

Lógica de Programação com



Java™

Nível Básico

Sumário

1 Introdução

1.1	O que é um algoritmo?	5
1.2	Linguagens de programação	6
1.3	Instalando o Java	7
1.4	O famoso “Olá mundo!”	10
1.5	Executando o primeiro algoritmo	14
1.6	Destrinchando o primeiro programa	15
1.7	Fazendo comentários em um programa Java	17

2 Variáveis e constantes

2.1	Tipos de dados existentes	22
2.2	Escolhendo bons nomes para variáveis e constantes	24

3 Entrada de dados

3.1	Lendo informações com o Scanner	27
3.2	Lendo diferentes tipos de dados	30

4 Operadores

4.1	Operadores aritméticos	32
4.2	Operadores de atribuição	33
4.3	Operadores relacionais	35
4.4	Operadores lógicos	37

5 Estruturas de decisão

5.1	Estrutura “if”	41
5.2	Estrutura “switch”	45

6 Vetores

6.1	Vetores de uma dimensão	48
-----	-------------------------------	----

7 Iteração

7.1	Iterando com o laço “while”	51
7.2	Iterando com o laço “for”	54
7.3	Quando utilizar o laço “for” ou o laço “while”	56

Capítulo 1

Introdução

Uma das tarefas mais difíceis para quem inicia os estudos sobre programação é entender conceitos que tratam de coisas muito abstratas. Para conseguir esse entendimento, o legal mesmo de você fazer é colocá-lo em prática.

Mas aí vem outro problema porque, para colocar em prática, é preciso criar um ambiente em seu computador com uma confiança mínima de que ele está correto, pois senão você terá duas grandes barreiras. A primeira é de não saber programar, a segunda é de nem mesmo conseguir executar o seus primeiros testes por não saber criar um ambiente para isso.

Criar um ambiente, basicamente, seria montar a estrutura necessária para poder programar em seu próprio computador. Toda linguagem de programação precisa disso e, por mais que não seja tão complexo assim, muitos podem desistir sem nem sequer ter feito um simples teste prático.

Por isso esse material foi criado, para ajudar você com essas duas coisas, ou seja, para ensinar a você **lógica de programação** juntamente com uma excelente de linguagem de programação como é o caso do **Java**.

Aqui, vamos começar conversando sobre o que é um algoritmo - conceito super importante em lógica de programação - e depois já começaremos com a parte prática. Faremos a configuração do ambiente Java e continuaremos falando sobre o que é necessário para a criação de algoritmos, como variáveis, estruturas de decisão, laços e outras coisas mais.

1.1. O que é um algoritmo?

Quem nunca viu uma definição sobre o que é um algoritmo pode se assustar um pouco somente porque isso está relacionado a programas de computadores, mas algoritmos não se restringem ao mundo do desenvolvimento de software.

Um exemplo disso são os passos que você segue para executar uma receita de bolo ou qualquer outra coisa. Sim, podemos considerar uma receita de bolo como um algoritmo.

Podemos chamar de algoritmo tudo o que exige uma sequência de passos organizados e que possui um fim. Mais especificamente, podemos ter a seguinte definição:

Algoritmo é uma sequência de passos ordenados e finitos.

Para te dar um exemplo mais parecido com o que encontramos em programas de computadores, vamos imaginar que estamos desenvolvendo um software onde uma pequena parte dele seria encontrar a velocidade média de um automóvel. Essa pequena parte seria o nosso algoritmo e ele teria os seguintes passos:

1. Obter a distância - em quilômetros - que foi percorrida
2. Obter o tempo gasto - em horas - para percorrer essa distância
3. Dividir a distância pelo tempo
4. Exibir o resultado da divisão

Como você pode reparar eu utilizei uma linguagem bem humana para criar o meu algoritmo. Agora, vou utilizar uma pseudo-linguagem de programação para representar esse mesmo algoritmo.

1. `distanciaPercorrida = obterInformacaoDoUsuario();`
2. `tempoGasto = obterInformacaoDoUsuario();`
3. `velocidadeMedia = distanciaPercorrida / tempoGasto;`
4. `exibirResultadoParaUsuario(velocidadeMedia);`

A versão do algoritmo que está acima já é bem mais próxima de uma linguagem de programação, mas ainda não podemos executá-la em um computador. Como mencionei, utilizei uma pseudo-linguagem que, inclusive, inventei agora

somente para que você se familiarize melhor com a linguagem Java quando ela for apresentada.

1.2. Linguagens de programação

São inúmeras as linguagens de programação com as quais você consegue implementar um algoritmo. Como exemplos, posso citar:

- Java
- C++
- Ruby
- Python
- JavaScript
- C#
- PHP

Todas as linguagens da lista acima são muito boas, mas, como você já sabe, utilizaremos a **linguagem Java**, e temos bons motivos para essa escolha:

- Recursos técnicos disponíveis
- Comunidade
- Mercado

Dentro do Java encontramos uma **ampla gama de recursos técnicos disponíveis**. Temos tudo que é preciso para criação de programas de computador de alta qualidade, sejam eles de pequeno, médio ou grande porte.

Um outro bom motivo é que o Java tem uma **comunidade gigante de programadores** tanto no Brasil quanto no resto do mundo.

A comunidade facilita o compartilhamento de conhecimento tanto para quem deseja aprender algo novo dentro da linguagem - como é o caso desse material aqui que você tem em mãos - quanto também para os momentos em que precisamos de ajuda para a correção de erros.

Por último e não menos importante, o **Java é a linguagem mais requerida pelo mercado de trabalho**. Isso significa que um bom programador Java tem

mais chances de se colocar no mercado do que um outro bom programador em qualquer uma das outras linguagens.

1.3. Instalando o Java

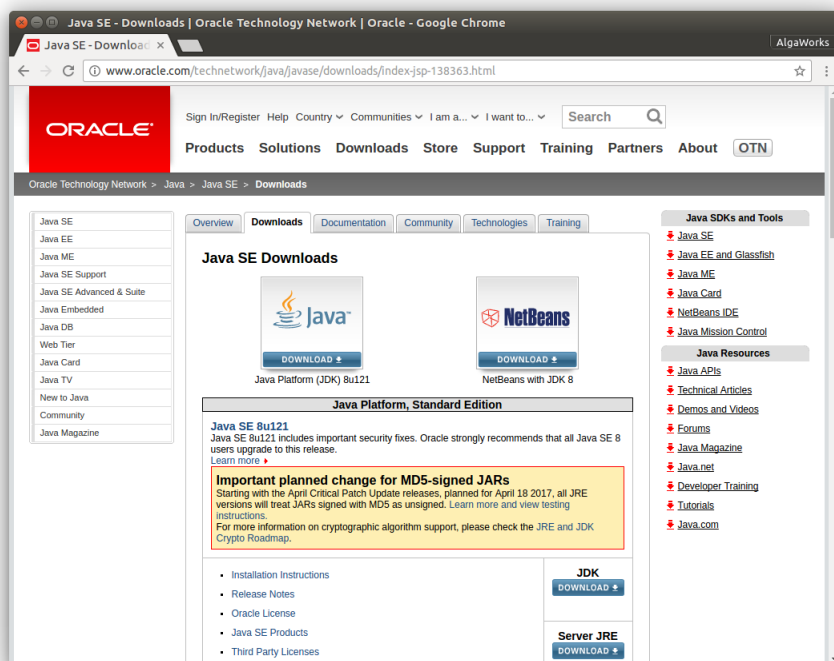
Agora iremos configurar o ambiente que vai tornar possível a execução de um algoritmo feito na linguagem de programação Java. Para começar, nós precisamos fazer a instalação do *Java Development Kit* - ou, em português, Kit de Desenvolvimento Java - muito conhecido pela sigla JDK.

O **JDK** nada mais é que um software, ou seja, dentro dele tem várias ferramentas que nos ajudam a criar nossos programas na linguagem Java. Apesar disso, dificilmente fazemos referência ao JDK como um software. Programas como ele são mais conhecidos como **compiladores** (na verdade, é muito mais do que isso, mas vamos com calma para você não se perder).

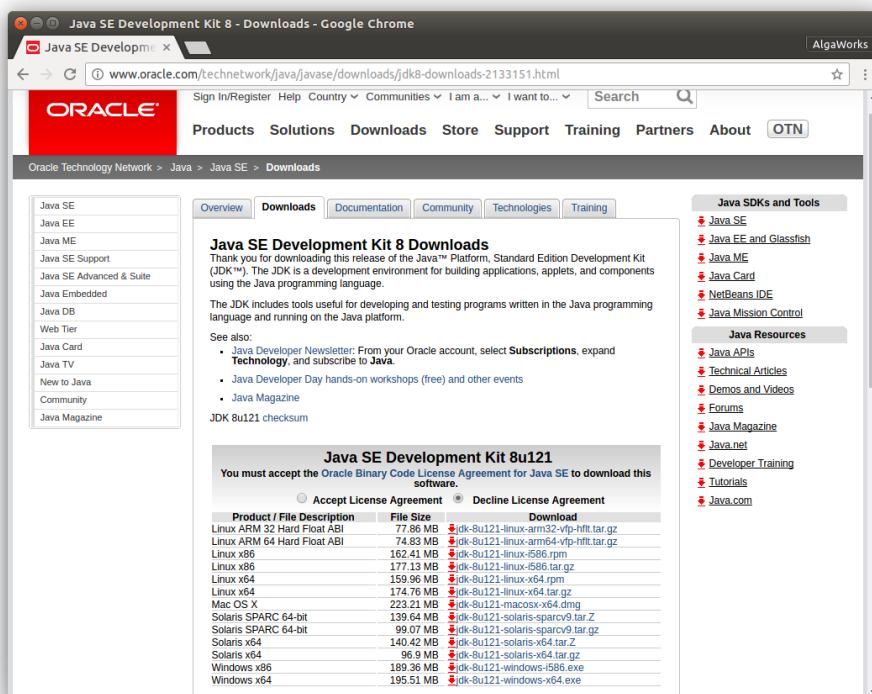
Ele se faz necessário porque, para criar um programa de computador feito com Java, precisamos que um outro programa - o JDK, no caso - execute os algoritmos.

