DEI - ISEP

Sintaxe da Linguagem Java

Fernando Mouta

Índice

1 Identificadores

Identificadores em Java:

- São usados para nomes de entidades declaradas, tais como, variáveis, constantes, nomes de classes ou métodos;
- Têm que começar por uma letra, _ ou $, seguida de letras, dígitos ou ambos.

Os identificadores são "case sensitive": upper case ≠ lower case

- Ex.: soma ≠ Soma

As **palavras-chave** da linguagem não podem ser utilizadas como identificadores:

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

Também não podem ser usados como identificadores os literais:

- null
- true
- false

2 Tipos de Dados

Tipos primitivos de dados:

- Contêm um valor único: inteiro, real, carácter ou booliano.

Tipos referência:

- Contêm uma referência para um objeto: arrays, classes ou interfaces.

Todas as variáveis têm que ser declaradas antes de serem usadas e devem ser de um tipo de dados.

O tipo de dados determina os valores que a variável pode ter e as operações que lhe podem ser aplicadas.

Tipos primitivos de dados:

byte	8 bits n.^{os} inteiros entre	–128 e 127
short	16 bits n.^{os} inteiros entre	–32768 e 32767
int	32 bits n.^{os} inteiros entre aprox.	$-2x10^8$ e $2x10^8$
long	64 bits n.^{os} inteiros entre aprox.	$-9x10^{18}$ e $9x10^{18}$
float	32 bits n.^{os} reais precisão simples	(1.4E-45 a 3.4E+38)
double	64 bits n.^{os} reais precisão dupla	(4.9E-324 a 1.7E+308)
char	16 bits caracteres em Unicode	
boolean	1 bit - pode ter o valor true ou false	

Unicode – código de carateres internacional *standard* capaz de representar a maior parte das línguas do mundo escritas.

Os ambientes existentes de Java lêem ASCII e convertem em Unicode.

Os primeiros 256 caracteres do Unicode são o conjunto de caracteres Latinos.

Como poucos editores de texto suportam caracteres Unicode, Java reconhece sequências *escape* \udddd, onde cada d é um dígito hexadecimal.

3 Literais

São os valores que as variáveis podem ter.

Há 4 tipos de literais:

- Numéricos
- De caracteres
- De string
- Lógicos ou booleanos

Literais Numéricos:

Ex.:	178	n.º no sistema decimal
	023	n.º no sistema octal
	0x5A	n.º no sistema hexadecimal
	0.25	n.º real
	2e5	n.º real
	2E-22	n.º real

Constantes inteiras são cadeias de dígitos octais, decimais ou hexadecimais.

O início da constante determina a base do número:

- 0 (zero) denota octal,
- 0x ou 0X denota hexadecimal
- e qualquer outro conjunto de dígitos é assumido como decimal.

Uma constante inteira é do tipo *long* se termina em l ou L (L é preferido porque l confunde-se com 1). Senão é assumida como sendo do tipo *int*.

Se um literal *int* é atribuído diretamente a uma variável *short* ou *byte* e se o seu valor está dentro da gama de valores válidos, o literal inteiro é convertido no respectivo tipo *short* ou *byte*.

Se a atribuição não é direta é necessário explicitar o casting.

Números em vírgula flutuante são expressos com números decimais com ponto decimal e expoente opcionais.

Constantes de vírgula flutuante são assumidas como *double*, a não ser que terminem por f ou F, o que as torna *float*. Se terminam por d ou D especificam uma constante *double*.

Uma constante *double* não pode ser atribuída diretamente a uma variável *float*, mesmo que o valor esteja dentro da gama de valores válidos.

Literais de Caracteres:

São expressos por um carácter único dentro de pelicas. Ex.:

‘a’

‘x’

‘8’

Alguns literais de caracteres representam caracteres que não são impressos ou acessíveis através do teclado – sequências escape:

\n	newline
\t	tab
\b	backspace
\r	carriage return
\\	backslash

Literais de String:

Cadeias de caracteres inserida entre aspas.

Podem conter sequências escape. Ex.:

```
String  s1  =  “Nome \tEndereço”;
```

Em Java uma *string* é um objeto e não é armazenada num *array* como em C.

Quando um literal de *string* é usado, Java armazena esse valor como um objeto String, sem termos de criar explicitamente um novo objeto, como é necessário com outros objetos.

