y));
```

Agora o resultado será "Resultado: 15".

Para declarar variáveis que contenham textos, usamos o tipo `String`.

```
String nome = "Maria";
```

Apesar do tipo `String` ser nativo da linguagem Java, ele não é um tipo primitivo, mas uma classe. Para entender a diferença você precisa aprender sobre orientação a objetos, que é um assunto muito importante, porém não é o objetivo desse e-book.

No exemplo abaixo, declaramos a variável `nome` do tipo `String` e `idade` do tipo `int`, e depois imprimimos na tela uma mensagem, concatenando o nome e a idade mais outros textos para formar uma frase completa.

```
String nome = "Maria";
int idade = 30;
System.out.println(nome + " tem " + idade + " anos");
```

O resultado na tela é "Maria tem 30 anos".

Capítulo 4

Operadores aritméticos

Existem 5 operadores aritméticos em Java que podemos usar para efetuar cálculos matemáticos. Uma operação pode ser de adição (+), subtração (-), multiplicação (*), divisão (/) ou módulo (%).

Só para lembrar, caso você tenha faltado das aulas de matemática, módulo é o resto da divisão entre dois números.

Outras operações, como exponenciação (potência), raiz quadrada e outras são fornecidas de maneiras diferentes (não vou falar sobre isso neste e-book).

Vejamos um exemplo simples usando as 5 operações aritméticas:

```
int soma = 2 + 10;
int subtracao = 6 - 10;
int multiplicacao = 8 * 3;
int divisao = 8 / 2;
int resto = 7 % 2;

System.out.println(soma);
System.out.println(subtracao);
System.out.println(multiplicacao);
System.out.println(divisao);
System.out.println(resto);
```

Os resultados das operações acima são: `12`, `-4`, `24`, `4` e `1`.

No último exemplo, calculamos valores literais (digitados "na mão"). Podemos ainda calcular valores de variáveis, tornando o programa muito mais dinâmico.

```
int notaAluno1 = 99;
int notaAluno2 = 80;
int notaAluno3 = 53;

int totalGeral = notaAluno1 + notaAluno2 + notaAluno3;
System.out.println(totalGeral);
```

Para praticar um pouco mais, queremos agora descobrir qual é a média de notas dos 3 alunos que temos. O que você acha que acontece se dividirmos por 3?

```
int totalGeral = notaAluno1 + notaAluno2 + notaAluno3 / 3;
```

A ideia é muito boa. Para descobrirmos a média de notas de 3 alunos, basta somarmos todas as notas e dividir por 3, mas como fizemos no exemplo acima, estamos dividindo a última nota por 3, e não o resultado da somatória de todas as notas. Por isso, o resultado dessa operação é 196.

Para ficar correto, temos que agrupar a somatória usando parênteses, assim dizemos ao compilador que queremos realizar a operação de soma antes da divisão.

```
int totalGeral = (notaAluno1 + notaAluno2 + notaAluno3) / 3;
```

Agora sim, o resultado da operação ficará correto, resultando na média de 77 pontos por aluno.

Capítulo 5

Operadores de comparação e igualdade

Operadores de comparação e de igualdade são usados para realizar comparações de conteúdos de variáveis ou valores literais. Os resultados das comparações sempre resultam em valores booleanos.

Os operadores de comparação disponíveis são `>` (maior), `>=` (maior ou igual), `<` (menor) e `<=` (menor ou igual), e os operadores de igualdade são `==` (igual) e `!=` (diferente). O exemplo abaixo usa todos estes operadores.

```
boolean maior = b > a; // 'b' é maior que 'a'?
boolean maiorOuIgual = b >= a; // 'b' é maior ou igual a 'a'?
boolean menor = a < b; // 'a' é menor que 'b'?
boolean menorOuIgual = a <= 10; // 'a' é menor ou igual a '10'?
boolean igual = a == b - c; // 'a' é igual a 'b' menos 'c'?
boolean diferente = a != c; // 'a' é diferente de 'c'?

