

Instruções de decisão, ciclos e vetores

Operadores relacionais

Operadores relacionais produzem um resultado do tipo **boolean**.

Os operadores de teste de igualdade (**==**)/desigualdade (**!=**) funcionam com todos os tipos primitivos e objetos.

Os restantes operadores relacionais (**>**, **>=**, **<**, **<=**) funcionam com todos os tipos primitivos exceto **boolean**.

Para objetos, os operadores de teste de (des)igualdade comparam referências e não objetos!

Exemplo:

```
Integer n1 = new Integer(22);  
Integer n2 = new Integer(22);  
System.out.println(n1 == n2); //false  
System.out.println(n1 != n2); //true
```

Operadores relacionais (cont.)

Se, em vez de comparar referências, pretende comparar **conteúdos** de objetos, poderá recorrer ao método **equals()** que existe para todos os objetos.

Este método funciona para todas as classes das bibliotecas Java mas (para já) não funcionará para as suas classes.

Exemplo:

```
Integer n1 = new Integer(22);
Integer n2 = new Integer(22);
System.out.println(n1 == n2); //false
System.out.println(n1.equals(n2)); //true

Pessoa p1 = new Pessoa();
p1.setName("Ana"); p1.setAge(18);
Pessoa p2 = new Pessoa();
p2.setName("Ana"); p2.setAge(18);
System.out.println(p1.equals(p2)); //false
```

Operadores lógicos

Os operadores lógicos (&& - *AND*, || - *OR*, ! - *NOT*) só podem ser aplicados a valores do tipo **boolean** e produzem um resultado do tipo **boolean**.

As expressões contendo operadores lógicos só são avaliadas até ao momento em que seja possível determinar o valor final, i.e. não é garantido que todas as partes da expressão serão executadas.

Exemplo:

```
char code = 'F';
```

```
boolean capitalLetter = (code >= 'A') && (code <= 'Z');  
System.out.println(capitalLetter);
```

Operador ternário

O único operador ternário (**?:**) é também conhecido como operador condicional, ou operador de decisão.

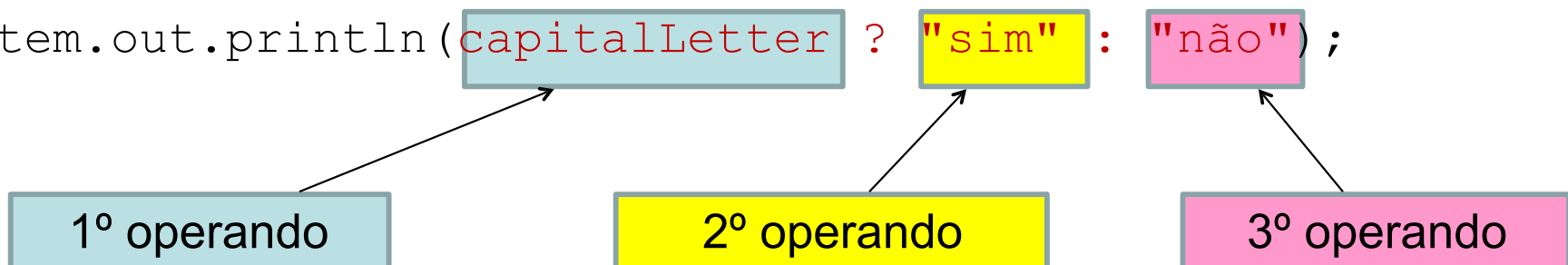
Este operador avalia uma expressão (1º operando) e, caso seja **true**, o resultado é igual ao 2º operando, caso contrário o resultado é igual ao 3º operando.

Exemplo:

```
char code = 'F';
```

```
boolean capitalLetter = (code >= 'A') && (code <= 'Z');
```

```
System.out.println(capitalLetter ? "sim" : "não");
```



Precedência de operadores

A ordem de execução de operadores numa expressão complexa rege-se pelas **regras de precedência**.

```
int a = 5;  
int b = -5;  
int c = ++a & b >>> 30;
```

Para alterar a ordem e/ou clarificar as expressões complexas sugere-se que usem parênteses.

```
c = (++a) & (b >>> 30);
```

Operators	Precedence
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	? :
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

Vetores

Vetores (*arrays*) guardam coleções de primitivas ou referências para objetos de mesmo tipo que ocupam posições consecutivas na memória e são referenciados pelo mesmo nome – nome do vetor.