Como strings são objetos, existem métodos para combinar strings, modificar strings e determinar se duas strings têm o mesmo valor.

Literais Booleanos:

São os valores true e false.

4 Variáveis

Localização de memória, utilizada durante a execução de um programa, para conter informação.

Cada variável possui 3 componentes:

- Nome
- Tipo de dados
- Valor

Nome: case sensitive, começa por letra, _ ou $.

Tipo de dados: byte, short, int, long, float, double, char, boolean, array, ou objeto.

Variáveis têm de ser declaradas antes de ser usadas.

Uma variável declarada numa classe (membro de uma classe) é automaticamente inicializada a:

0	variáveis numéricas
'\0'	caracteres
false	booleanos
null	(referência a) objetos

Mas uma variável local (declarada num método) deve ser inicializada explicitamente antes de ser usada num programa, senão dá erro de compilação.

Variáveis locais podem ser declaradas em qualquer lugar dentro de um método.

Ex.:

```
int  a;
int  b,c,d;
int  a = 10, b = 12;
```

Atribuição de valores a variáveis: `a = b = c = 2;`

5 Constantes

Tipo especial de variável cujo valor nunca muda durante a execução do programa.

A declaração tem de ser precedida da palavra-chave **final** e incluir a atribuição de um valor a essa variável.

Ex.:

```
final  int  max  =  10;
final  double  taxa  = 0.17;
```

Scope (âmbito de validade) de uma variável determina quer a visibilidade quer o tempo de vida da variável.

- Ex.:

```
{
      int x = 10;
      //  só x está disponível
      {
            int y = 20;
            //  x  e y estão ambos disponíveis
      }
      //  só x está disponível, y fora de scope
}
```

Em Java não é permitido o seguinte (legal em C e C++):

```
{
      int x = 10;
      {
            int x = 20;
      }
}
```

Resulta um erro ao compilar.

Todas as instruções terminam por **;** exceto a instrução composta ou bloco de instruções:

```
{     instr1;
      instr2;
      instr3;
}
```

6 Comentários

/*	. . .	*/	início e fim de comentário; o comentário pode estender-se por mais que uma linha.
//			tudo à direita deste símbolo até ao fim da linha é comentário.
/**	. . .	*/	início e fim de comentário utilizado para gerar documentação (javadoc).

7 Casting

Java é uma linguagem fortemente tipada que verifica sempre que possível a compatibilidade de tipos em tempo de compilação.

Pode ser necessário realizar conversão de um tipo de dados noutro:

- Em operações de atribuição de um valor a uma variável;
- Em operandos dentro de expressões;
- Ou quando se usam valores como parâmetros de métodos.

Alguns tipos de conversão são feitos automaticamente (implicitamente) enquanto outros têm de ser forçados explicitamente (quando a compatibilidade de um tipo só pode ser determinada em tempo de execução ou quando se pretende efetuar a conversão entre tipos primitivos de dados que diminuem a gama de valores representáveis).

Java permite efetuar o casting de qualquer tipo primitivo de dados para outro qualquer tipo primitivo de dados exceto *boolean*.

7.1 Conversão Implícita

Quando é necessário realizar uma conversão, de um tipo primitivo de dados para outro tipo primitivo de dados que suporte uma maior gama de valores, não é necessário explicitar o casting - geralmente não há perda de informação.

Java efetua a conversão implícita de tipos inteiros em tipos de vírgula flutuante, mas não o inverso.

Não há perda na gama de valores representáveis quando se passa de *long* para *float*, mas nesta conversão implícita pode-se perder precisão, porque um *float* tem 32 bits enquanto um *long* tem 64 bits.

Um *char* pode ser usado onde um *int* seja válido

7.2 Conversão Explícita

Quando é necessário realizar uma conversão de um tipo de dados para outro tipo de dados que apenas suporte uma menor gama de valores temos de efetuar o casting explicitamente porque se corre o risco de perder informação.

```
( tipo ) ( expressão )
```

Ex.:

```
int  x = (int) (y/0.1);
```

Quando um número em vírgula flutuante (*float* ou *double*) é convertido num inteiro, a parte fracional é cortada.

Na conversão entre inteiros os bits mais significativos são cortados.