System.out.println(maior);
System.out.println(maiorOuIgual);
System.out.println(menor);
System.out.println(menorOuIgual);
System.out.println(igual);
System.out.println(diferente);
```

Coincidentemente, todas as comparações acima resultam no valor booleano `true` (verdadeiro). Perceba que na quarta linha incluímos um valor literal para fazer a comparação, e na quinta linha fizemos uma operação aritmética subtraindo duas variáveis antes de realizar a comparação. Fizemos isso só para exercitarmos um

pouco mais a linguagem Java.

Já que falamos de operadores de igualdade, vale a pena falarmos do operador unário `!` (ponto de exclamação). Este operador serve para negar um valor booleano ou uma expressão booleana.

Se tivermos uma variável booleana com `true` atribuído a ela, podemos negar a própria variável para o valor passar a valer `false`.

```
boolean bloqueado = true;
bloqueado = !bloqueado; // passa a valer false
```

Podemos também negar uma expressão que possui um operador de comparação. Por exemplo:

```
boolean resultado = !(b > a); // ruim, mas válido
```

O código acima compara se `b` é maior que `a` e inverte o resultado. É o mesmo que fazer `b <= a`, porém com uma legibilidade um pouco pior (pois exige mais raciocínio para entender o código).

Capítulo 6

Estruturas de decisão

As estruturas de decisão nos permitem ter fluxos (ou caminhos) alternativos dentro do nosso programa.

Isso é fundamental para qualquer bom programa, que precisa tomar decisão durante a sua execução, seguindo caminhos diferentes de acordo a lógica do sistema.

Neste e-book nós vamos estudar a estrutura com a instrução `if` e suas variações.

6.1. Instrução `if`

Como em muitas linguagens de programação, em Java usamos a instrução `if` (se) para controlar um fluxo básico.

No exemplo abaixo, escrevemos um código-fonte que calcula o IMC de uma pessoa para exibir uma mensagem caso seu peso esteja abaixo do ideal. Para saber se o peso está abaixo do ideal, precisamos verificar se o resultado do cálculo do IMC é menor que 18.5.

```
int peso = 76;
double altura = 1.82;

double imc = peso / (altura * altura);

if (imc < 18.5) {
    System.out.println("Abaixo do peso ideal.");
```

```
}
```

Veja que usamos uma instrução `if` para comparar se a variável `imc` é menor que `18.5`, através da expressão `imc < 18.5`.

```
if (imc < 18.5) {
    System.out.println("Abaixo do peso ideal.");
}
```

A instrução `if` deve receber uma expressão booleana agrupada por parênteses.

No exemplo acima, a chamada de `System.out.println` só será executada caso o valor da variável `imc` seja menor que `18.5`.

6.2. Instrução `else if`

Se o cálculo do IMC não for menor que `18.5`, queremos que uma nova condição seja verificada para identificar se o peso do usuário é o ideal. Para isso, usamos a instrução `else if` (senão se), que recebe uma expressão booleana e é analisada apenas se a primeira instrução `if` for falsa.

```
if (imc < 18.5) {
    System.out.println("Abaixo do peso ideal.");
} else if (imc < 25) {
    System.out.println("Peso ideal.");
}
```

Agora, caso o IMC do usuário não seja menor que `18.5`, verificamos se é menor que `25` (ou seja, maior que `18.5` e menor que `25`) e imprimimos outra mensagem dizendo que o peso é ideal.

6.3. Instrução `else`

Finalizamos nosso programa incluindo outras condições que indicam o grau de obesidade de uma pessoa. Veja no último caso que usamos a instrução `else` (caso contrário ou senão).

Caso nenhuma das condições expressadas no `if` e nos `else if`s forem

verdadeiras, o bloco de código de `else` será executado.

```
if (imc < 18.5) {
    System.out.println("Abaixo do peso ideal.");
} else if (imc < 25) {
    System.out.println("Peso ideal.");
} else if (imc < 30) {
    System.out.println("Acima do peso.");
} else if (imc < 35) {
    System.out.println("Obesidade grau I.");
} else if (imc < 40) {
    System.out.println("Obesidade grau II.");
} else {
    System.out.println("Obesidade grau III.");
}
```

A instrução `else` não recebe nenhuma expressão booleana, pois ela será executada apenas se todas as instruções anteriores forem falsas.

6.4. Programar ifs sem abrir e fechar blocos é legal?

Existe a possibilidade modificar o código-fonte do último exemplo sem mudar o comportamento do programa, eliminando as aberturas e fechamentos de blocos (chaves).