Para a instalação do JDK você vai precisar visitar o site da Oracle - empresa que mantém o Java - e fazer o download:

<http://www.oracle.com/technetwork/java/javase/downloads/>



Feito isso, clique no retângulo com o ícone da xícara de café - onde está escrito “Java”. Esse clique vai te levar para a seguinte página:



Agora escolha a versão de download do JDK de acordo com o seu sistema operacional (Linux, Mac, Windows, etc). Depois do download, basta executar o arquivo baixado. A instalação em si é bem simples.

Para aquelas pessoas que usam alguma distribuição Linux, como o caso do Ubuntu, podem procurar o JDK em seu gerenciador de pacotes. No Ubuntu, por exemplo, é possível utilizar os comandos:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

Com a instalação concluída - seja por download do arquivo através do site ou pelo gerenciador de pacotes - abra seu terminal (ou prompt de comando) e digite:

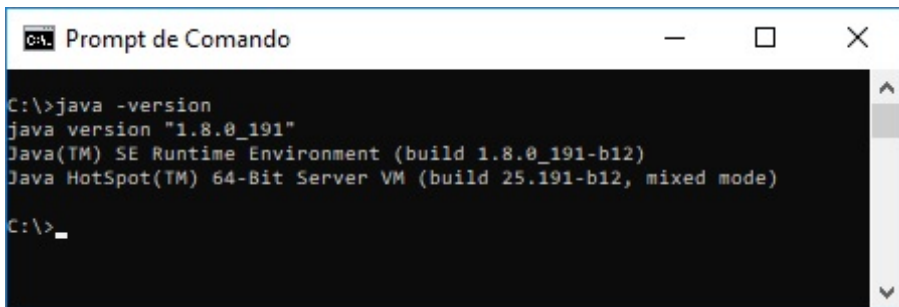
```
$ java -version
```

A resposta deve ser parecida com o conteúdo abaixo:

```
java version "1.8.0_121"  
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)  
Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)
```

Se tiver aparecido algo como acima é porque o Java está instalado e funcionando em sua máquina.

Para quem não se lembra, o terminal (ou console ou prompt de comando - **CMD**) é aquela janelinha, muitas vezes da cor preta, que utilizamos para executar comandos que não precisam de uma interface gráfica. Exemplo:



```
C:\>java -version  
java version "1.8.0_191"  
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)  
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)  
C:\>_
```

1.4. O famoso “Olá mundo!”

Com o Java instalado e funcionando, podemos criar o famoso “Olá Mundo!”. Famoso porque no aprendizado de todas as linguagens é um costume criar um pequeno programa com a finalidade de imprimir (exibir) na tela justamente o texto “Olá Mundo!”.

Para começar a organizar as coisas, crie uma pasta em qualquer lugar do seu computador, onde você irá deixar os arquivos que vamos precisar criar durante nosso estudo, começando pelo arquivo *OlaMundo.java*.

Esse arquivo é um arquivo de texto puro normal, porém como vamos programar em Java, nós temos que usar a extensão “.java”.

Muito cuidado ao nomear o arquivo, pois iremos usar esse nome dentro do nosso programa. Note que as letras “O” e “M” são em maiúsculo. Agora, veja o conteúdo inicial do arquivo *OlaMundo.java*:

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

Repare que na linha 1 da estrutura do nosso programa foi necessário utilizar o termo “OlaMundo”, que deve ser exatamente igual ao nome do arquivo, inclusive, as partes maiúsculas e minúsculas.

O `public static void main`, na linha 3, é o ponto de entrada em nosso programa, ou seja, tudo que estiver dentro do bloco delimitado pelas chaves será executado ao rodar o nosso pequeno programa. Não mude nada nessa linha, ou pode dar tudo errado. Como você está só começando agora, não faria sentido explicar o motivo de cada palavra da linha 3.

Feito isso, vamos incluir agora a linha responsável pela impressão do texto “Olá Mundo!”. Impressão essa que será feita na tela (terminal ou prompt de comando). Veja como vai ficar:

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

A única instrução que temos em nosso programa “Olá Mundo” é a impressão do próprio texto “Olá Mundo!”. Impressão essa que é feita com o auxílio do comando `System.out.println`. Repare que, ao final da instrução, foi colocado um sinal “;” (ponto e vírgula), pois, toda instrução Java precisa encerrar com esse sinal.

Agora, **precisamos compilar o arquivo**. Essa compilação vai gerar um outro, de mesmo nome e com a extensão “.class”, com um conteúdo que chamamos de

bytecode. Isso significa que iremos preparar o arquivo para ser executado pela máquina. Essa compilação é feita com um comando que executamos através do terminal ou prompt de comando. Então, abra o seu terminal e execute:

```
$ javac OlaMundo.java
```

Lembrando que para executar o comando acima é necessário estar dentro da pasta onde você salvou o arquivo.

Para entrar na pasta pelo terminal, você pode utilizar o comando *cd* (*change directory* - ou, em português, alterar diretório). Supondo que sua pasta com os arquivos se chame “LogicaProgramacao”, então o comando seria algo como:

```
$ cd /caminho/da/pasta/com/o/arquivo/LogicaProgramacao
```

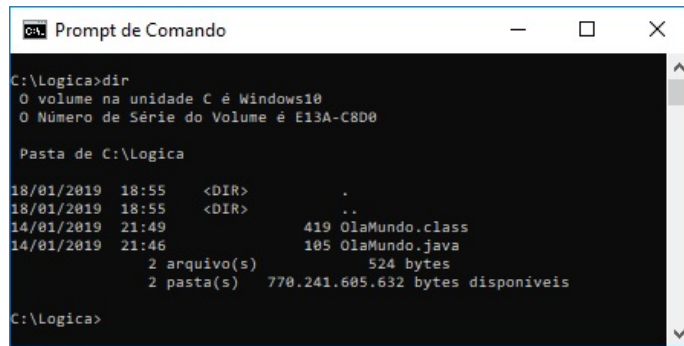
No Windows ficaria algo do tipo:

```
C:\> cd C:\caminho\da\pasta\com\o\arquivo\LogicaProgramacao
```

Agora você pode executar o comando *javac* apresentado anteriormente. E muito cuidado para não executar o comando errado como, por exemplo, o comando *java*, *javap*, etc.. Agora é o momento do comando *javac*.

Outra coisa importante é que, sempre que você alterar o arquivo com a extensão “.java”, será necessário compilar o arquivo novamente antes da nova execução.

Logo depois que o *javac* for executado, será gerado um novo arquivo no mesmo lugar em que está o arquivo *OlaMundo.java*. O arquivo gerado terá o mesmo nome, mas com uma extensão diferente. Você pode conferir isso utilizando o comando *ls* - ou *dir* para quem usa Windows - que serve para listar os arquivos do diretório. Veja o meu caso:



```
C:\Logica>dir
O volume na unidade C é Windows10
O Número de Série do Volume é E13A-C8D0

Pasta de C:\Logica

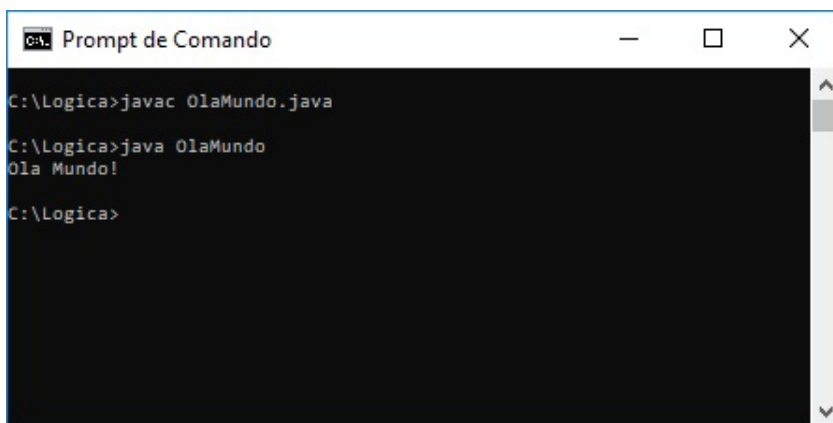
18/01/2019  18:55    <DIR>          .
18/01/2019  18:55    <DIR>          ..
14/01/2019  21:49                419 OlaMundo.class
14/01/2019  21:46                105 OlaMundo.java
                2 arquivo(s)                524 bytes
                2 pasta(s)       770.241.605.632 bytes disponiveis

C:\Logica>
```

Chegou o momento de, realmente, executar esse nosso pequeno programa. Vamos fazer isso com o comando *java*:

```
$ java OlaMundo
```

Repare que agora o comando é o *java*, e não o *javac*, e também que você não deve adicionar a extensão do arquivo, pois o comando *java* acessa diretamente o arquivo com a extensão “.class” - referente a versão compilada do nosso arquivo *OlaMundo.java*. Olhe o resultado:



```
C:\Logica>javac OlaMundo.java

C:\Logica>java OlaMundo
Ola Mundo!

C:\Logica>
```

Se no final for impresso o texto “Olá Mundo!”, quer dizer que você conseguiu executar nosso pequeno programa chamado *OlaMundo.java*.

Parabéns! Esse pode ser considerado seu primeiro marco nos estudos sobre lógica e também de programação no geral. O primeiro passo foi dado.