8 Operadores

8.1 Operadores Aritméticos

+ adição

- subtração

***** multiplicação

/ divisão (se entre inteiros produz resultados inteiros)

% módulo (resto da divisão inteira).

A divisão inteira trunca o resultado, não arredonda.

Operadores abreviados de atribuição:

```
a = a + b;      a += b;
a = a - b;      a -= b;
a = a * b;      a *= b;
a = a / b;      a /= b;
```

Incrementar o valor de uma variável:

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

Em qualquer uma destas notações o valor da variável é incrementado, mas:

- Se o operador ++ é colocado antes da variável,o valor da variável usado na instrução é o novo valor depois de incrementado.
- Se o operador ++ é colocado depois da variável, o valor da variável usado na instrução é o valor antes de incrementado.

Ex.:
```
b = 3;
a = ++b;
```
Depois da execução: a = 4, b = 4.

```
b = 3;
a = b++;
```
Depois da execução: a = 3, b = 4.

Decrementar o valor de uma variável:

```
i  = i - 1;      -- i;      i --;
```

-- i primeiro decrementa e depois usa i.
i -- primeiro usa i e depois decrementa.

8.2 Operadores *Bitwise*

~	**inversão**	– operador unário
&	**and**	– operador binário
|	**or**	– operador binário
^	**exclusive-or**	– operador binário

Estes operadores só realizam operações em valores inteiros.

O operador inversão inverte os bits que constituem o valor inteiro, e os outros operadores realizam a respectiva operação ao nível do bit: o bit de ordem *n* do resultado é calculado usando os bits de ordem *n* de cada operando.

Valores *byte*, *short*, e *char* são convertidos em *int* antes de uma operação bitwise ser aplicada. Ainda se um operador bitwise binário tem um operando *long*, o outro é convertido para *long* antes da operação.

8.3 Operadores *Shift* (Operadores Binários)

<<	left shift
>>	right shift
>>>	unsigned right shift

Estes operadores causam o desvio dos bits do operando da esquerda o número de vezes especificado no outro operando.

Valores byte, short, e char do operando esquerdo são convertidos em int antes da operação ser aplicada. Se o operando esquerdo é um int só os últimos 5 bits do operando direito são considerados (num int, 32 bits, só se pode efetuar desvios 32 vezes); se o operando esquerdo é long só os últimos 6 bits do operando direito são considerados (num long, 64 bits, só se pode efetuar desvios 64 vezes).

O operador << efetua o desvio para a esquerda sendo os bits à direita cheios com 0´s.

O operador >> efetua o desvio para a direita sendo os bits à esquerda cheios com o valor do bit mais à esquerda antes da operação.

O operador >>> efetua o desvio para a direita sendo os bits à esquerda cheios com 0´s.

8.4 Operadores Lógicos

Os operadores lógicos AND, OR, XOR e NOT produzem um valor booleano baseado na relação lógica dos seus argumentos. Só podem ser aplicados a valores booleanos.

AND (E)

expr1 && expr2 ou *expr1 & expr2*
verdadeiro (true) se e só se *expr1* e *expr2* são verdadeiras.

expr1 && expr2: se *expr1* é falsa, *expr2* já não é avaliada.
expr1 & expr2: ambas as expressões são avaliadas.

OR (OU)

expr1 || expr2 ou *expr1 | expr2*
verdadeiro (true) se *expr1* ou *expr2* é verdadeira.

expr1 || expr2: se *expr1* é verdadeira, *expr2* já não é avaliada.
expr1 | expr2: ambas as expressões são avaliadas.

XOR (OU Exclusivo)

expr1 ^ expr2 verdadeiro se apenas uma das expressões é verdadeira.

NOT

! *expr* verdadeiro se *expr* é falsa, e falsa se *expr* é verdadeira.

8.5 Operadores Relacionais (de Comparação)

Os operadores relacionais avaliam a relação entre os valores dos operandos e produzem um valor booleano.

= =	igual
!=	diferente
<	menor que
>	maior que
<=	menor ou igual a
>=	maior ou igual a
instanceof	determina se uma referência a um objeto (o operando esquerdo) é uma instância da classe, interface ou tipo de array especificado no operando direito.