Vetores são definidos e usados com um par de **[]**. Os elementos individuais são acedidos através de índices. O índice do primeiro elemento é **0**.

Declaração duma referência para um vetor de inteiros:

```
int[] vet1;   ou int vet2[];
```

Agora temos uma referência para um vetor para não alocarmos memória para os inteiros que o vetor deve armazenar.

Inicialização de vetores

A reserva de memória para os elementos do vetor e a sua inicialização é realizada de modos seguintes:

- inicialização com valores por omissão:

```
vet1 = new int[3]; // vetor com 3 elementos: 0, 0 ,0
```

- declaração e inicialização com valores específicos (*aggregate initialization*)

```
int[] vet1 = { 1, 2, 3 }; //vetor com 3 elementos: 1, 2, 3
```

- inicialização dinâmica (*dynamic aggregate initialization*)

```
vet1 = new int[] { 1, 2, 3}; //vetor com 3 elementos: 1,2,3
```


Acesso a elementos do vetor

Uma vez inicializado um vetor, podemos alterar objetos/primitivas que ele guarda mas não é possível alterar o tamanho do vetor (número de elementos).

```
int[] vet1 = { 1, 2, 3 }; //vetor com 3 elementos: 1, 2, 3  
vet1[0] = 11; //vet1 contém 3 elementos: 11, 2, 3  
vet1[2] = 33; //vet1 contém 3 elementos: 11, 2, 33
```

O tamanho de cada vetor pode ser lido com o membro **length**.

```
System.out.println(vet1.length); //3
```

Vetores de objetos

Se criar um vetor de objetos, na realidade cria um vetor de referências para objetos.

```
Integer[] a = new Integer[3];  
a[0] = 10; //autoboxing  
a[1] = 20;  
System.out.println(a[2]); //null  
a[2] = 30;  
System.out.println(a[2]); //30
```

O valor por omissão duma referência é **null**. Todas as referências devem ser inicializadas criando objetos novos (de tipo **Integer** no exemplo).

Sintaxe alternativa com **{}**:

```
Integer[] a = new Integer[] { new Integer(10), 20, 30 };
```

ou

```
Integer[] a = { new Integer(10), 20, 30 };
```

Atribuição de vetores

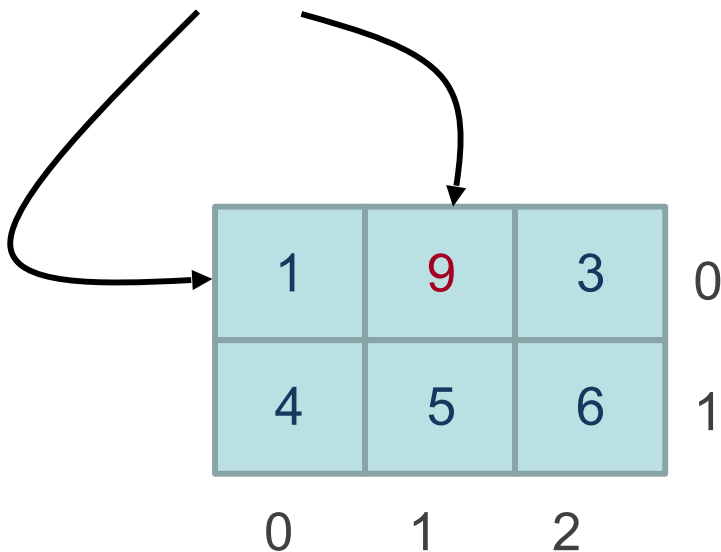
Se atribuir um vetor a outro, já que os vetores são manipulados através de referências, passa a ter duas referências a apontarem para a mesma coleção de objetos.

```
Integer[] a = { 10, 20, 30 };  
Integer[] b = a;  
b[1] = 200;  
System.out.println(a[1]); //200
```

Vetores multidimensionais

É possível criar vetores multidimensionais, i.e. vetores de vetores:

```
int[][] a = { { 1, 2, 3, }, { 4, 5, 6, } } ;  
System.out.println(a.length); //2  
System.out.println(a[0].length); //3  
a[0][1] = 9;
```



Vetores multidimensionais (cont.)