1.5. Executando o primeiro algoritmo

Lembra do primeiro algoritmo que criamos no início do livro? Pois então, como primeiro programa formal, iremos implementar ele na linguagem Java.

Para isso precisaremos utilizar alguns recursos da lógica de programação e da linguagem Java que ainda não estudamos, mas não precisa se preocupar, porque veremos mais detalhes sobre esses recursos no capítulos seguintes. Os recursos utilizados agora, e que iremos ver melhor depois, são:

- Variáveis com seus respectivos tipos
- Operadores aritméticos
- Escrita de informações no terminal

Por ora, gostaria que você executasse o programa Java que vou te passar para que entenda como funciona um programa de computador um pouco mais complexo que um simples “Olá Mundo!”.

Vamos ao programa então. Crie, no diretório que separou para os exemplos do nosso livro, um arquivo chamado *CalcularVelocidadeMedia.java*. Esse arquivo terá o seguinte conteúdo:

```
public class CalcularVelocidadeMedia {  
  
    public static void main(String[] args){  
        System.out.println("=== CÁLCULO DA VELOCIDADE MÉDIA ===");  
  
        double distanciaPercorrida = 150;  
        System.out.println("A distância percorrida foi de "  
            + distanciaPercorrida + "Km.");  
  
        double tempoGasto = 1.5;  
        System.out.println("O tempo gasto foi de "  
            + tempoGasto + " horas.");  
  
        double velocidadeMedia = distanciaPercorrida / tempoGasto;  
        System.out.println("Para encontrar a velocidade média "  
            + "dividimos a distância (" + distanciaPercorrida  
            + ") pelo tempo gasto no "  
            + "percurso (" + tempoGasto + ").");  
    }  
}
```

```

        System.out.println("A velocidade média é de "
            + velocidadeMedia + "Km/h");
    }
}

```

A partir daqui o processo será o mesmo que foi para o arquivo *OlaMundo.java*, ou seja, será feita uma compilação utilizando o comando *javac* e, depois, a execução de fato com o comando *java*. Veja os comandos:

```

$ javac CalcularVelocidadeMedia.java
$ java CalcularVelocidadeMedia

```

Todo o processo, você pode conferir na imagem abaixo:

```

C:\Logica>javac CalcularVelocidadeMedia.java

C:\Logica>java CalcularVelocidadeMedia
=== CALCULO DA VELOCIDADE MEDIA =====
A distancia percorrida foi de 150.0Km.
O tempo gasto foi de 1.5 horas.
Para encontrar a velocidade média dividimos a distância (150.0) pelo
tempo gasto no percurso (1.5).
A velocidade media e de 100.0Km/h

C:\Logica>_

```

No exemplo acima, o valor escolhido para distância percorrida foi 150Km, o tempo gasto foi de 1,5 horas e o resultado foi 100Km/h. Repare que o tempo não é dado em horas e minutos e sim em fração de horas. No caso, o valor 1,5 é, na verdade, 1 hora e 30 minutos.

1.6. Destrinchando o primeiro programa

Vamos falar um pouco agora sobre cada linha do nosso primeiro programa. Aquele criado no tópico anterior.

A primeira coisa que temos é a impressão de uma mensagem que funciona como uma espécie de título do nosso pequeno programa:

```
System.out.println("=== CÁLCULO DA VELOCIDADE MÉDIA =====");
```

Essa impressão é feita da mesma forma que fizemos com o texto “Olá Mundo!” que foi referente a primeira execução apresentada aqui pelo livro. Sempre que se deparar com o comando `System.out.println` entenda que algo será impresso no console.

Como próximo passo, é feita a declaração de uma variável que guardará o valor referente a distância percorrida:

```
double distanciaPercorrida = 150.0;
```

Estudaremos variáveis mais para frente. Por ora, entenda que uma variável é como se fosse uma caixinha da nossa memória onde nós podemos guardar algum dado. No caso acima, foi a distância percorrida.

Na próxima linha, temos uma outra impressão levemente diferente da anterior:

```
System.out.println("A distância percorrida foi de "  
    + distanciaPercorrida + "Km.");
```

A diferença da impressão acima para anterior é que fizemos o uso do sinal de “+” que, quando utilizado com textos, serve para juntar o lado esquerdo com o lado direito - um processo que chamamos de concatenação. Dentro da programação, o sinal de “+” funciona como um operador de adição, mas quando ele tem um texto de um dos lados, vai acontecer que o outro lado será concatenado, formando uma só sentença.

Essa técnica de concatenação é muito útil onde for necessário incluir um valor dinâmico juntamente com um texto - no caso acima, o valor dinâmico é aquele para qual o termo `distanciaPercorrida` está fazendo referência.

As próximas duas linhas fazem a mesma coisa que as duas anteriores, só que agora para o tempo gasto. Por isso, vou simplesmente deixá-las aqui sem repetir a explicação:

```
double tempoGasto = 1.5;  
System.out.println("O tempo gasto foi de "  
    + tempoGasto + " horas.");
```


Quase no fim, o que é feito é a aplicação da fórmula que resulta na velocidade média e também é feita uma impressão explicando o que foi feito para encontrar a velocidade média:

```
double velocidadeMedia = distanciaPercorrida / tempoGasto;
System.out.println("Para encontrar a velocidade média "
    + "dividimos a distância (" + distanciaPercorrida
    + ") pelo tempo gasto no "
    + "percurso (" + tempoGasto + ").");
```

Depois de calculada a velocidade é feita a exibição do resultado no terminal:

```
System.out.println("A velocidade média é de: " + velocidadeMedia + "Km/h");
```

A impressão acima encerra nosso primeiro programa.

1.7. Fazendo comentários em um programa Java

Tem um recurso no Java - e em todas as outras linguagens de programação - que vou utilizar algumas vezes durante os algoritmos do livro. É o recurso de comentários.

Comentário é algo que você escreve dentro de um programa, mas é ignorado pelo compilador. Eles servem para ajudar a documentar seu algoritmo.

Podemos fazer comentários de bloco e também de linha. O de bloco pode ser feito assim:

```
/*
Comentário aqui.
Podemos ter quantas linhas quisermos.
*/
```

Algumas vezes esse comentário de bloco pode vir melhor formatado, como é o caso abaixo:

```
/**
 * Comentário melhor formatado.
 * Podemos ter quantas linhas quisermos.
 */
```

Esses asteriscos que ficam no meio são somente para ajudar na formatação e eles não tem efeito técnico. O que importa mesmo é o que fica entre “/” e “*/”.

Podemos também fazer um comentário de linha. Esses devem começar com duas barras. Veja:

```
// Comentário aqui.
```

```
// Para ter outra linha, preciso colocar duas barras novamente.
```

Ao longo do livro você verá exemplos deles dentro dos programas.

Capítulo 2

Variáveis e constantes

Entramos no primeiro conceito específico de lógica de programação. É o momento de saber o que são variáveis, constantes e o papel deles dentro do algoritmo computacional.

Variáveis e constantes tem a função de guardar os valores que são manipulados durante todo o algoritmo. Eles são os responsáveis por dar todo o dinamismo que temos dentro das linguagens de programação. Isso porque, ao invés de utilizar o valor diretamente, fazemos o uso das variáveis ou constantes. Observe o exemplo:

```
double distanciaPercorrida = 250.0;
```

A variável acima nos ajuda a fazer referência para o valor - 250, no caso - nos outros pontos do algoritmo onde forem necessários. Daí não precisamos ficar escrevendo o número 250 em vários lugares. Só escrevemos o 250 no momento da declaração da variável. O melhor é que, se o número mudar de 250 para 251, nós precisamos alterar somente no momento da declaração.

Uma outra coisa que é necessário conhecer para entender melhor a variável é o que são tipos, e, mais especificamente, o tipo *double*. Adiantando, o tipo *double* representa os números decimais dentro do Java. Veremos mais detalhes sobre esse assunto no próximo tópico.

Agora, qual a diferença entre uma variável e uma constante? Na verdade, a diferença está no nome de cada um: uma variável pode variar seu valor e uma

constante não pode. Na prática, tudo que declaramos até aqui - como a variável `distanciaPercorrida` - foram variáveis, e uma constante você declara assim:

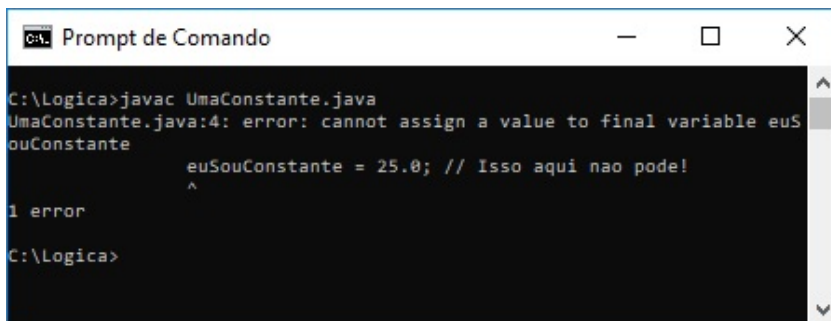
```
final double distanciaPercorrida = 250.0;
```

Reparou a diferença? Foi somente o uso da palavra `final` no início da declaração.

Caso você insista, conscientemente ou não, em mudar o valor de uma constante, e crie um programa como abaixo:

```
public class UmaConstante {  
  
    public static void main(String[] args) {  
        final double euSouConstante = 10.0;  
        euSouConstante = 25.0; // Isso aqui não pode!  
    }  
}
```

... então, logo que tentar utilizar o comando `javac`, já vai obter o erro abaixo:



The screenshot shows a Windows Command Prompt window titled "Prompt de Comando". The command executed is `C:\Logica>javac UmaConstante.java`. The output shows an error on line 4: `UmaConstante.java:4: error: cannot assign a value to final variable euSouConstante`. The error points to the line `euSouConstante = 25.0; // Isso aqui nao pode!`. Below the error message, it says "1 error" and the prompt returns to `C:\Logica>`.