8.6 Operador Ternário Condicional ?

O operador condicional ? : tem 3 operandos:

```
valor = ( exprBool ? expr0 : expr1 ) ;
```

Se *exprBool* é verdadeira, o operador produz o valor de *expr0*, senão produz o valor de *expr1*.

```
if ( exprBool )  valor = expr0;
else  valor = expr1;
```

A diferença principal entre o operador condicional e a instrução if é que o operador condicional produz um valor.

Exemplo:

```
static  int  menor ( int i, int j )  {
      return  i < j  ?  i  :  j ;
}
static  int  menorAlternativo ( int i, int j )  {
      if  (i < j)  return  i ;
            return  j ;
}
```

8.7 Operador + Usado para Concatenar Strings

O operador + pode ser usado para concatenar Strings. Se um dos operandos numa operação + é uma String então o outro operando é convertido para uma String.

Ex.:

```
int  x = 1; y = 2;
String  s= "Valores de x e y: ";
System.out.println( s + x + y );
```

Java possui definições de conversão para String de todos os tipos primitivos de dados. Todos os tipos primitivos de dados são impliciatmente convertidos em objetos String quando usados em expressões deste tipo, mas não noutras alturas. Por exemplo, a um método que recebe um parâmetro String deve ser-lhe passado uma String.

Se o operando a converter para String é um objeto, o seu método toString() é invocado.

8.8 Precedência dos Operadores (Ordem de Avaliação de Expressões)

Hierarquia de precedências (das mais altas para as mais baixas):

```
++  (postfixo),   -- (postfixo)
++  (prefixo),    -- (prefixo),     + (unário), - (unário), ~,    !
(tipo)
*,    /,    %
+ (binário),- (binário)
<<,   >>,   >>>
<,    >,    >=,   >=    instanceof
==,   !=
&
```

```
^
|
&&
||
?:
=,    +=,   -=,   *=,   /=,   %=,          >>=,  <<=,  >>>=, &=,
~=,   !=
```

9 Estruturas de Controlo do Fluxo do Programa

9.1 Estruturas de Decisão

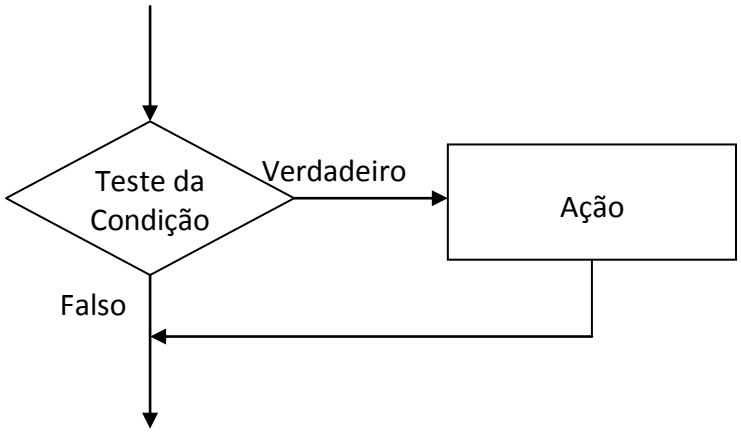
9.1.1 Estrutura *if*

```
if ( cond )  instrução;
```

ou

```
if ( cond )  {
      blocoDeInstruções
 }
```

A condição *cond* é avaliada; se for verdadeira (true) a instrução ou bloco de instruções são executados; se for falsa (false) a instrução ou bloco de instruções são ignorados.



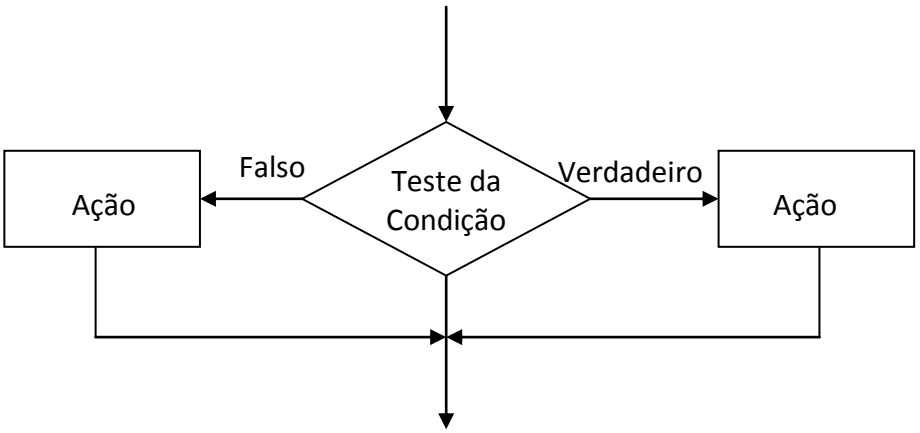
9.1.2 Estrutura *if . . . else*

```
if ( cond )  instrução1;
else instrução2;
```

ou

```
if ( cond )  {
      blocoDeInstruções1
else {
      blocoDeInstruções2
}
```

