

Variáveis e fluxo

Dentro de um bloco, podemos declarar variáveis e usá-las. Em Java, toda variável tem um tipo que não pode ser mudado, uma vez que declarado:

```
tipoDaVariavel nomeDaVariavel;
```

Por exemplo, é possível ter uma `idade` que guarda um número inteiro:

```
int idade;
```

Com isso, você declara a variável `idade`, que passa a existir a partir daquela linha. Ela é do tipo `int`, que guarda um número inteiro. A partir daí, você pode usá-la, primeiramente atribuindo valores.

A linha a seguir é a tradução de: "**idade deve valer quinze**".

```
idade = 15;
```

Comentários em Java

Para fazer um comentário em java, você pode usar o `//` para comentar até o final da linha, ou então usar o `/* */` para comentar o que estiver entre eles.

```
/* comentário daqui,  
ate aqui */
```

```
// uma linha de comentário sobre a idade  
int idade;
```

Além de atribuir, você pode utilizar esse valor. O código a seguir declara novamente a variável `idade` com valor 15 e imprime seu valor na saída padrão através da chamada a `System.out.println`.

```
// declara a idade  
int idade;  
idade = 15;  
  
// imprime a idade  
System.out.println(idade);
```

Por fim, podemos utilizar o valor de uma variável para algum outro propósito, como alterar ou definir uma segunda variável. O código a seguir cria uma variável chamada `idadeNoAnoQueVem` com valor de **idade mais um**.

```
// calcula a idade no ano seguinte
int idadeNoAnoQueVem;
idadeNoAnoQueVem = idade + 1;
```

No mesmo momento que você declara uma variável, também é possível inicializá-la por praticidade:

```
int idade = 15;
```

Você pode usar os operadores `+`, `-`, `/` e `*` para operar com números, sendo eles responsáveis pela adição, subtração, divisão e multiplicação, respectivamente. Além desses operadores básicos, há o operador `%` (módulo) que nada mais é que o **resto de uma divisão inteira**.

Veja alguns exemplos:

```
int quatro = 2 + 2;
int tres = 5 - 2;

int oito = 4 * 2;
int dezesseis = 64 / 4;

int um = 5 % 2; // 5 dividido por 2 dá 2 e tem resto 1;
               // o operador % pega o resto da divisão inteira
```

Como rodar esses códigos?

Você deve colocar esses trechos de código dentro do bloco `main` que vimos no capítulo anterior. Isto é, isso deve ficar no miolo do programa. Use bastante `System.out.println`, dessa forma você pode ver algum resultado, caso contrário, ao executar a aplicação, nada aparecerá.

Por exemplo, para imprimir a `idade` e a `idadeNoAnoQueVem` podemos escrever o seguinte programa de exemplo:

```

class TesteIdade {

    public static void main(String[] args) {
        // imprime a idade
        int idade = 20;
        System.out.println(idade);

        // gera uma idade no ano seguinte
        int idadeNoAnoQueVem;
        idadeNoAnoQueVem = idade + 1;

        // imprime a idade
        System.out.println(idadeNoAnoQueVem);
    }
}

```

Representar números inteiros é fácil, mas como guardar valores reais, tais como frações de números inteiros e outros? Outro tipo de variável muito utilizado é o `double`, que armazena um número com ponto flutuante (e que também pode armazenar um número inteiro).

```

double pi = 3.14;
double x = 5 * 10;

```

O tipo `boolean` armazena um valor verdadeiro ou falso, e só: nada de números, palavras ou endereços, como em algumas outras linguagens.

```

boolean verdade = true;

```

`true` e `false` são palavras reservadas do Java. É comum que um `boolean` seja determinado através de uma **expressão booleana**, isto é, um trecho de código que retorna um booleano, como o exemplo:

```

int idade = 30;
boolean menorDeIdade = idade < 18;

```

O tipo `char` guarda um, e apenas um, caractere. Esse caractere deve estar entre aspas simples. Não se esqueça dessas duas características de uma variável do tipo `char`! Por exemplo, ela não pode guardar um código como `''` pois o vazio não é um caractere!

```
char letra = 'a';  
System.out.println(letra);
```

Variáveis do tipo `char` são pouco usadas no dia a dia. Veremos mais a frente o uso das `Strings`, que usamos constantemente, porém estas não são definidas por um tipo primitivo.