Repare que houve um erro na linha 5 e é justamente nela que foi feita a tentativa de alterar o valor da constante declarada.

Já com variáveis, é possível e, muitas vezes, necessário fazer esse tipo de coisa:

```
public class UmaVariavel {  
  
    public static void main(String[] args) {  
        double euSouVariavel = 10.0;  
        euSouVariavel = 25.0;  
        System.out.println("Fim");  
    }  
}
```

```
    }  
}
```

O programa acima seria compilado (utilizando o *javac*) e executado (utilizando-se o comando *java*) tranquilamente.

Note também que para alterar o valor da variável eu não preciso repetir o tipo:

```
double euSouVariavel = 10.0;  
euSouVariavel = 25.0;
```

Caso o tipo fosse repetido, o Java iria entender como se você estivesse declarando duas variáveis com o mesmo nome, e isso é errado. Se o programa fosse assim:

```
public static void main(String[] args) {  
    double euSouVariavel = 10.0;  
    double euSouVariavel = 25.0;  
    System.out.println("Fim");  
}
```

... ele nem iria passar pela compilação (aquela feita com *javac*).

Antes que você pense que não faz sentido criar uma variável com um valor e já alterar esse valor na linha logo abaixo, eu já vou avisar que, realmente, não faz sentido. Isso foi só para mostrar para você o que dá, tecnicamente, para fazer.

A alteração do valor de uma variável faria mais sentido se combinada com uma estrutura de decisão. Nós veremos sobre isso mais para frente, por ora, fique com o nosso exemplo completo sobre variáveis:

```
public class Variaveis {  
  
    public static void main(String[] args) {  
        System.out.println("=====");  
  
        // Altere o valor da variável abaixo  
        // para realizar testes.  
        double precoProduto = 115.0;  
        System.out.println("O preço do produto é de "  
            + precoProduto + " reais");  
  
        double percentualDesconto = 5.0;
```

```

System.out.println("O desconto será de "
    + percentualDesconto + "%.");

// Regra de três para descobrir qual o valor
// do desconto.
double desconto = precoProduto * percentualDesconto / 100;

// Preço do produto menos o desconto para
// sabermos o valor do produto com o
// desconto que foi calculado na linha anterior.
double precoComDesconto = precoProduto - desconto;

System.out.println("Com desconto, o produto sai "
    + "por R$" + precoComDesconto + ".");

System.out.println("Fim!");
System.out.println("=====");
}
}

```

Faça uma análise do algoritmo acima, execute ele algumas vezes alterando o valor das variáveis `precoProduto` e `percentualDesconto`, e observe os resultados para preços e descontos diferentes.

2.1. Tipos de dados existentes

No Java nós temos vários e vários tipos de dados, mas dentre esses muitos tipos, temos alguns que são principais. Antes de mencioná-los, vou agrupá-los em 3 categorias:

- Numéricos
- Textos
- Lógicos

Dentro dos numéricos nós ainda temos dois subgrupos:

- Inteiros
- Decimais

Quanto aos tipos inteiros, nós temos:

- `byte` (armazena valores de -128 até 127)
- `short` (vai de -32768 até 32767)
- `int` (ocupa o intervalo de -2^{31} a $2^{31}-1$)
- `long` (intervalo de -2^{63} a $2^{63}-1$)

Sobre a diferença, como você pode notar acima, basicamente, é a capacidade de cada um.

Para número inteiros, no livro, nós iremos adotar o tipo *int*. Ficaremos com o *int* porque ele é um tipo muito utilizado quando se deseja fazer referências aos números inteiros.

Como exemplo de uma variável do tipo *int*, podemos ter:

```
int idade = 18;
```

Já os decimais, são:

- `float`
- `double`

No caso dos decimais, aqui no livro, iremos utilizar o *double* - como você já devia estar esperando por já termos feito isso no primeiro algoritmo que criamos. Esse tipo *double* dispensa exemplos, pois já foi mostrado anteriormente.

Com relação ao grupo de tipos que armazenam texto, temos:

- `char` (armazena um carácter por vez)
- `String` (armazena textos de qualquer tamanho)

Dentre esses dois, o *String* é muito mais útil para nós trabalharmos durante os nossos algoritmos. Isso é simplesmente porque o *char* só consegue armazenar um carácter por vez.

Como exemplo de *String*, podemos ter:

```
String nome = "João da Silva";
```

Note que o texto - o nome João da Silva, no caso acima - deve ficar entre aspas duplas, caso contrário, resultaria em erros.

Por fim, no grupo referente ao tipo lógico, temos somente o tipo:

- `boolean`

Esse é um tipo importante porque nos ajuda com o gerenciamento das decisões que nosso algoritmo tem ou pode tomar.

O tipo *boolean* pode assumir somente dois valores: ou ele é verdadeiro ou falso. O valor para verdadeiro é representado como *true* e o falso como *false*. Exemplos:

```
boolean euSouVerdadeiro = true;  
boolean euSouFalso = false;
```

Iremos ver melhor sobre isso quando estivermos falando sobre operadores lógicos e também sobre as estruturas de decisão.

2.2. Escolhendo bons nomes para variáveis e constantes

Não é difícil encontrar por aí, programadores que armazenariam um valor referente a uma distância percorrida em uma variável com o nome *disPer* - sendo *dis* referente a distância e *Per* referente a percorrida:

```
double disPer = 250.0;
```

Não ficou um nome tão ruim, mas pense você... Se não tivesse sido comentado que o valor é referente a distância percorrida, será que você saberia o que significa? Desculpa, mas a minha aposta é que não, não saberia.

O pior nem são os nomes abreviados como ficou a variável declarada acima. Não é tão difícil encontrar casos também de variáveis nomeadas assim:

```
double disper = 250.0;
```

Repare que está tudo em caixa baixa, ou seja, em minúsculo, e isso deixa parecendo que o nome da variável é composto por uma palavra somente,

aumentando a chance de acontecer uma confusão quanto ao significado da variável e, por conseguinte, ao uso dessa variável dentro do algoritmo.

Para fechar, em um último exemplo que também é ruim, tem programadores que usariam, simplesmente, as iniciais *dp* para nomear essa variável:

```
double dp = 250.0;
```

... dificultando entender, ainda mais, o significado que ela tem dentro de todo o algoritmo.

Na verdade, não estou, de forma alguma, proibindo você de usar nomes de variáveis como acabei de apresentar acima, ou seja, abreviadas. Quem sou eu para fazer isso? :)

Minha intenção é mostrar para você que o nome de uma variável é muito importante na composição de um algoritmo e, por isso, você precisa tomar cuidado com os nomes que escolhe para elas.

Em alguns poucos casos - onde temos algoritmos pequenos - a abreviação é aceitável, mas caso fique na dúvida, aconselho que não abrevie. Com o tempo você vai conseguir identificar onde seria legal abreviar e onde não seria.

E qual seria o motivo de nos preocuparmos com os nomes de nossas variáveis? O primeiro motivo é que nossa memória pode falhar, ou seja, mesmo que você crie um algoritmo que só você vai mexer, caso você termine ele e precise voltar um mês depois para alguma alteração, provavelmente, terá dificuldades em lembrar de algumas coisas, o que inclui o porquê de ter declarado determinadas variáveis dentro dele.

O segundo e principal motivo é que muitas vezes você não irá criar um algoritmo sozinho e, muito menos, um software todo sozinho. Claro que é possível e você pode fazer isso, mas em boa parte das vezes - eu diria, a maioria - você vai criar software juntamente com outras pessoas - ainda mais se for trabalhar como funcionário em uma empresa onde, quase certo, fará parte de uma equipe com outros programadores.

Para que tenha uma referência a seguir, procure não abreviar as palavras caso o nome da sua variável contenha 3 palavras ou menos. Se passar disso, caso você

queira, pode abreviar algumas delas, mas sempre com muito cuidado para não prejudicar o entendimento do que sua variável representa no algoritmo como um todo.

Como última dica, use o padrão de nomes chamado *camelCase*. Esse é um padrão que diz que, em nome de variáveis compostas por mais de uma palavra, devemos, a partir da segunda palavra, iniciá-la com a primeira letra em maiúscula, como foi o caso daquelas que declaramos em nosso algoritmo. Veja alguns outros exemplos:

```
double distanciaPercorrida = 0.0;
double tempoGasto = 0.0;
int idadeMinimaParaTirarCarteiraDeMotorista = 18;
String nomeDoProduto = "Caneca de alumínio 500ML";
```

Procure dar bons nomes para suas variáveis sempre. Não fique com preguiça só porque você acha que o nome ficará grande. Você vai ver, principalmente quando começar a criar programas maiores, que vale a pena.

Capítulo 3

Entrada de dados

O que torna nossos algoritmos, realmente, dinâmicos é o fato de podemos receber dados externos a ele.

No algoritmo passado quando declaramos uma variável, como a que está abaixo, nós já guardamos um valor arbitrário nela (o valor 150, no caso).

```
double distanciaPercorrida = 150;
```

Mas, para tornar o nosso algoritmo mais poderoso, o legal é que o valor da variável pudesse ser digitado, através do terminal, pela pessoa que está utilizando nosso programa (o usuário). Dentro do Java, temos algumas técnicas para conseguir isso e veremos, nesse capítulo, a melhor delas para nosso caso.