A condição *cond* é avaliada; se for verdadeira (true) a instrução1 ou o bloco de instruções 1 é executado; se for falsa (false) a instrução2 ou bloco de instruções 2 é executado.



Podem-se encadear vários *if ... else*

```
if ( cond1 )  instr1;
else if ( cond 2)  instr2;
else if ( cond 3)  instr3;
else instr4;
```

Um *else* é sempre associado com o último *if* exceto se se usam chavetas para exprimir de um modo diferente.

Ex.1:

```
if ( cond1 )
if ( cond 2 )  instrA;
else instrB;
```

Ex.2:

```
if ( cond1 )  {
      if ( cond 2 )  instrA;
}
else instrB;
```

No Ex.1 o compilador associa o *else* ao segundo *if* enquanto que no Ex.2 o *else* é associado ao primeiro *if*.

9.1.3 Estrutura *switch*

A estrutura switch é usada para substituir estruturas encadeadas de cláusulas:

```
if . . . else if . . .
```

mas só quando o valor da expressão a testar é do tipo primitivo: **byte**, **short**, **int** ou **char**. Também funciona com tipos **enumerados**, **String**, **Charater**, **Byte**, **Short** e **Integer**.

```
switch ( expr )  {
      case valor1:
            instrução1;
            break;
      case valor2:
            instrução2;
            break;
       . . .
      default:
            instruçãon;
}
```

A instrução *break* causa o abandono da estrutura *switch.*

A instrução *default* é opcional.

Exemplo:

```
int  deci;
switch ( ch )  {
      case '0': case '1': case '2': case '3': case '4':
      case '5': case '6': case '7': case '8': case '9':
            deci = ch - '0';
            break;
      case 'a': case 'b': case 'c': case 'd': case 'e': case 'f':
            deci = ch - 'a' + 10;
            break;
      case 'A': case 'B': case 'C': case 'D': case 'E': case 'F':
            deci = ch - 'A' + 10;
            break;
      default:
            System.out.println("Caracter nao hexadecimal");
}
```

9.2 Estruturas de Repetição

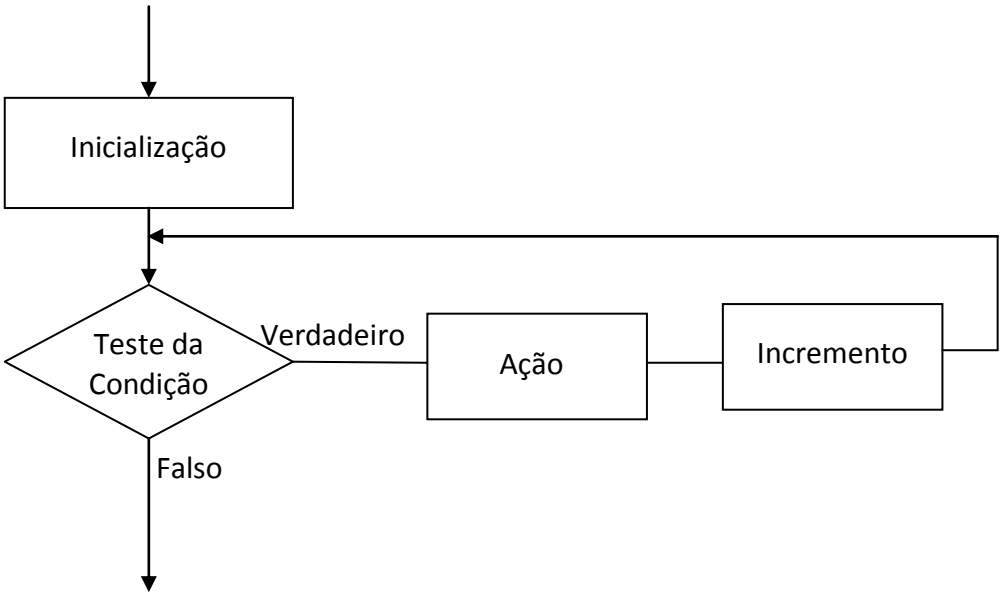
9.2.1 Estrutura *for*

A estrutura *for* é usada para repetir uma instrução um número especificado de vezes até se verificar uma dada condição.

```
for ( inicialização; teste; incremento )  instrução;
```