Os vetores que compõem um determinado nível não precisam de ter comprimento igual.



Exemplo:

```
/*um edifício com 3 entradas, número diferente de andares  
por entrada e número variável de apartamentos por andar*/  
int[][] building = new int[3][]; //entradas/andares  
//apartamentos em cada entrada/andar  
building[0] = new int[4];  
building[1] = new int[] { 2, 3, 3, 2, 0};  
building[2] = new int[] { 2, 1, 3};
```

Impressão dos elementos de vetores

Para imprimir todos os elementos dum vetor **unidimensional** pode usar o método **toString** da classe **java.util.Arrays**.

Para imprimir todos os elementos dum vetor **multi-dimensional** pode usar o método **deepToString** da classe **java.util.Arrays**.

```
vet1: [1, 2, 3]
# apartamentos: [[0, 0, 0], [2, 3, 3, 2, 0], [2, 1, 3]]
```

Exemplo:

```
import java.util.Arrays;
...
int[] vet1 = { 1, 2, 3 };
System.out.println("vet1: " + Arrays.toString(vet1)) ;

int[][] building = { {0, 0, 0}, {2, 3, 3, 2, 0}, {2, 1, 3} };
System.out.println("# apartamentos: " +
    Arrays.deepToString(building)) ;
```

Instruções de decisão

Para implementar programas mais complexos, temos a necessidade de executar instruções de forma condicional.

Em Java existem dois tipos de instruções de **decisão/seleção**: **if** e **switch**.

A instrução **if** tem o formato seguinte:

```
if (expressãoBooleana)
    fazer_isto;
else //opcional
    fazer_aquilo;
```

Exemplo:

```
Scanner sc = new Scanner(System.in);
int number = sc.nextInt();
if (number % 2 == 0)
    System.out.println("O número é par");
else
    System.out.println("O número é ímpar");
sc.close();
```

Instrução de decisão if

É possível encadear várias instruções **if**:

```
if (condição1)
    bloco1;
else if (condição2)
    bloco2;
else
    bloco3;
```

Se um bloco inclui mais que uma instrução, deve ser delimitado por parênteses curvos.

Exemplo:

```
int faltas = sc.nextInt(); sc.nextLine();
String regime = sc.nextLine();
if (faltas <= 3)
    System.out.println("Pode ir ao exame teórico.");
else if (!regime.equals("T"))
    System.out.println("Reprovado por faltas.");
else
{ System.out.println("Aluno trabalhador sem a/c.");
  System.out.println("Deve fazer exame prático.");
}
```


Instrução de seleção switch

A instrução **switch** avalia uma expressão integral (inteiro, carater ou **enum**) e, consoante o seu valor, executa um dos comandos:

```
switch (expressão)
{
    case valor1: bloco1; break;
    case valor2: bloco2; break;
    // ...
    default: blocoFinal;
}
```

- O resultado da expressão é pesquisado na lista de alternativas existentes em cada **case**, pela ordem com que são especificados.
- Se a pesquisa for bem sucedida, o bloco de código correspondente é executado. Se houver a instrução **break**, a execução do **switch** termina. Caso contrário serão executadas todas as opções seguintes até que apareça **break** ou seja atingido fim do **switch**.
- Se a pesquisa não for bem sucedida e se o **default** existir, o bloco de código correspondente (`blocoFinal`) é executado.

Instrução de seleção **switch** (cont.)

Exemplo:

```
Scanner sc = new Scanner(System.in);  
int mes = sc.nextInt();  
int dias;  
  
switch (mes)  
{ case 4:  
  case 6:  
  case 9:  
  case 11: dias = 30; break;  
  case 2:  dias = 28; break;  
  default: dias = 31;  
}  
  
System.out.println("Mês tem " + dias + " dias");  
sc.close();
```

Ciclos

Para além da execução condicional de instruções, por vezes existe a necessidade de executar instruções repetidamente.