Tipos primitivos e valores

Esses tipos de variáveis são tipos primitivos do Java: o valor que elas guardam são o real conteúdo da variável. Quando você utilizar o **operador de atribuição** = o valor será **copiado**.

```
int i = 5; // i recebe uma cópia do valor 5  
int j = i; // j recebe uma cópia do valor de i  
i = i + 1; // i vira 6, j continua 5
```

Aqui, `i` fica com o valor de 6. Mas e `j`? Na segunda linha, `j` está valendo 5. Quando `i` passa a valer 6, será que `j` também muda de valor? Não, pois o valor de um tipo primitivo sempre é copiado.

Apesar da linha 2 fazer `j = i`, a partir desse momento essas variáveis não tem relação nenhuma: o que acontece com uma, não reflete em nada com a outra.

Outros tipos primitivos

Vimos aqui os tipos primitivos que mais aparecem. O Java tem outros, que são o `byte`, `short`, `long` e `float`.

Cada tipo possui características especiais que, para um programador avançado, podem fazer muita diferença.

Casting e promoção

Alguns valores são incompatíveis se você tentar fazer uma atribuição direta. Enquanto um número real costuma ser representado em uma variável do tipo `double`, tentar atribuir ele a uma variável `int` não funciona porque é um código que diz: "**i deve valer d**", mas não se sabe se `d` realmente é um número inteiro ou não.

```
double d = 3.1415;  
int i = d; // não compila
```

O mesmo ocorre no seguinte trecho:

```
int i = 3.14;
```

O mais interessante, é que nem mesmo o seguinte código compila:

```
double d = 5; // ok, o double pode conter um número inteiro
int i = d; // não compila
```

Apesar de 5 ser um bom valor para um `int`, o compilador não tem como saber que valor estará dentro desse `double` no momento da execução. Esse valor pode ter sido digitado pelo usuário, e ninguém vai garantir que essa conversão ocorra sem perda de valores.

Já no caso a seguir, é o contrário:

```
int i = 5;
double d2 = i;
```

O código acima compila sem problemas, já que um `double` pode guardar um número com ou sem ponto flutuante. Todos os inteiros representados por uma variável do tipo `int` podem ser guardados em uma variável `double`, então não existem problemas no código acima.

Às vezes, precisamos que um número quebrado seja arredondado e armazenado num número inteiro. Para fazer isso sem que haja o erro de compilação, é preciso ordenar que o número quebrado seja **moldado (casted)** como um número inteiro. Esse processo recebe o nome de **casting**.

```
double d3 = 3.14;
int i = (int) d3;
```

O casting foi feito para moldar a variável `d3` como um `int`. O valor de `i` agora é 3.

O mesmo ocorre entre valores `int` e `long`.

```
long x = 10000;
int i = x; // não compila, pois pode estar perdendo informação
```

E, se quisermos realmente fazer isso, fazemos o casting:

```
long x = 10000;
int i = (int) x;
```

Alguns **castings** aparecem também:

```
float x = 0.0;
```

O código acima não compila pois todos os literais com ponto flutuante são considerados `double` pelo Java. E `float` não pode receber um `double` sem perda de informação, para fazer isso funcionar podemos escrever o seguinte:

```
float x = 0.0f;
```

A letra `f`, que pode ser maiúscula ou minúscula, indica que aquele literal deve ser tratado como `float`.

Outro caso, que é mais comum:

```
double d = 5;  
float f = 3;  
  
float x = f + (float) d;
```

Você precisa do casting porque o Java faz as contas e vai armazenando sempre no maior tipo que apareceu durante as operações, no caso o `double`.

E, uma observação: no mínimo, o Java armazena o resultado em um `int`, na hora de fazer as contas.

Até casting com variáveis do tipo `char` podem ocorrer. O único tipo primitivo que não pode ser atribuído a nenhum outro tipo é o `boolean`.

Castings possíveis

Abaixo estão relacionados todos os casts possíveis na linguagem Java, mostrando a conversão **de** um valor **para** outro. A indicação **Impl.** quer dizer que aquele cast é implícito e automático, ou seja, você não precisa indicar o cast explicitamente (lembrando que o tipo `boolean` não pode ser convertido para nenhum outro tipo).

PARA:	byte	short	char	int	long	float	double
DE:							
byte	----	<i>Impl.</i>	(char)	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
short	(byte)	----	(char)	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
char	(byte)	(short)	----	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
int	(byte)	(short)	(char)	----	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
long	(byte)	(short)	(char)	(int)	----	<i>Impl.</i>	<i>Impl.</i>
float	(byte)	(short)	(char)	(int)	(long)	----	<i>Impl.</i>
double	(byte)	(short)	(char)	(int)	(long)	(float)	----

Tamanho dos tipos Na tabela abaixo, estão os tamanhos de cada tipo primitivo do Java.