ou

```
for ( inicialização; teste; incremento )  {
      blocoDeInstruções
}
```



Ex.:

```
for (int i = 1;  i < 5;  i++)  System.out.println(i);
```

Na inicialização e no incremento pode haver mais do que uma instrução separadas por vírgulas, sendo neste caso estas instruções avaliadas sequencialmente.

Ex.:

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

9.2.2 Estrutura *while*

A estrutura *while* é usada para que enquanto uma condição seja verdadeira, uma instrução ou um bloco de instruções sejam executados repetidamente.

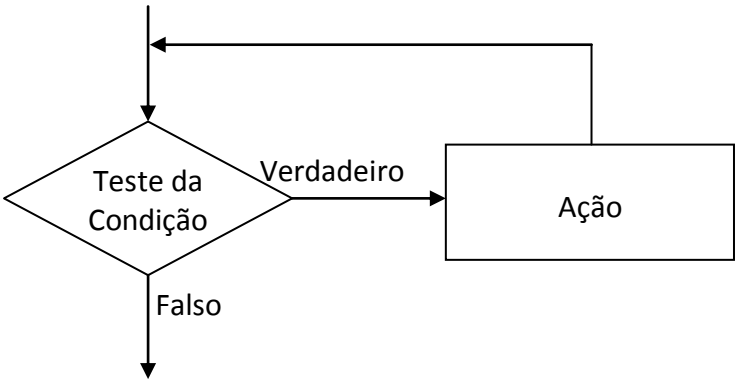
```
while( cond )  instrucao;
```

ou

```
while( cond )  {
      blocoDeInstruções
}
```

Ex:

```
int i = 1;
while( i < 5 ) {
      System.out.println(i);
      i ++;
}
```



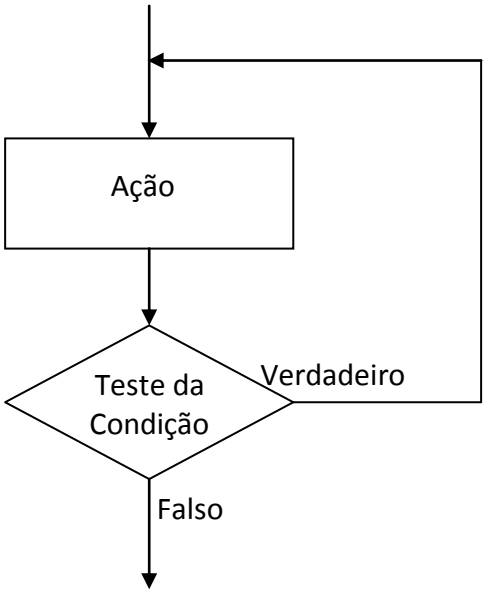
9.2.3 Estrutura *do . . . while*

A estrutura *do . . . while* é usada para repetir uma instrução ou um bloco de instruções enquanto uma condição for verdadeira.

```
do  instrução;
while( cond );
```

ou

```
do  {
      blocoDeInstruções
}  while( cond );
```



A condição só é testada no fim.

A instrução ou bloco de instruções é executado pelo menos uma vez.

9.2.4 Instrução *break*

A instrução *break* interrompe a execução de um ciclo. O programa continua executando as instruções fora desse ciclo.

```
while (cond1) {
      instrA;
      if  (cond2)  break;
      instrB;
}
```

Se *cond2* é verdadeira o ciclo é terminado mesmo que *cond1* se mantenha verdadeira.

A instrução *break* também pode ser utilizada em ciclos *for* e *do*, interrompendo a execução desses ciclos.