A um conjunto de instruções que são executadas repetidamente designamos por **ciclo**.

Um ciclo pode ser do tipo **condicional** (**while** e **do...while**) ou do tipo **contador** (**for**).

Normalmente utilizamos ciclos condicionais quando o número de iterações é desconhecido e ciclos do tipo contador quando sabemos à partida o número de iterações.

Ciclo while

O ciclo **while** executa enquanto a condição do ciclo esteja verdadeira. A condição é testada antes de cada iteração do ciclo.

```
while (condição)
    bloco_a_executar;
```

Exemplo:

```
Scanner sc = new Scanner(System.in);
int nota = -1;
while ( (nota > 20) || (nota < 0) )
{
    System.out.println("Insira a nota do aluno.");
    nota = sc.nextInt();
}
sc.close();
```

Ciclo do while

O ciclo **do...while** primeiro executa e só depois testa a condição para verificar se é necessário repetir a execução. A condição é testada no fim de cada iteração do ciclo.

do

 bloco_a_executar;

while (condição);

Exemplo:

```
Scanner sc = new Scanner(System.in);  
int nota;  
do  
{  
    System.out.println("Insira a nota do aluno.");  
    nota = sc.nextInt();  
} while ( (nota > 20) || (nota < 0) );  
sc.close();
```

Ciclo for

O ciclo **for** é mais geral pois suporta todas as situações de execução repetida. Este ciclo, antes da 1ª iteração, faz **inicialização** (só uma vez), depois realiza o **teste** duma condição, executa, e, no fim de cada iteração, faz uma espécie de **atualização**. A condição é testada **no início** de cada iteração do ciclo.

```
for (inicialização; condição; atualização)
    bloco_a_executar;
```

Note o
campo
vazio

Exemplos:

```
for (int i = 1 ; i <= 10 ; i++)
    System.out.println(i + " * " + i + " = " + i*i);
```

```
for (int nota = -1; (nota > 20) || (nota < 0); )
{
    System.out.println("Insira a nota do aluno.");
    nota = sc.nextInt();
}
```

```
nota = 20; //erro de compilação
```

O alcance da variável
nota é limitado ao
corpo do ciclo

Ciclo for (cont.)

Num ciclo **for** é possível omitir inicialização e/ou condição e/ou atualização. Se todos os campos estão vazios, obtemos um ciclo infinito:

```
for ( ; ; )  
    bloco_a_executar;
```

Nos campos de inicialização e atualização é possível incluir vários comandos separados por vírgulas. Estes comandos serão avaliados sequencialmente.

Exemplo:

```
for (int i = 0, j = i + 2; i < 10; i++, j++)  
    System.out.println(i * j);
```

0
3
8
15
24
35
48
63
80
99

Instruções `break` e `continue`

Podemos terminar a execução dum bloco de instruções com duas instruções especiais: `break` e `continue`.

A instrução `break` permite a saída imediata do bloco de código que está a ser executado. É usada normalmente em `switch` e em ciclos, terminando-os.

A instrução `continue` permite terminar a execução da iteração corrente, forçando a passagem para a iteração seguinte (i.e. não termina o ciclo).

Instruções **break** e **continue** (cont.)

Exemplo:

```
for (int nota, tentativa = 1; ; tentativa++)  
{  
    System.out.println("Insira a nota do aluno.");  
    nota = sc.nextInt();  
    if ((nota > 20) || (nota < 0))  
    {  
        System.out.println("Note que a nota não pode ser "  
        + " negativa nem maior que 20.");  
        if (tentativa < 3)  
        {  
            System.out.println("Experimente mais uma vez.");  
            continue;  
        }  
        else  
        {  
            System.out.println("Não tem mais tentativas.");  
            break;  
        }  
    }  
    break;  
}
```

Ciclos com etiquetas (*labels*)

Podemos terminar a execução dum ciclo com as instruções **break** e **continue**.

É possível **aninhar** ciclos uns dentro de outros. Logo deve haver possibilidade de indicar do qual dos ciclos a sair (**break**) ou qual dos ciclos a continuar (**continue**). Por omissão os comandos **break** e **continue** aplicam-se a ciclo mais “próximo”.

