Capítulo 2

Tipos, Variáveis e Operadores

Este capítulo irá revelar os conceitos básicos da linguagem Java. Examinaremos aqui os tipos de dados existentes, as regras para declaração de variáveis, os operadores e suas precedências. Como será notado, a sintaxe da linguagem Java é muito semelhante àquela usada pela linguagem C/C++.

2.1 – Tipos de Dados Primitivos

Existem oito tipos básicos de variável para o armazenamento de inteiros, números em ponto flutuante, caracteres e valores booleanos. Estes tipos básicos são denominados tipos primitivos por não serem considerados objetos.

Os tipos primitivos de Java recebem da linguagem um tratamento diferenciado dos demais objetos. A razão por este tratamento especial é que a criação de pequenos objetos, especialmente variáveis simples, acaba não sendo muito eficiente, já que em Java, os objetos são alocados dinamicamente no monte (heap). Por essa razão, Java resgata a abordagem usada por C e C++. Isto é, ao invés de os tipos primitivos serem alocados dinamicamente na região de monte, estas variáveis são alocadas na pilha, o que é muito mais eficiente.

Ao contrário da maior parte das linguagens de programação, os tipos primitivos da linguagem Java possuem o mesmo tamanho e características, independente do sistema operacional ou da plataforma usada.

2.1.1– Tipos de Dados Inteiros

Existem quatro tipos primitivos que podem ser usados na representação de números inteiros, sendo diferenciados pelo tamanho de armazenamento e alcance (menor e maior valor representável).

Tipo	Tamanho	Alcance
Byte	8 bits	-128 até 127
short	16 bits	-32.768 até 32.767
int	32 bits	-2.147.483.648 até 2.147.483.647
long	64 bits	-9223372036854775808 até 9223372036854775807

Tabela 2. 1 – Tipos Inteiros

Todos estes tipos possuem sinal, o que significa que os números armazenados podem ser positivos ou negativos. O tipo usado para uma determinada variável depende do intervalo de valores que ela poderá comportar. Nenhuma variável inteira poderá armazenar valores que sejam grandes ou pequenos demais para seu tipo designado; portanto, você deverá tomar cuidado ao designar tipos às variáveis.

O exemplo seguinte mostra esta situação:

```
// precisaoByte.java
public class precisaoByte {

 public static void main(String[] args) {
  byte i = 127;
  System.out.println("i: " + i);
  i++;
  System.out.println("i: " + i);
 }
}
```

Exemplo 2. 1 – Perda de Precisão em Tipos Inteiros

O programa acima tem como saída:

```
i: 127
i: -128
```

A razão para este estranho resultado foi a tentativa de atribuirmos à variável i um valor mais alto que o maior número comportado pelo seu tipo. Lembre-se que o tipo byte pode representar números de –128 a 127.

2.1.2– Tipos de Dados em Ponto Flutuante

No Java existem duas representações para números em ponto flutuante que se diferenciam pela precisão oferecida: o tipo *float* permite representar valores reais com precisão simples (representação interna de 32 bits) enquanto o tipo *double* oferece dupla precisão (representação interna de 64 bits). Os valores em ponto flutuante do Java estão em conformidade com o padrão IEEE754.

Tipo	Tamanho	Alcance
float	4 bytes	aprox. ±3.402823 E+38F
double	8 bytes	aprox. ±1.79769313486231 E+308

Tabela 2. 2 – Tipos em Ponto Flutuante

Deve ser utilizado o ponto como separador de casas decimais. Quando necessário, expoentes podem ser escritos usando o caractere 'e' ou 'E', como nos seguintes valores:

1.44E6 (= 1.44 x 10^6 = 1,440,000) ou 3.4254e-2 (= 3.4254 x 10^{-2} = 0.034254).

Considere o valor 1/3 representado na notação decimal:

```
0.3333333333333333
```

O resultado dessa divisão é uma dízima periódica. Não há limite para o número de casas decimais requerido para representar exatamente o valor original. No tipo *float*, que possui precisão simples, apenas 7 casas decimais podem ser representadas. Já o tipo double possui precisão de 15 casas decimais.