9.2.5 Instrução *continue*

A instrução *continue* quando incluída no bloco de instruções de um ciclo, faz com que sejam ignoradas as instruções que se lhe seguem no bloco de instruções iniciando-se um novo ciclo depois do teste da condição de ciclo no caso dos ciclos *while* e *do*, e depois do incremento e do teste no caso do ciclo *for*.

Ex.:

```
while ( cond1 )  {
      instrA;
      if ( cond2 )  continue;
            instrB;
}
```

Se *cond2* for avaliada como verdadeira a *instrB* é ignorada e a *cond1* é avaliada. Se for verdadeira o ciclo repete-se.

Assim pode haver iterações em que a instrB é executada e iterações em que o não é.

No caso de ciclos encadeados o efeito das instruções *break* e *continue* afeta apenas o ciclo onde estão integradas.

Ex.:

```
for (int i=1; i<5; i++)  {
      while  ( cond1 )  {
            instrA;
            while  ( cond2 )  {
                  instrX;
                  if  ( cond3 )  break;
                  instrY;
            }
      }
}
```

Se *cond3* é verdadeira a instrução *break* cancela a execução do ciclo onde está integrada e a execução do programa continua no ciclo *while (cond1)* começando por testar *cond1*.

Suponhamos que se pretendia que fosse continuado o ciclo *for* quando *cond3* fosse verdadeira.

Deve-se usar um *label*, e a instrução *continue label*.

Designemos o *label* por *retomar:*

Ex.:

```
retomar:
for (int i=1; i<5; i++)  {
      while  ( cond1 )
            instrA;
            while  ( cond2 )  {
                  instrX;
                  if  ( cond3 )  continue retomar;
                  instrY;
            }
      }
}
```

continue retomar leva a retomar a execução do ciclo *for*, começando pelo incremento seguido do teste.

Um *label* é um identificador seguido por dois pontos.

Em Java um *label* é usado quando se tem ciclos encaixados e se pretende que o efeito das instruções break ou continue se propaguem através de mais que um nível.

Em Java um *label* coloca-se imediatamente antes de uma instrução de iteração ou *switch*.

Um continue pára a execução da iteração corrente e regressa ao princípio do ciclo para começar uma nova iteração.

Um continue com label salta para o label e entra no ciclo imediatamente a seguir ao label.

Um break abandona o ciclo sem executar o resto das instruções do ciclo.

Um break com label abandona não só o ciclo corrente mas todos os ciclos até ao denotado pelo *label*, o qual também abandona.

Quando um break de um *label* causa a saída de um método pode-se usar simplesmente return.

10 Arrays

As semelhanças entre Java e C relativamente a *arrays* são apenas superficiais e baseadas apenas na sintaxe.

Os *arrays* em C estão muito relacionados com apontadores.

Java não revela os apontadores aos programadores.

Não existem variáveis do tipo apontador. Java trata os apontadores implicitamente.

Arrays são um tipo referência. *Arrays* são objetos.

Quando se declara um *array* obtém-se uma variável que pode conter uma referência a um *array*. Mas ainda é necessário criar o *array*.

Ao declararmos um *array* (assim como outro qualquer objeto) apenas obtemos uma localização de memória que pode conter um apontador para o objeto. Quando se instancia o objeto preenche-se essa localização.

Declaração de *array*:

```
        int numeros [];
ou      int [] numeros;
```

Criação do *array*:

```
numeros = new int [20];
```

Declaração e criação de um *array*:

```
        int numeros [] = new int [20];
ou      int [] numeros = new int [20];
```

Em Java todos estes dados são automaticamente inicializados a:

0	para inteiros;
0.0	para reais;
'\0'	para caracteres;
false	para booleanos;
null	para variáveis referência.

Depois de um *array* ser criado com um dado tamanho, não é possível modificar esse tamanho. O tamanho de um *array* é um campo de dados da classe Array.

Em Java os *arrays* são alocados dinamicamente (em tempo de execução) enquanto em C têm o tamanho fixo em tempo de compilação.

Para alterar o tamanho de um *array* em Java poder-se-ia criar um outro *array* com o tamanho desejado e do mesmo tipo, copiar o conteúdo do *array* para o novo *array* criado ("System.arraycopy(...);") e finalmente copiar a variável referência do novo array para a variável referência do antigo array.

Um array também pode ser criado por inicialização direta dos elementos do *array*:

```
int valores [] = {1, 2, 3, 4, 5, 6};
```

Os literais incluem-se dentro de chavetas.