```
if (imc < 18.5)
    System.out.println("Abaixo do peso ideal.");
else if (imc < 25)
    System.out.println("Peso ideal.");
else if (imc < 30)
    System.out.println("Acima do peso.");
else if (imc < 35)
    System.out.println("Obesidade grau I.");
else if (imc < 40)
    System.out.println("Obesidade grau II.");
else
    System.out.println("Obesidade grau III.");
```

O resultado é o mesmo e, apesar de parecer que o código está mais limpo e bonito, existe um risco muito grande quando programamos assim.

Imagine se precisássemos adicionar uma nova mensagem a ser exibida para o

usuário, quando o grau de obesidade for III. Poderíamos tentar incluir uma linha que deve ser executada na instrução `else`.

```
if (imc < 18.5)
    System.out.println("Abaixo do peso ideal.");
else if (imc < 25)
    System.out.println("Peso ideal.");
else if (imc < 30)
    System.out.println("Acima do peso.");
else if (imc < 35)
    System.out.println("Obesidade grau I.");
else if (imc < 40)
    System.out.println("Obesidade grau II.");
else
    System.out.println("Obesidade grau III.");
    System.out.println("Muito cuidado com seu peso.");
```

Ao executar esse programa, se o IMC for acima de 40 (nível máximo de obesidade), as mensagens aparecerão como esperado, como se o nosso sistema estivesse funcionando corretamente.

```
Obesidade grau III.
Muito cuidado com seu peso.
```

Agora, se executamos o programa novamente (sem fazer qualquer alteração) e informarmos um peso ideal para uma determinada altura, veja que algo de errado acontece.

```
Peso ideal.
Muito cuidado com seu peso.
```

A mensagem "Muito cuidado com seu peso" apareceu para alguém que tem peso ideal, mas essa não era nossa intenção.

O problema é que, quando não usamos as chaves para abrir e fechar blocos, nos enganamos facilmente. Sem as chaves, cada instrução `if`, `else if` ou `else` faz referência apenas para a primeira instrução logo em seguida.

O último exemplo é como se tivéssemos feito:

```
...
else
    System.out.println("Obesidade grau III.");
```

```
System.out.println("Muito cuidado com seu peso.");
```

Ou ainda:

```
...
else {
    System.out.println("Obesidade grau III.");
}

System.out.println("Muito cuidado com seu peso.");
```

Por existir essa "facilidade" de errar, eu recomendo que você evite programar `if/else/else if` sem usar blocos.

6.5. Escopo de variáveis

O escopo de variáveis define em qual parte do programa a variável pode ser referenciada através de seu nome.

Para usarmos uma variável, ela deve ser declarada antes, mas só isso não é suficiente. Deve-se observar se a variável não foi declarada em um bloco mais interno que o código que está tentando usá-la. O código-fonte abaixo é um exemplo que nem mesmo compila:

```
int idade = 39;
boolean podeDirigir = idade >= 18;

if (!podeDirigir) {
    // variável nomeResponsavel está sendo declarada apenas para o
    // o bloco deste if
    String nomeResponsavel = "João";
}

if (podeDirigir) {
    System.out.println("Você pode dirigir.");
} else {
    // não compila! variável nomeResponsavel não está acessível
    System.out.println("Você não pode dirigir. Fale com "
        + nomeResponsavel + ".");
}
```

O código acima não compila porque a variável `nomeResponsavel` ficaria visível apenas para o bloco do `if (!podeDirigir) { ... }`.

Quando criamos variáveis em Java, elas ficam acessíveis apenas dentro do bloco o qual ela foi declarada e seus sub-blocos.

Para o exemplo anterior compilar, precisaríamos declarar a variável `nomeResponsavel` fora do bloco do `if`, deixando-a visível para todo o método `main` do programa.