3.1. Lendo informações com o Scanner

O Scanner é uma funcionalidade do Java que nos permite, de uma maneira fácil, capturar um valor do usuário do nosso programa através do terminal. Veja abaixo como utilizar:

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

```
double distanciaPercorrida = scanner.nextDouble();
```

```
scanner.close();
```

Vamos entender agora o trecho acima. A primeira linha está declarando uma variável chamada `scanner` (mesmo nome que o recurso que estamos utilizando, mas, como é o nome da variável, então podemos deixar com o “S” inicial em minúsculo), simplesmente, para preparar o recurso, para que ele possa ser utilizado:

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

Quando a execução chega na segunda linha do trecho de exemplo, ela para e espera que o usuário informe um valor e aperte a tecla ENTER:

```
double distanciaPercorrida = scanner.nextDouble();
```

Observe, ainda, que o trecho que, realmente, faz o programa parar e esperar por um valor, que será digitado pelo usuário, é esse:

```
scanner.nextDouble();
```

A terceira e última linha é uma necessidade de encerramento formal que a funcionalidade `Scanner` possui. Essa funcionalidade precisa ser encerrada explicitamente e isso é feito dessa forma:

```
scanner.close();
```

Muito importante também notar que precisamos avisar ao Java que vamos utilizar essa funcionalidade. Isso é feito através de uma técnica que chamamos de importação (repare a palavra *import* abaixo):

```
import java.util.Scanner;
```

A linha acima deve ficar bem no início do arquivo. Vamos a um exemplo completo para que você entenda melhor:

```
import java.util.Scanner;
```

```
public class LendoInformacoesComScanner {  
  
    public static void main(String[] args) {  
        System.out.println("=====");  
  
        Scanner scanner = new Scanner(System.in);
```

```

System.out.print("Digite uma distância: ");
double distanciaPercorrida = scanner.nextDouble();

System.out.println("A distância digitada foi "
    + distanciaPercorrida + "Km.");

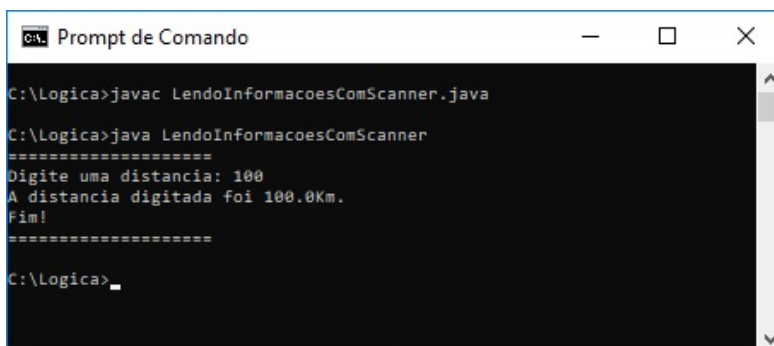
scanner.close();

System.out.println("Fim!");
System.out.println("=====");
}
}

```

Antes de prosseguir quero que note o método `System.out.print`, pois, é a primeira vez que usamos ele. A diferença do `println` para o `print` é que a impressão feita da primeira forma exibe um texto no terminal e depois passa para a próxima linha enquanto que, o `print`, imprime o texto e deixa o cursor na mesma linha.

A alternativa de impressão com `print` é útil quando queremos pedir que o usuário informe alguma coisa no terminal. Considerando o exemplo acima, o cursor vai se posicionar logo após os ":" para que o usuário digite uma distância. Veja no exemplo abaixo onde foi informado o valor 510, ele foi digitado logo após o sinal ":" o que não seria possível se a impressão desse texto tivesse sido feita com `println`.



```

C:\Logica>javac LendoInformacoesComScanner.java

C:\Logica>java LendoInformacoesComScanner
=====
Digite uma distancia: 100
A distancia digitada foi 100.0Km.
Fim!
=====

C:\Logica>_

```

3.2. Lendo diferentes tipos de dados

Como vimos no capítulo sobre variáveis, podemos declarar variáveis de diferentes tipos (numéricos, textos, booleanos, etc.). De forma parecida, podemos ler, através do terminal, diferentes tipos também.

Para que você entenda o necessário para continuarmos com o conteúdo de lógica de programação aqui do livro, acredito que um simples exemplo já será o suficiente. Vamos a ele:

```
import java.util.Scanner;

public class LendoDiferentesTiposDeDados {

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

        Scanner scanner = new Scanner(System.in);

        // -----

        System.out.print(" >>> Digite um texto: ");
        String texto = scanner.nextLine();
        System.out.println("O texto digitado "
            + "foi " + texto);

        // -----

        // Apesar do Java trabalhar com o "." como
        // separador de decimais, o Scanner vai receber
        // um decimal no formato em que seu computador estiver
        // configurado, ou seja, se você usa o
        // português brasileiro, então, para digitar,
        // terá que usar a vírgula como separador
        // de decimais. A própria funcionalidade Scanner vai
        // transformar o número com vírgula em algo
        // que o Java entenda, ou seja, um número
        // que tem o "." como separador de decimais.
        System.out.print(" >>> Digite um decimal: ");
        double numeroDecimal = scanner.nextDouble();
        System.out.println("O número decimal digitado "
            + "foi " + numeroDecimal);
    }
}
```

```

// -----

System.out.print(" >>> Digite um inteiro: ");
int numeroInteiro = scanner.nextInt();
System.out.println("O número inteiro digitado "
    + "foi " + numeroInteiro);

// -----

System.out.print(" >>> Digite um boolean (true ou false): ");
boolean booleano = scanner.nextBoolean();
System.out.println("O booleano digitado "
    + "foi " + booleano);

// -----

scanner.close();

System.out.println("Fim!");
System.out.println("=====");
}
}

```

Execute o programa acima várias vezes e execute diferentes valores para cada tipo de dado pedido.

Capítulo 4

Operadores

Os operadores são recursos super importantes dentro da lógica de programação e, claro, das linguagens de programação como um todo.

Eles nos permitem fazer cálculos, estabelecer relações, atribuir valores e tomar decisões dentro dos nossos algoritmos.

Nessa parte do livro veremos com mais detalhes sobre eles e o papel que cada tipo de operador cumpre em nossos algoritmos.

4.1. Operadores aritméticos

Esses são os operadores que nos permitem realizar cálculos matemáticos básicos. São eles:

- + e - (adição e subtração)
- * e / (multiplicação e divisão)
- % (módulo ou resto de uma divisão)

Como esses operadores são aqueles que você aprendeu na matemática e já os conhece, então para ajudar vamos a um pequeno programa com exemplos:

```
public class OperadoresAritmeticos {  
  
    public static void main(String[] args) {  
        System.out.println("=====");  
    }  
}
```



```

int adicao = 5 + 2;
System.out.println("Resultado da adição de 5 com 2: " + adicao);

int subtracao = 5 - 2;
System.out.println("Resultado da subtração de 5 pelo 2: "
    + subtracao);

int multiplicacao = 5 * 2;
System.out.println("Resultado da multiplicação de 5 pelo 2: "
    + multiplicacao);

double divisao = 5.0 / 2.0;
System.out.println("Resultado da divisão de 5 pelo 2: " + divisao);

double moduloOuResto = 5.0 % 2.0;
System.out.println("Resto da divisão de 5 pelo 2: "
    + moduloOuResto);

System.out.println("Fim!");
System.out.println("=====");
}
}

```

O que você precisa fazer agora é criar um arquivo chamado *OperadoresAritmeticos.java* e testar esse exemplo. Fique a vontade também para alterar ele e fazer alguns testes seus.

Uma coisa que você pode fazer também é utilizar variáveis, ao invés de colocar os números diretamente. Algo como:

```

int umNumero = 5;
int outroNumero = 2;
int adicao = umNumero + outroNumero;

```

4.2. Operadores de atribuição

Operadores de atribuição são aqueles que fazem com que um valor qualquer - um número, por exemplo - passe para uma variável.

O mais comum deles nós já utilizamos aqui várias vezes. É o operador “=” (igual). Recapitulando, veja ele sendo utilizado abaixo:

```
String ultimoNome = "Silva";
```

Os outros operadores de atribuição que temos é o operador “=” combinado com algum operador aritmético. São eles:

- +=
- -=
- *=
- /=
- %=

Eles não são essenciais dentro da programação, mas ajudam a simplificar nosso código. Por exemplo, o trecho abaixo:

```
int umNumero = 5;  
umNumero = umNumero + 2; // A variável umNumero vai guardar o valor 7
```

... pode ser substituído por:

```
int umNumero = 5;  
umNumero += 2; // A variável umNumero vai continuar guardando o valor 7
```

Resumindo, o operador “+=” faz com que a variável, que fica do lado esquerdo - a variável *umNumero*, no caso - seja somada com o valor que está do lado direito - o número 2 - e o resultado atualiza o valor que a variável do lado esquerdo estava guardando.

De forma análoga funcionam os outros operadores apresentados acima. Execute o exemplo abaixo para poder colocar em prática o uso deles:

```
public class OperadoresDeAtribuicao {  
  
    public static void main(String[] args) {  
        System.out.println("=====");  
  
        int umNumero = 5;  
  
        umNumero += 2;
```

```

System.out.println("Resultado da adição de 5 com 2: " + umNumero);

umNumero = 5; // Voltando o valor para 5.

umNumero -= 2;
System.out.println("Resultado da subtração de 5 pelo 2: "
    + umNumero);

umNumero = 5; // Voltando o valor para 5.

umNumero *= 2;
System.out.println("Resultado da multiplicação de 5 pelo 2: "
    + umNumero);

umNumero = 5; // Voltando o valor para 5.

umNumero /= 2;
System.out.println("Resultado da divisão de 5 pelo 2: "
    + umNumero);

umNumero = 5; // Voltando o valor para 5.

umNumero %= 2;
System.out.println("Resto da divisão de 5 pelo 2: " + umNumero);

System.out.println("Fim!");
System.out.println("=====");
}
}

```

4.3. Operadores relacionais

Como o próprio nome diz, esses operadores estabelecem uma relação entre um valor e outro. Por exemplo, se eu quiser saber se uma variável é maior que a outra, posso fazer assim:

```

int idadePessoa = 21;
int idadeMinima = 18;

// Leia a expressão abaixo da seguinte forma: "idadePessoa

```

```
// é maior ou igual a idadeMinima?"  
boolean temIdadeMinima = idadePessoa >= idadeMinima;
```

O resultado de uma relação é sempre um valor booleano que podemos atribuir para uma variável do tipo *boolean*. Caso a relação seja verdadeira, o valor gerado por ela será *true* e, caso seja falsa, então o valor gerado será *false*.