TIPO	TAMANHO
boolean	1 bit
byte	1 byte
short	2 bytes
char	2 bytes
int	4 bytes
float	4 bytes
long	8 bytes
double	8 bytes

O if e o else

A sintaxe do `if` no Java é a seguinte:

```
if (condicaoBooleana) {
    codigo;
}
```

Uma **condição booleana** é qualquer expressão que retorne `true` ou `false`. Para isso, você pode usar os operadores `<`, `>`, `<=`, `>=` e outros. Um exemplo:

```
int idade = 15;
if (idade < 18) {
```

```
        System.out.println("Não pode entrar");
    }
```

Além disso, você pode usar a cláusula `else` para indicar o comportamento que deve ser executado no caso da expressão booleana ser falsa:

```
int idade = 15;
if (idade < 18) {
    System.out.println("Não pode entrar");
} else {
    System.out.println("Pode entrar");
}
```

Você pode concatenar expressões booleanas através dos operadores lógicos "**E**" e "**OU**". O "**E**" é representado pelo `&&` e o "**OU**" é representado pelo `||`.

Um exemplo seria verificar se ele tem menos de 18 anos **e** se ele não é amigo do dono:

```
int idade = 15;
boolean amigoDoDono = true;
if (idade < 18 && amigoDoDono == false) {
    System.out.println("Não pode entrar");
}
else {
    System.out.println("Pode entrar");
}
```

Esse código poderia ficar ainda mais legível, utilizando-se o operador de negação, o `!`. Esse operador transforma o resultado de uma expressão booleana de `false` para `true` e vice versa.

```
int idade = 15;
boolean amigoDoDono = true;
if (idade < 18 && !amigoDoDono) {
    System.out.println("Não pode entrar");
}
else {
    System.out.println("Pode entrar");
}
```

Repare na linha 3 que o trecho `amigoDoDono == false` virou `!amigoDoDono`. **Eles têm o mesmo valor.**

Para comparar se uma variável tem o **mesmo valor** que outra variável ou valor,

utilizamos o operador `==`. Repare que utilizar o operador `=` dentro de um `if` vai retornar um erro de compilação, já que o operador `=` é o de atribuição.

```
int mes = 1;
if (mes == 1) {
    System.out.println("Você deveria estar de férias");
}
```

O While

O `while` é um comando usado para fazer um **laço (loop)**, isto é, repetir um trecho de código algumas vezes. A ideia é que esse trecho de código seja repetido enquanto uma determinada condição permanecer verdadeira.

```
int idade = 15;
while (idade < 18) {
    System.out.println(idade);
    idade = idade + 1;
}
```

O trecho dentro do bloco do `while` será executado até o momento em que a condição `idade < 18` passe a ser falsa. E isso ocorrerá exatamente no momento em que `idade == 18`, o que não o fará imprimir 18.

```
int i = 0;
while (i < 10) {
    System.out.println(i);
    i = i + 1;
}
```

Já o `while` acima imprime de 0 a 9.

O For

Outro comando de **loop** extremamente utilizado é o `for`. A ideia é a mesma do `while`: fazer um trecho de código ser repetido enquanto uma condição continuar verdadeira. Mas além disso, o `for` isola também um espaço para inicialização de variáveis e o modificador dessas variáveis. Isso faz com que fiquem mais legíveis, as variáveis que são relacionadas ao loop:

```
for (inicializacao; condicao; incremento) {
    codigo;
}
```

Um exemplo é o a seguir:

```
for (int i = 0; i < 10; i = i + 1) {  
    System.out.println("olá!");  
}
```

Repare que esse `for` poderia ser trocado por:

```
int i = 0;  
while (i < 10) {  
    System.out.println("olá!");  
    i = i + 1;  
}
```

Porém, o código do `for` indica claramente que a variável `i` serve, em especial, para controlar a quantidade de laços executados. Quando usar o `for`? Quando usar o `while`? Depende do gosto e da ocasião.

Pós incremento ++

`i = i + 1` pode realmente ser substituído por `i++` quando isolado, porém, em alguns casos, temos essa instrução envolvida em, por exemplo, uma atribuição:

```
int i = 5;  
int x = i++;
```

Qual é o valor de `x`? O de `i`, após essa linha, é 6.

O operador `++`, quando vem após a variável, retorna o valor antigo, e incrementa (pós incremento), fazendo `x` valer 5.