Uma série de valores dentro de chavetas só podem ser usados em instruções de inicialização e não em instruções de atribuição posteriores.

Os índices dos *arrays* são verificados em tempo de execução (“runtime”).

Se um índice tenta uma referência fora dos limites do array, causa uma exceção e termina a execução.

Os elementos de um array são acedidos do modo usual:

```
valores[0] = 1;
```

O tamanho de um *array* obtem-se referenciando <nomeDoArray>.length

```
int a[] = new int [20];
a.length
```

e não a.length() porque “length” é um “final data field” criado e instanciado quando o objeto *array* é criado. Não é uma chamada a um método.

10.1 *Arrays* de *Arrays*

Não há *arrays* multidimensionais em Java.

Declaração de *arrays* de *arrays*:

```
int matriz [] [];
matriz = new int [2] [3];
```

ou
```
int matriz [] [] = new int [2] [3];
```
ou
```
int [] [] matriz = new int [2] [3];
```

Também se podem criar *arrays* de *arrays* por inicialização direta:

```
int matriz [] [] = { {4, 6, 1}, {8, 1, 2}}
```

Assim

```
matriz[0][0] = 4      matriz[1][0] = 8
matriz[0][1] = 6      matriz[1][1] = 1
matriz[0][2] = 1      matriz[1][2] = 2
```

Em *arrays* de *arrays*, os *arrays* de nível mais baixo não necessitam de ter todos o mesmo tamanho.

Ex.

```
int b [] [] = { {1, 2}, {3, 4, 5}}
```

Assim

```
b[0][0] = 1      b[1][0] = 3
b[0][1] = 2      b[1][1] = 4
                 b[1][2] = 5
```

b[0] tem 2 elementos e b[1] tem 3 elementos.

10.2 Passagem de *Arrays* como Parâmetros na Chamada de Métodos

Os *arrays*, porque são objetos em Java, são passados a métodos através da variável referência.

O nome de um *array* é uma referência para o objeto que contém os elementos do *array* e para a variável de instância "length" que indica o número de elementos do *array*.

Para passar um argumento *array* a um método, especifica-se o nome do *array* sem parênteses retos. Não é necessário passar o tamanho do *array*, porque em Java todo o objeto *Array* conhece o seu próprio tamanho (através da variável de instância "length").

Exemplo:

Escreva um programa que calcule o triângulo de Pascal até uma profundidade de 10, armazenando cada fila do triângulo num *array* de tamanho apropriado e colocando cada *array* correspondente a uma fila num *array* de 10 *arrays* de inteiros.

Um triângulo de Pascal é o seguinte padrão de números inteiros:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
    1   6   15  20  15  6   1
  1   7   21  35  35  21  7   1
```

no qual cada inteiro dentro do triângulo é a soma dos dois inteiros acima dele.

```
public class TrianguloPascal {
  private int triangulo[][];

  /** Cria um triângulo de Pascal até uma profundidade especificada. */
  public TrianguloPascal(int linhas) {
    triangulo = new int[linhas][];
    for (int lin = 0; lin < linhas; lin++) {
      triangulo[lin] = new int[lin + 1];
      if (lin == 0)  triangulo[0][0] = 1;
      else
        for (int col = 0; col <= lin; col++) {
          triangulo[lin][col] = 0;
          /* se não está na extremidade direita, adiciona o nodo de cima
             à direita */
          if (col < lin)  triangulo[lin][col] += triangulo[lin - 1][col];
          /* se não está na extremidade esquerda, adiciona o nodo de cima
            à esquerda */
          if (col > 0)  triangulo[lin][col] += triangulo[lin - 1][col - 1];
        }
    }
  }
  /** Imprime o triangulo de Pascal */
  public void print() {
    for (int i = 0; i < triangulo.length; i++) {
      for (int j = 0; j < triangulo[i].length; j++)
        System.out.print(triangulo[i][j] + "  ");
      System.out.println();
    }
  }

  /** Cria um triângulo de Pascal até uma profundidade de 10 e imprime-o.
*/
  public static void main(String[] args) throws java.io.IOException {
    TrianguloPascal tp = new TrianguloPascal(10);
    tp.print();
    System.in.read();
  }
}
```