```
String nomeResponsavel = "";
int idade = 39;
boolean podeDirigir = idade >= 18;

if (!podeDirigir) {
    nomeResponsavel = "João";
}

if (podeDirigir) {
    System.out.println("Você pode dirigir.");
} else {
    // não compila! variável nomeResponsavel não está acessível
    System.out.println("Você não pode dirigir. Fale com "
        + nomeResponsavel + ".");
}
```

Agora o programa pode ser compilado e executado normalmente.

Obviamente, este código é só um exemplo didático. Não teria sentido adicionar o bloco `if (!podeDirigir) {` apenas para atribuir o nome do responsável. Isso poderia ser feito no `else`.

6.6. Operadores lógicos

Nos últimos exemplos, usamos as regras da Organização Mundial de Saúde para calcular o IMC. Existem outras regras mais detalhadas, como as da NHANES II survey (USA 1976-1980), que indicam os seguintes critérios para adultos:

Condição	IMC em mulheres	IMC em homens

Abaixo do peso	Menor que 19.1	Menor que 20.7
No peso ideal	Entre 19.1 e 25.8	Entre 20.8 e 26.4
Um pouco acima do peso	Entre 25.9 e 27.3	Entre 26.5 e 27.8
Acima do peso ideal	Entre 27.4 e 32.3	Entre 27.9 e 31.1
Obeso	Maior que 32.3	Maior que 31.1

Como você pode ver, essas regras são um pouco mais complicadas, pois usam critérios diferentes entre homens e mulheres. Apesar disso, não é tão difícil programá-las.

Existem diversas formas de codificar um programa com esses critérios. Uma forma seria aninhar (agrupar) `ifs`, colocando um bloco dentro de outro.

```
String nome = "Maria";
int peso = 49;
double altura = 1.58;
char sexo = 'F';

double imc = peso / (altura * altura);

if (sexo == 'F') {
    if (imc < 19.1) {
        System.out.println("Abaixo do peso.");
    } else if (imc <= 25.8) {
        System.out.println("Peso ideal.");
    }

    // e continua...
} else {
    if (imc < 20.7) {
        System.out.println("Abaixo do peso.");
    } else if (imc <= 26.4) {
        System.out.println("Peso ideal.");
    }

    // e continua...
}
```

No exemplo acima, definimos algumas variáveis no início, como peso, altura e

sexo (F ou M).

Quando colocamos `ifs` dentro de `ifs`, estamos dizendo que o `if` mais interno só será avaliado caso o primeiro seja verdadeiro.

```
if (sexo == 'F') {

    // só será avaliado se sexo for Feminino
    if (imc < 19.1) {
        System.out.println("Abaixo do peso.");
    } else if (imc <= 25.8) {
        System.out.println("Peso ideal.");
    }

    // e continua...
}
```

Ao invés de incluirmos `ifs` dentro de `ifs`, podemos usar o **operador lógico** "E", que é representado por &&.

```
if (sexo == 'F' && imc < 19.1) {
    System.out.println("Abaixo do peso.");
} else if (sexo == 'F' && imc <= 25.8) {
    System.out.println("Peso ideal.");
} else if (sexo == 'F' && imc <= 27.3) {
    System.out.println("Um pouco acima do peso.");
} else if (sexo == 'F' && imc <= 32.3) {
    System.out.println("Acima do peso ideal.");
} else if (sexo == 'F') {
    System.out.println("Obeso.");
} else if (sexo == 'M' && imc < 20.7) {
    System.out.println("Abaixo do peso.");
} else if (sexo == 'M' && imc <= 26.4) {
    System.out.println("Peso ideal.");
} else if (sexo == 'M' && imc <= 27.8) {
    System.out.println("Um pouco acima do peso.");
} else if (sexo == 'M' && imc <= 31.1) {
    System.out.println("Acima do peso ideal.");
} else if (sexo == 'M') {
    System.out.println("Obeso.");
}
```

O operador lógico && avalia as expressões do lado esquerdo e direito e retorna apenas um resultado booleano.

Para a expressão completa ser verdadeira, tanto o lado direito como o lado esquerdo devem ser verdadeiras, mas para que a expressão completa seja falsa, pelo menos um lado deve ser falso.

Ainda existe o operador lógico "OU", representado por `||`. Podemos ainda mudar o código-fonte de nosso exemplo para usá-lo.