Podemos encarar as relações como se fossem perguntas que nossos algoritmos fazem para os valores participantes da relação que está sendo feita.

Por exemplo, a relação acima é o mesmo que perguntar: “O valor da variável *idadePessoa* é maior ou igual que o valor que está na variável *idadeMinima*?”. Caso a resposta seja “sim”, então será gerado o valor *true* e, caso seja “não”, será gerado o valor *false*.

A lista com todos operadores, você pode ver abaixo:

- > (maior que)
- < (menor que)
- >= (maior ou igual que)
- <= (menor ou igual que)
- == (igual a)
- != (diferente de)

Voltando ao assunto de operadores, pegue o exemplo abaixo, execute e procure alterá-lo para criar seus próprios testes:

```
public class OperadoresRelacionais {  
  
    public static void main(String[] args) {  
        System.out.println("=====");  
  
        // Faça testes alterando o valor das duas variáveis abaixo.  
        int primeiraVariavel = 1;  
        int segundaVariavel = 1;  
  
        System.out.println("O valor da primeira variável é "  
            + primeiraVariavel + " e o da segunda é "  
            + segundaVariavel + ".");  
  
        boolean primeiraVariavelEMaiorQueASegunda
```

```

        = primeiraVariavel > segundaVariavel;
System.out.println("Primeira variável é maior que a segunda? "
    + primeiraVariavelEMaiorQueASegunda);

boolean primeiraVariavelEMenorQueASegunda
    = primeiraVariavel < segundaVariavel;
System.out.println("Primeira variável é menor que a segunda? "
    + primeiraVariavelEMenorQueASegunda);

boolean primeiraVariavelEMaiorIgualASegunda
    = primeiraVariavel >= segundaVariavel;
System.out.println("Primeira variável é maior ou igual a segunda? "
    + primeiraVariavelEMaiorIgualASegunda);

boolean primeiraVariavelEMenorIgualASegunda
    = primeiraVariavel <= segundaVariavel;
System.out.println("Primeira variável é menor ou igual a segunda? "
    + primeiraVariavelEMenorIgualASegunda);

boolean primeiraVariavelEIgualASegunda
    = primeiraVariavel == segundaVariavel;
System.out.println("Primeira variável é igual a segunda? "
    + primeiraVariavelEIgualASegunda);

boolean primeiraVariavelEDiferenteDaSegunda
    = primeiraVariavel != segundaVariavel;
System.out.println("Primeira variável é diferente da segunda? "
    + primeiraVariavelEDiferenteDaSegunda);

System.out.println("Fim!");
System.out.println("=====");
    }
}

```

4.4. Operadores lógicos

Os operadores lógicos trabalham com os valores booleanos para facilitar as tomadas de decisões mais complexas dentro dos nossos algoritmos, e assim como no operador relacional, a avaliação de operadores lógicos sempre gera um valor booleano (verdadeiro ou falso).

Temos 3 operadores lógicos:

- `&&` (Chamado de operador E)
- `||` (Chamado de operador Ou)
- `!` (Chamado de operador de negação ou reversão)

Primeiro vou mostrar o operador `&&`. Ele é utilizado para avaliar dois ou mais valores booleanos de uma vez só, sendo que, para ele gerar o valor *true* - ou seja, verdadeiro - será necessário que todos os valores envolvidos também sejam *true*. Se pelo menos um valor avaliado for falso, então o resultado gerado no final será *false*.

Para entender melhor como usar o operador `&&`, suponha um usuário em uma loja virtual - ou e-commerce - que tem como política aplicar um desconto para seus usuários que são VIPs e que fizeram uma compra com um valor alto.

```
boolean usuarioVIP = true;
boolean compraComValorAlto = true;
boolean aplicarDesconto = usuarioVIP && compraComValorAlto;
```

Utilizando o operador lógico `&&`, como na terceira linha do trecho acima, você consegue saber se é para aplicar um desconto ou não. No final das contas, se a variável *aplicarDesconto* estiver guardando o valor *true*, então o desconto é aplicado e, caso contrário, não.

Agora, imagine que essa mesma loja virtual deseje melhorar sua política de compra. Para isso ela decide bonificar os usuários que são VIPs (independente do valor da compra dele) ou os usuários que fizeram compras acima de R\$100,00. Veja como ficaria:

```
boolean aplicarDesconto = usuarioVIP || compraComValorAlto;
```

Dessa forma a variável *aplicarDesconto* vai guardar o valor *true* se o usuário é VIP ou se a compra que estiver sendo feita for em um valor maior que R\$100,00. Nós conseguimos esse comportamento através do uso do operador lógico `||`.

Temos ainda mais um operador lógico que é o `!` (operador de negação). Ele, simplesmente, pega um valor verdadeiro e transforma em falso, e um falso em verdadeiro.

Imagine agora uma outra loja virtual que venda somente produtos alcoólicos. Sendo assim, ela teria que exigir um cadastro para concluir a compra, com a informação referente a idade da pessoa. Com isso, conseguiria fazer a seguinte validação:

```
int idadeMinima = 18;
int idadeUsuario = 17;

boolean menorDeIdade = idadeMinima > idadeUsuario;
boolean permiteConcluirCompra = !menorDeIdade;
```

Para o exemplo acima, o que vai acontecer é que a variável *permiteConcluirCompra* vai receber o valor *false*, apesar da variável *menorDeIdade* ser verdadeira. Isso, claro, por causa do operador lógico de negação - ou reversão.

A variável *permiteConcluirCompra* só receberia o valor *true* se a variável *menorDeIdade* fosse *false*.

A graça dos operadores lógicos está quando utilizamos as estruturas de decisão, e veremos sobre elas nos próximos tópicos. Por ora, faça testes com o que aprendeu até aqui utilizando, como base, o programa abaixo:

```
public class OperadoresLogicos {

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

        // Faça testes alterando o valor das quatro variáveis abaixo.
        // Lembrando, elas podem ser true ou false.
        boolean usuarioVIP = true;
        boolean compraComValorAlto = true;
        boolean menorDeIdade = true;
        boolean temProdutoAlcoolicoNoCarrinho = true;

        System.out.println("Usuário é VIP? " + usuarioVIP);
        System.out.println("A compra tem um valor alto? "
            + compraComValorAlto);
        System.out.println("É menor de idade? " + menorDeIdade);
        System.out.println("Tem produto alcoólico no carrinho? "
            + temProdutoAlcoolicoNoCarrinho);

        System.out.println("=====");
    }
}
```

```

    boolean aplicarDesconto = usuarioVIP && compraComValorAlto;
    System.out.println("O desconto deve ser aplicado? "
        + "(usuarioVIP && compraComValorAlto): "
        + aplicarDesconto);

    aplicarDesconto = usuarioVIP || compraComValorAlto;
    System.out.println("O desconto deve ser aplicado? "
        + "(usuarioVIP || compraComValorAlto): "
        + aplicarDesconto);

    // Leia assim: "se não for menor de idade ou se
    // não tiver produto alcoólico".
    boolean permiteConcluirCompra
        = !menorDeIdade || !temProdutoAlcoolicoNoCarrinho;
    System.out.println("Pode concluir compra? "
        + "(!menorDeIdade || !temProdutoAlcoolicoNoCarrinho): "
        + permiteConcluirCompra);

    System.out.println("Fim!");
    System.out.println("=====");
}
}

```


Capítulo 5

Estruturas de decisão

Extremamente importantes, as estruturas de decisão nos permitem ter fluxos - ou caminhos - alternativos dentro do nosso sistema, e isso é fundamental para conseguirmos criar algoritmos dinâmicos.

Dentro do Java - e em outras linguagens também - nós temos duas estruturas que são: a estrutura *if* e a estrutura *switch*. Veremos sobre elas agora, começando pela estrutura *if*.

5.1. Estrutura “if”

Essa é a estrutura de decisão mais utilizada dentro da programação. Traduzindo, a palavra *if* significa “se”.

Para utilizá-la você precisa de um valor booleano. Esse valor pode vir do literal *true* ou *false*, pode vir de uma expressão relacional, de uma expressão lógica, de uma variável ou da combinação de tudo isso. Aqui alguns exemplos:

```
if (true) { // Literal "true" sendo utilizada.  
    // Bloco de código do if  
}
```

```
if (7 > 2) { // Expressão relacional.  
    // Bloco de código do if  
}
```

```
if (true && true) { // Expressão lógica.
```

```

    // Bloco de código do if
}

if (7 > 2 && 7 > 5) { // Expressão lógica e expressões relacionais.
    // Bloco de código do if
}

if (euSouUmaVariavelBooleana) { // Utilizando variável
    // Bloco de código do if
}

// Variável e expressão lógica
if (euSouUmaVariavelBooleana && euSouOutra) {
    // Bloco de código do if
}

```

Caso a expressão que está entre os parênteses do *if* gerem o valor *true*, então o Java executa também as linhas que estão no bloco de código do *if*, ou seja, entre as chaves.