Esta funcionalidade é assegurada com **etiquetas** (*labels*) que podemos atribuir a blocos de execução cíclicos.

Os comandos **break** e **continue** podem utilizar uma etiqueta para indicar a que ciclo a que refere a instrução.

Exemplo de ciclos com etiquetas

```
class Labels {  
    public static void main(String[] args) {  
        int[][] arrayInts = { { 1, 2, 3, 4 }, { 22, 33, 44, 55 },  
                               { 23, 45, 78, 12 } };  
  
        int guess = (int) (Math.random() * 100);  
        boolean found = false;  
  
        int i, j = 0;  
        search: for (i = 0; i < arrayInts.length; i++)  
            for (j = 0; j < arrayInts[i].length; j++)  
                if (arrayInts[i][j] == guess) {  
                    found = true;  
                    break search;  
                }  
  
        if (found)  
            System.out.println("Encontrado " + guess +  
                               " na posição " + i + ", " + j);  
        else  
            System.out.println(guess + " não existe");  
    }  
}
```

Ciclo for (sintaxe foreach)

O ciclo **for** quando usado com vetores, tem uma forma mais sucinta (chamada **foreach**).

Exemplo:

```
Scanner sc = new Scanner(System.in);  
double[] a = new double[5];  
for (int i = 0; i < a.length; i++)  
    a[i] = sc.nextDouble();  
  
for (double el : a) /*define a variável el de tipo double  
que receberá sequencialmente todos os elementos do vetor a*/  
    System.out.println(el); //accede o elemento selecionado  
  
sc.close();
```

Geração de números aleatórios

A classe **java.util.Random** permite gerar números pseudo-aleatórios.

Exemplo:

```
Random rand = new Random();  
int randInt = rand.nextInt();  
randInt = rand.nextInt(20); //gera um número entre 0 e 19  
randInt = rand.nextInt(91) + 10; //gera n: 10 <= n <= 100  
double randDouble = rand.nextDouble(); //gera n: 0 <= n < 1.0d
```

Gere aleatoriamente 6 letras maiúsculas do alfabeto Inglês e preenche com elas um vetor 2×3. Imprime de seguida o conteúdo do vetor recorrendo à sintaxe foreach.

Geração de números aleatórios (cont.)

```
Random rand = new Random();  
char [][] arr = new char [2][3];  
for (int linha = 0; linha < arr.length; linha++)  
    for(int coluna = 0; coluna < arr[0].length; coluna++)  
        arr[linha][coluna] = (char)('A' + rand.nextInt(26));  
  
for (char[] linha : arr)  
    for(char el : linha)  
        System.out.println(el);
```

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
0																				
20														!	"	#	\$	%	&	'
40	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~		€		,	f	„	...	†	‡	^	%	Š	€

Exercícios

Qual é a saída do programa seguinte?

```
public static void main(String[] args) {  
    int i = 3;  
    i++;  
    System.out.println(i);  
    ++i;  
    System.out.println(i);  
    System.out.println(++i);  
    System.out.println(i++);  
    System.out.println(i);  
  
    if (i >= 0)  
        if (i == 0)  
            System.out.println("first string");  
        else System.out.println("second string");  
    System.out.println("third string");  
}
```

Exercícios (cont.)

Qual é a saída deste programa?

```
public class BreakContinue {
    public static void main(String[] args) {
        int i = 0;
        outer: for ( ; true; )
            inner: for(; i < 10; i++) {
                System.out.println("i = " + i);
                if (i == 1) { System.out.println("continue");      continue; }
                if (i == 2) {
                    System.out.println("break") ;
                    i++; break;
                }
                if (i == 4) {
                    System.out.println("continue outer");
                    i++; continue outer;
                }
                if(i == 6) {
                    System.out.println("break outer") ; break outer; }
                for(int k = 0; k < 5; k++)
                    if(k == 3) {
                        System.out.println("continue inner");
                        continue inner;
                    }
            }
        }
    }
}
```

i = 0
continue inner
i = 1
continue
i = 2
break
i = 3
continue inner
i = 4
continue outer
i = 5
continue inner
i = 6
break outer