```
if ((sexo == 'F' && imc < 19.1) || (sexo == 'M' && imc < 20.7)) {
    System.out.println("Abaixo do peso.");
} else if ((sexo == 'F' && imc <= 25.8)
        || (sexo == 'M' && imc <= 26.4)) {
    System.out.println("Peso ideal.");
} else if ((sexo == 'F' && imc <= 27.3)
        || (sexo == 'M' && imc <= 27.8)) {
    System.out.println("Um pouco acima do peso.");
} else if ((sexo == 'F' && imc <= 32.3)
        || (sexo == 'M' && imc <= 31.1)) {
    System.out.println("Acima do peso ideal.");
} else {
    System.out.println("Obeso.");
}
```

Veja que usando o operador `||`, conseguimos diminuir bastante a quantidade de linhas de programação, porém deixamos as expressões booleanas dos `ifs` mais complexas, pois agrupamos duas expressões que usam o operador `&&` dentro de outra que usa o operador `||`.

O operador lógico `||` avalia as expressões dos dois lados e retorna um único resultado booleano. Para que a expressão completa seja verdadeira, pelo menos um lado deve ser verdadeiro.

Como boa prática, evite ter blocos dentro de blocos, mas também sempre avalie a simplicidade de leitura do código. Sem dúvidas, com o uso de orientação a objetos seria possível deixar esse código muito mais elegante e legível.

Capítulo 7

Estruturas de repetição

Estruturas de repetição são usadas para executar blocos de código diversas vezes, em um *loop*, enquanto uma condição for verdadeira.

Existem várias formas de fazer isso com Java. Neste e-book você vai aprender as mais básicas, com `while` e `for`.

7.1. Estrutura `while`

O `while` é uma das formas de fazer loops (laços) em Java.

O exemplo abaixo imprime todos os números em um intervalo de 10 a 20.

```
int numeroInicial = 10;
int numeroFinal = 20;

int numeroAtual = numeroInicial;

while (numeroAtual <= numeroFinal) {
    System.out.println(numeroAtual);
    numeroAtual = numeroAtual + 1;
}
```

O bloco de código dentro do `while` será executado enquanto a condição `numeroAtual <= numeroFinal` for verdadeira, ou seja, enquanto o valor da variável `numeroAtual` for menor ou igual ao valor da variável `numeroFinal`.

Apenas uma curiosidade: podemos incrementar o valor da variável `numeroAtual`

usando o operador de incremento ++. Por exemplo:

```
// numeroAtual = numeroAtual + 1;
numeroAtual++;
```

7.2. Estrutura `for`

Uma estruturas de repetição muito usada é a `for`, pois ela fornece uma forma fácil de iterar (percorrer) em um intervalo de valores.

A forma básica da instrução `for` é a seguinte:

```
for (iniciação; condição; incremento) {
    // bloco de código do loop
}
```

A expressão de iniciação é executada uma única vez assim que o loop é iniciado. Normalmente é declarada uma variável de controle com um valor inicial.

A expressão de condição é semelhante a outras estruturas de repetição. Você deve incluir uma expressão booleana que definirá a permanência ou término do laço. Essa expressão é analisada a cada iteração do loop.

A última expressão, de incremento, é executada após cada iteração do laço e normalmente é usada para modificar o valor da variável de controle, seja incrementando ou decrementando.

Vamos a um exemplo:

```
int numeroFinal = 9;

for (int i = 1; i <= numeroFinal; i++) {
    System.out.println(i);
}
```

O código acima executa o laço `for` e imprime números de `1` até `9`. Note que uma variável de controle `i` foi declarada na seção de iniciação, avaliada na condição e incrementada na seção de incremento.

Capítulo 8

Conclusão

Parabéns, mergulhador!

Espero que este e-book tenha reforçado seu conhecimento sobre os fundamentos e sintaxe da linguagem Java, para que você possa participar do **MSR** mais confiante.

Não esqueça que o primeiro módulo do **MSR** será publicado no dia **16 de maio** e o curso ficará disponível só por uma semana. Anote na sua agenda!

Um abraço!