Se você tivesse usado o `++` antes da variável (pré incremento), o resultado seria 6:

```
int i = 5;  
int x = ++i; // aqui x valera 6
```

Controlando loops

Apesar de termos condições booleanas nos nossos laços, em algum momento, podemos decidir parar o loop por algum motivo especial sem que o resto do laço seja executado.

```
for (int i = x; i < y; i++) {
```

```
    if (i % 19 == 0) {
        System.out.println("Achei um número divisível por 19 entre x e y");
        break;
    }
}
```

O código acima vai percorrer os números de x a y e parar quando encontrar um número divisível por 19, uma vez que foi utilizada a palavra chave `break`.

Da mesma maneira, é possível obrigar o loop a executar o próximo laço. Para isso usamos a palavra chave `continue`.

```
for (int i = 0; i < 100; i++) {
    if (i > 50 && i < 60) {
        continue;
    }
    System.out.println(i);
}
```

O código acima não vai imprimir alguns números. (Quais exatamente?)

Escopo das variáveis

No Java, podemos declarar variáveis a qualquer momento. Porém, dependendo de onde você as declarou, ela vai valer de um determinado ponto a outro.

```
// aqui a variável i não existe
int i = 5;
// a partir daqui ela existe
```

O **escopo da variável** é o nome dado ao trecho de código em que aquela variável existe e onde é possível acessá-la.

Quando abrimos um novo bloco com as chaves, as variáveis declaradas ali dentro **só valem até o fim daquele bloco**.

```
// aqui a variável i não existe
int i = 5;
// a partir daqui ela existe
while (condicao) {
    // o i ainda vale aqui
    int j = 7;
    // o j passa a existir
}
// aqui o j não existe mais, mas o i continua dentro do escopo
```

No bloco acima, a variável `j` pára de existir quando termina o bloco onde ela foi declarada. Se você tentar acessar uma variável fora de seu escopo, ocorrerá um erro de compilação.

```
EscopoDeVariavel.java:8: cannot find symbol
symbol   : variable j
location: class EscopoDeVariavel
    System.out.println(j);
                        ^
1 error
```

O mesmo vale para um `if`:

```
if (algumBooleano) {
    int i = 5;
}
else {
    int i = 10;
}
System.out.println(i); // cuidado!
```

Aqui a variável `i` não existe fora do `if` e do `else`! Se você declarar a variável antes do `if`, vai haver outro erro de compilação: dentro do `if` e do `else` a variável está sendo redeclarada! Então o código para compilar e fazer sentido fica:

```
int i;
if (algumBooleano) {
    i = 5;
}
else {
    i = 10;
}
System.out.println(i);
```

Uma situação parecida pode ocorrer com o `for`:

```
for (int i = 0; i < 10; i++) {
    System.out.println("olá!");
}
System.out.println(i); // cuidado!
```

Neste `for`, a variável `i` morre ao seu término, não podendo ser acessada de fora do `for`, gerando um erro de compilação. Se você realmente quer acessar o contador depois do loop terminar, precisa de algo como:

```
int i;
```

```
for (i = 0; i < 10; i++) {  
    System.out.println("olá!");  
}  
System.out.println(i);
```

Um bloco dentro do outro

Um bloco também pode ser declarado dentro de outro. Isto é, um `if` dentro de um `for`, ou um `for` dentro de um `for`, algo como:

```
while (condicao) {  
    for (int i = 0; i < 10; i++) {  
        // código  
    }  
}
```

Para saber mais

- Vimos apenas os comandos mais usados para controle de fluxo. O Java ainda possui o `do..while` e o `switch`. Pesquise sobre eles e diga quando é interessante usar cada um deles.
- Algumas vezes, temos vários laços encadeados. Podemos utilizar o `break` para quebrar o laço mais interno. Mas, se quisermos quebrar um laço mais externo, teremos de encadear diversos `ifs` e seu código ficará uma bagunça. O Java possui um artifício chamado **labeled loops**; pesquise sobre eles.
- O que acontece se você tentar dividir um número inteiro por 0? E por 0.0?
- Existe um caminho entre os tipos primitivos que indicam se há a necessidade ou não de casting entre os tipos. Por exemplo, `int -> long -> double` (um `int` pode ser tratado como um `double`, mas não o contrário). Pesquise (ou teste), e posicione os outros tipos primitivos nesse fluxo.
- Além dos operadores de incremento, existem os de decremento, como `--i` e `i--`. Além desses, você pode usar instruções do tipo `i += x` e `i -= x`, o que essas instruções fazem? Teste.