Uma observação sobre os 4 primeiros exemplos, é que eles funcionam, mas claro, não fazem muito sentido, pois não estão dinâmicos. A expressão “7 > 2”, por exemplo, sempre vai retornar verdadeiro e o bloco de código do *if* sempre será executado. Para dinamizar o *if* você precisa ter, pelo menos, uma variável envolvida. Vejamos:

```

int numero = 7;
if (numero > 2) {
    // Bloco de código do if
}

```

Você pode argumentar agora que a variável ficou estática e que assim não mudou coisa alguma. Estaria certo se dissesse isso, mas a ideia é que agora a variável seja dinamizada através de algum recurso que busque o valor dela externamente ao algoritmo. Inclusive, já foi dado exemplos disso. Olhe:

```

int numero = scanner.nextInt();
if (numero > 2) {
    // Bloco de código do if
}

```

Agora sim, tudo dinâmico! Vamos então para um exemplo mais prático. Nesse exemplo a regra é que será dado um desconto caso o preço de um produto seja de alto valor.

```
double precoProduto = scanner.nextDouble();

boolean produtoAltoValor = precoProduto >= 100;

double percentualDesconto = 0.0;

if (produtoAltoValor) { // Estrutura de decisão if
    // Desconto para produtos de alto valor.
    percentualDesconto = 5.0;
}

// Regra de três
double desconto = (precoProduto * percentualDesconto) / 100;
double precoComDesconto = precoProduto - desconto;

System.out.println("O produto sairá por R$" + precoComDesconto + ".");
```

Repare que, se o nosso *if* tivesse a expressão dentro dele, então não seria necessário ter a variável *produtoAltoValor*, e ele ficaria assim:

```
if (precoProduto >= 100) {
    percentualDesconto = 5.0;
}
```

Não é errado fazer o *if* como está acima, inclusive, é algo bem comum até, mas tem uma vantagem bacana de usar variáveis booleanas que é o fato do nosso algoritmo ficar mais entendível, pois o nome da variável - se bem escolhido - vai explicar melhor o porquê de estar fazendo a relação.

Para melhorar esse exemplo, vamos imaginar agora que seja necessário imprimir, no terminal, uma mensagem caso nenhum desconto seja dado. Poderíamos fazer um outro *if* assim:

```
if (percentualDesconto == 0.0) {
    System.out.println("Não será aplicado desconto algum.");
}
```

Só que nós temos um outro recurso na estrutura de decisão *if* que vai nos ajudar com isso e tornar nosso algoritmo mais organizado. O recurso se chama *else* - ou, em português, “senão”. Veja:

```
// Se tem alto valor, aplique o desconto
if (produtoAltoValor) {
    percentualDesconto = 5.0;
} else { // Senão, não aplique desconto algum.
    System.out.println("Não será aplicado desconto algum.");
}
```

Agora, vou passar o exemplo completo para que você faça seus testes:

```
import java.util.Scanner;

public class EstruturaDeDecisaoIf {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("=====");

        System.out.print("Informe o preço do produto: ");
        double precoProduto = scanner.nextDouble();

        boolean produtoAltoValor = precoProduto >= 100;

        double percentualDesconto = 0.0;

        // Se maior ou igual 100, aplique o desconto.
        if (produtoAltoValor) {
            percentualDesconto = 5.0;
        } else { // Senão, não aplique desconto algum.
            System.out.println("Não será aplicado desconto algum.");
        }

        double desconto = (precoProduto * percentualDesconto) / 100;
        double precoComDesconto = precoProduto - desconto;

        System.out.println("O produto sairá por R$"
            + precoComDesconto + ".");

        System.out.println("Fim!");
    }
}
```

```

        System.out.println("=====");

        scanner.close();
    }
}

```

5.2. Estrutura “switch”

O *switch* é uma estrutura de decisão que, de acordo com determinado valor, executa um bloco de código específico. Ao contrário do *if*, ele lida com valores que não são booleanos.

Vamos supor que você receba, no seu algoritmo, um número de 1 a 7 e precise imprimir qual o dia da semana a que o número se refere (domingo, segunda-feira, terça-feira, etc). Veja como seria:

```

String nomeDoDiaDaSemana = "";

switch(5) { // Usando o 5 diretamente...
    case 1: nomeDoDiaDaSemana = "Domingo";
        break;
    case 2: nomeDoDiaDaSemana = "Segunda-feira";
        break;
    case 3: nomeDoDiaDaSemana = "Terça-feira";
        break;
    case 4: nomeDoDiaDaSemana = "Quarta-feira";
        break;
    case 5: nomeDoDiaDaSemana = "Quinta-feira";
        break;
    case 6: nomeDoDiaDaSemana = "Sexta-feira";
        break;
    case 7: nomeDoDiaDaSemana = "Sábado";
        break;
    default: nomeDoDiaDaSemana = "[Não encontrado!]";
}

System.out.println("O dia da semana é: " + nomeDoDiaDaSemana);

```

Acredito que já tenha entendido bastante do *switch* só com o exemplo, mas algumas coisas não são tão óbvias e veremos sobre elas agora.

A primeira coisa é sobre o *break*, que serve para encerrar o bloco de código de cada *case*. Se você remover todos os comandos *break*, a variável *nomeDoDiaDaSemana* vai sempre terminar o *switch* com o valor “Sábado”. Esse comportamento se dá porque, depois que o *switch* encontra o respectivo *case* referente ao valor avaliado entre os parênteses, este *case* será executado e também todos aqueles que estiverem abaixo dele. No caso do valor 5, a variável *nomeDoDiaDaSemana* vai receber os textos “Quinta-feira”, “Sexta-feira” e, por último, “Sábado”.

A segunda coisa é sobre a opção *default* ou opção padrão. Caso o valor avaliado não seja encontrado em nenhum *case*, então o bloco executado será o bloco referente a opção *default*.

Para testes mais específicos, tem um exemplo completo aqui:

```
import java.util.Scanner;

public class EstruturaDeDecisaoSwitch {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("=====");

        System.out.print("Informe o dia (número): ");
        int diaDaSemana = scanner.nextInt();

        String nomeDoDiaDaSemana = "";

        switch(diaDaSemana) {
            case 1: nomeDoDiaDaSemana = "Domingo";
                    break;
            case 2: nomeDoDiaDaSemana = "Segunda-feira";
                    break;
            case 3: nomeDoDiaDaSemana = "Terça-feira";
                    break;
            case 4: nomeDoDiaDaSemana = "Quarta-feira";
                    break;
            case 5: nomeDoDiaDaSemana = "Quinta-feira";
                    break;
            case 6: nomeDoDiaDaSemana = "Sexta-feira";
```

```

        break;
    case 7: nomeDoDiaDaSemana = "Sábado";
        break;
    default: nomeDoDiaDaSemana = "[Não encontrado!>";
}

System.out.println("0 dia da semana é " + nomeDoDiaDaSemana + ".");

System.out.println("Fim!");
System.out.println("=====");

scanner.close();
}
}

```

Capítulo 6

Vetores

Um vetor é um tipo especial de valor que nos permite agrupar subvalores dentro dele. Se, por um lado, nós trabalhamos até aqui com variáveis que conseguem guardar apenas um valor, por outro, usando vetores, nós podemos guardar vários valores em uma mesma variável.

Eles são muito úteis em conjunto com os laços que aprenderemos em capítulos futuros. Por ora, vou mostrar o que é um vetor de 1 dimensão e como manipular os valores que estão dentro dele.

Sobre as dimensões que um vetor pode ter, teoricamente, elas podem ser infinitas, mas, na grande maioria dos casos, usamos vetores de uma dimensão apenas, e, por isso, isso é o suficiente para nós que estamos começando com a lógica de programação.

6.1. Vetores de uma dimensão

Antes de mostrar um vetor, repare esse exemplo de uma variável que guarda somente um valor:

```
// Um produto apenas  
double precoProduto = 150.0;
```

Nada demais até aqui, não é mesmo? Mas agora compare com o vetor abaixo que vai guardar vários valores em uma só variável:


```
// Aqui temos o preço de 4 produtos.
```

```
double[] precoProdutos = new double[]{ 150.0, 20.0, 10.0, 100.0 };
```

A primeira diferença são os colchetes logo depois que informamos o tipo:

```
double[]
```

A outra diferença é a maneira como nós iniciamos o vetor:

```
new double[]{ };
```

Depois, bastou colocarmos os valores que gostaríamos de guardar no vetor entre as chaves do mesmo e separados por vírgula.

Cada valor que o vetor guarda possui um índice com o qual podemos acessá-lo. Esse índice é atribuído ao valor de acordo com a sua posição dentro do vetor. O que é importante ter em mente sobre isso é que os índices de um vetor começam do número 0 e não do 1, ou seja, o vetor *precoProdutos*, que criamos acima, tem 4 posições, mas seus índices vão de 0 até 3.

Para acessar, por exemplo, o valor 20.0 do nosso vetor, usamos o índice 1 dessa forma:

```
// 0 índice é 1, mas o produto é o segundo,
```

```
// pois, os índices começam do 0.
```

```
double precoProduto = precoProdutos[1];
```

```
// Vai imprimir 20.0
```

```
System.out.println("0 preço do segundo produto é: " + precoProduto);
```

Você também pode alterar o valor de alguma posição dentro do vetor. Veja:

```
// Alterando o valor da posição 2 (de índice 1).
```

```
precoProdutos[1] = 21.0;
```

```
// Vai imprimir 21.0 ao invés de 20.0.
```

```
// Repare também que estou mandando imprimir diretamente
```

```
// pelo array, sem a intermediação de uma variável como
```

```
// foi na impressão do trecho anterior.
```

```
System.out.println("0 preço do segundo produto é: " + precoProdutos[1]);
```

Para mais testes você pode usar o exemplo completo:

```

public class Vetores {

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

        // Faça testes incluindo mais preços aqui no
        // vetor e, depois, altere e imprima diferentes
        // posições.
        double[] precoProdutos = new double[]{ 150.0, 20.0, 10.0, 100.0 };

        // 0 índice é 1, mas o produto é o segundo,
        // pois, os índices começam do 0.
        double precoProduto = precoProdutos[1];

        // Vai imprimir 20.0
        System.out.println("0 preço do segundo produto é " + precoProduto);

        // Alterando o valor da posição 2 (de índice 1).
        precoProdutos[1] = 21.0;
        System.out.println("0 segundo produto teve seu preço alterado.");

        // Vai imprimir 21.0 ao invés de 20.0.
        // Repare também que estou mandando imprimir diretamente
        // pelo array, sem a intermediação de uma variável como
        // foi na impressão do trecho anterior.
        System.out.println("0 preço do segundo produto é " + precoProdutos[1]);

        System.out.println("Fim!");
        System.out.println("=====");
    }
}

```

Capítulo 7

Iteração

A iteração é um recurso muito utilizado dentro da programação e não é à toa. Muitas vezes precisamos fazer um processamento que tem uma estrutura repetida e que só varia quanto ao valor que ele manipula.

Não é raro termos listas com cerca de 1000 registros para serem processados da mesma forma. Imagine se tivéssemos que repetir um trecho de código 1000 vezes só porque o valor muda. Não é nada viável isso. Ainda mais que a lista, na maioria das vezes, varia de tamanho - uma hora 1000 registros, outra hora 580, outra 150 e por aí vai.

Para que você aprenda a lidar com esse tipo de situação, foram criados os tópicos desse capítulo. Um falando do iterador *for* - mais conhecido como laço *for* - e outro para falar do laço *while*.

7.1. Iterando com o laço “while”

O laço *while* - ou enquanto, em português - é utilizado para iterações onde o que importa é alcançar uma determinada condição para que o mesmo se encerre. Veja a estrutura dele:

```
while ( ) {  
    // Bloco do while  
}
```

Entre parênteses é o lugar onde colocamos a condição. Enquanto a condição retornar o valor *true*, as iterações continuam.

Como condição podemos usar qualquer coisa que resulte em um valor booleano. Veja exemplos:

```
while (espacoEmDiscoDisponivel > 0) { // Expressão lógica.  
    // Bloco de código do while.  
  
    /*  
    Nesse while, de alguma forma, o bloco  
    precisaria atualizar o valor da variável  
    espacoEmDiscoDisponivel fazendo com que  
    essa variável se aproxime, a cada iteração,  
    do valor 0. E, quando ela chegar ao valor 0 ou  
    abaixo dele, então as iterações se encerram.  
    */  
}
```

```
while (tentarNovamente) { // Utilizando variável  
    // Bloco de código do while.  
  
    /*  
    Aqui a intenção é que exista alguma regra  
    que, em algum momento, torne o valor da  
    variável tentarNovamente como false para  
    que as iterações se encerrem.  
    */  
}
```

Agora veja um exemplo comum e um pouco mais prático sobre o uso do laço *while*:

```
int iteracao = 0;  
  
while (iteracao < 10) {  
    System.out.println("Iteração de número " + iteracao + ".");  
  
    iteracao = iteracao + 1;  
}
```

No exemplo acima, serão impressas as iterações de 0 até 9, pois, quando a variável iteração chega ao valor 10, então as iterações se encerram.

Para podermos utilizar o laço *while* em um algoritmo real, vamos fazer um joguinho muito simples, é o seguinte: vamos utilizar uma funcionalidade do Java que pega um número de forma aleatória, e a pessoa terá que adivinhá-lo. Olha só:

```
import java.util.Random;
import java.util.Scanner;

public class IterandoComOLacoWhile {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Random random = new Random();

        System.out.println("=====");

        // Aleatórios de 0 até 9
        int numeroAleatorio = random.nextInt(10);
        boolean tentarNovamente = true;

        System.out.println("Aperte CTRL+C, a qualquer momento, para parar.");

        // Enquanto tentarNovamente igual a true, itera novamente.
        while (tentarNovamente) {
            System.out.print("Tente adivinhar o número: ");
            int numero = scanner.nextInt();

            // Repare que a relação abaixo é a de diferença, ou seja,
            // enquanto diferente, retorna true e, por isso,
            // tenta novamente.
            tentarNovamente = numeroAleatorio != numero;

            if (tentarNovamente) {
                System.out.println("Errado!");
            }
        }

        System.out.println("Parabéns! Você adivinhou. Era o número "
            + numeroAleatorio + " mesmo.");

        System.out.println("Fim!");
        System.out.println("=====");
    }
}
```

```
}  
}
```

Crie o seu programa *IterandoComOLacoWhile.java*, inclua o código-fonte acima e divirta-se.

7.2. Iterando com o laço “for”

O laço *for* - ou para, em português - serve para que possamos lidar com as iterações onde o número de vezes que se precisa iterar é conhecido. Veja como é a estrutura de um laço *for*:

```
for ( ; ; ) {  
    // Bloco do laço for  
}
```

Repare que entre os parênteses do *for*, temos 3 posições separadas por ponto e vírgula. A primeira posição é utilizada para incluirmos uma expressão qualquer para iniciar nossas iterações. Veja o exemplo mais comum de se incluir nessa primeira posição:

```
int i = 0;
```

Note acima que o nome da variável ficou apenas como *i*. Destaco isso, pois, nomear variáveis dessa maneira - usando apenas uma vogal - foi uma coisa que desaconselhei que você fizesse. Só que, no caso do laço *for*, isso é uma coisa tão comum que não tem perigo de causar confusão. Você pode ver essa letra “i” como uma abreviação para a palavra “iteração”, pois, é justamente isso que ela vai fazer nesse caso nosso e na grande maioria do que vemos dentro dos programas: controlar qual é o número da iteração corrente.

Na segunda posição, nós colocamos uma expressão que precisa retornar um valor booleano. Essa é a posição onde, na maioria das vezes, colocamos uma relação entre o número da iteração corrente e a quantidade de iterações desejada. Exemplo:

```
i < 10;
```

Com a expressão acima nós avisamos ao laço *for* que, enquanto a variável *i* for menor que 10, então pode iterar, ou seja, vai iterar com o *i* variando de 0 até 9, e quando *i* chegar ao valor 10, as iterações irão parar.

A terceira posição do laço serve para colocarmos uma expressão que será avaliada toda vez que uma iteração acabar. Chamamos ela de “expressão de iteração”. Ocupando essa posição, na maioria das vezes, o que você verá é simplesmente a variável *i* aumentando uma unidade. Exemplo:

```
i = i + 1;
```

E, no Java, nós podemos substituir a expressão acima por:

```
i++
```

... que, nesse caso, é a mesma coisa.

Observe o *for* completo agora:

```
for (int i = 0; i < 10; i++) {  
    System.out.println("Iteração número: " + i);  
}
```

O *for* acima irá resultar em 10 voltas em nosso laço - lembre que, na primeira posição do *for*, o *i* começa com 0 e, a cada volta, ele é incrementado uma unidade.

Agora vamos para um exemplo mais prático. Imagine que você esteja trabalhando em um algoritmo que vai funcionar em um software de loja virtual. Mais especificamente, vamos supor que esteja criando um algoritmo que precise fazer a soma do preço dos produtos que um usuário da loja colocou no carrinho.

Para realizar a tarefa acima é conveniente usar o laço *for*, pois o número de iterações é conhecido, ou seja, o número de iterações é a mesma quantidade de itens que o carrinho de compras possui.

Importante notar que esse número não tem que ser conhecido para nós, seres humanos. Ele deve ser conhecido pelo seu algoritmo, ou seja, caso existir alguma variável no seu algoritmo que contenha o número de iterações, então podemos dizer que essa quantidade é conhecida.

Como exemplo de um caso onde o laço *for* seria útil, veja abaixo:

```
public class IterandoComOLacoFor {

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

        // Altere os valores do vetor abaixo para
        // poder observar o comportamento do laço for
        double[] carrinhoDeCompras
            = new double[] { 57.0, 175.0, 25.0, 10.0 };

        double valorTotalDoCarrinho = 0.0;
        for (int i = 0; i < carrinhoDeCompras.length; i++) {
            valorTotalDoCarrinho += carrinhoDeCompras[i];
            System.out.println(">>>> Iteração " + i);
            System.out.println(">> Valor do produto corrente R$"
                + carrinhoDeCompras[i] + ".");
            System.out.println(">> O total parcial é de R$"
                + valorTotalDoCarrinho + ".");
        }

        System.out.println("O valor total é:" + valorTotalDoCarrinho);

        System.out.println("Fim!");
        System.out.println("=====");
    }
}
```

O que você precisa agora é fazer seus testes e observar o comportamento do laço *for* na prática.

7.3. Quando utilizar o laço “for” ou o laço “while”

Cheguei a mencionar que o *for* serve para quando o número de iterações é conhecido e que o *while* itera até que determinada condição seja atendida.

Mas, uma curiosidade interessante sobre esses laços é que, tecnicamente, é possível fazer as mesmas coisas com os dois. Claro, que vai mudar a estrutura da iteração, mas, no final das contas, existem as mesmas possibilidades para eles.

No final das contas, é mais uma questão de semântica do que de possibilidades. Para tomar a sua decisão de qual vai usar pode usar a dica que já mencionei. Resumindo:

- Número de iterações é conhecido? Então use o *for*;
- A iteração deve acontecer até que se chegue a uma determinada condição? Use o *while*.