Note que precisão é algo diferente de "tamanho do número". O tipo float pode representar números tão grandes quanto 3.4E+38, mas o número de casas decimais no número poderá ser de no máximo 7.

Veja o exemplo seguinte:

```
// precisaoFloat.java
public class precisaoFloat {

 public static void main(String[] args) {
  double d = 1.230000875E+2;
  System.out.println(d);
  float f = (float)d;
  System.out.println(f);
 }
}
```

Exemplo 2. 2 - Perda de Precisão em tipos de ponto flutuantes

O número 1.230000875E+2 possui precisão de nove casas, mas a variável f, do tipo float, possui precisão de 7 casas, portanto haverá perda de informação, já que as demais casas serão descartadas e a precisão só será garantida até a sétima casa decimal. O programa acima tem como saída:

```
123.0000875
123.000084
```

2.1.3– Tipos de Dados Caractere

O tipo *char* permite a representação de caracteres individuais. Como o Java utiliza sua representação interna no padrão Unicode, cada caractere ocupa 16 bits (2 bytes) sem sinal, o que permite representar até 32.768 caracteres diferentes, facilitando, assim, a internacionalização de aplicações Java.

Embora teoricamente você possa usar qualquer caractere Unicode em sua aplicação Java, os mesmos só poderão ser exibidos caso o browser (no caso dos applets) ou o sistema operacional (no caso das aplicações) possua suporte a estes caracteres. Por exemplo, você não conseguirá exibir caracteres do alfabeto kanji em uma máquina rodando a versão em português do windows 95. Qualquer tentativa de imprimir um caractere não suportado pelo sistema operacional no modo texto, imprimirá o símbolo de interrogação `'?'`. Por exemplo, no windows 95, qualquer caractere com código acima de 127 resulta em `'?'`. Já no caso de uso de modo gráfico, ainda no windows 95, consegue-se imprimir ainda caracteres até o código 255. A partir daí, é impresso apenas `'□'`.

Embora o tipo primitivo char seja usado para representar caracteres individuais, o conteúdo do mesmo guarda informações referentes ao código unicode do caractere representado. Ao analisarmos a natureza deste código, verificaremos que o código é um número. Podemos, portanto, realizar operações aritméticas em tipos char, como mostra o exemplo abaixo:

```
//charTest.java
public class charTeste {

 public static void main(String[] args) {
  char i = 'a';
  System.out.println("i: " + i);
  i++;
  System.out.println("i: " + i);
  i++;
  System.out.println("i: " + i);
  i = (char)(i * 1.2);
  System.out.println("i: " + i);
  int j = i;
  System.out.println("j: " + j);
 }
}
```

Exemplo 2. 3 – Operações Numéricas com Caracteres

A execução deste programa produzirá a seguinte saída:

```
I: a
I: b
I: c
I: v
I: 118
```

Alguns caracteres são considerados especiais pois não possuem uma representação visual, sendo a maioria caracteres de controle e outros caracteres cujo uso é reservado pela linguagem. Tais caracteres podem ser especificados dentro dos programas como indicado na tabela abaixo, ou seja, precedidos por uma barra invertida (‘\’):

Representação	**Significado**
\n	Pula linha *(newline* ou *linefee*d)
\r	Retorno de carro *(carriage retur*n)
\b	Retrocesso *(backspac*e)
\t	Tabulação *(horizontal tabulatio*n)
\f	Nova página *(formfee*d)
\’	Apóstrofe
\”	Aspas
\\	Barra invertida
\u223d	Caractere UNICODE de código 233d
\137	Octal

Tabela 2. 3 - Representação de Caracteres Especiais

O valor literal de um caractere deve estar delimitado por aspas simples (' ').

2.1.4– Tipos de Dados Lógico

Em Java dispõe-se do tipo lógico *boolean* capaz de assumir os valores *false* (falso) ou *true* (verdadeiro). Deve ser destacado que não existem equivalência entre os valores do tipo lógico e valores inteiros tal como usualmente definido na linguagem C/C++. O exemplo abaixo mostra essa situação:

```
// testeInvalido.Java
public class testeInvalido {

 public static void main(String[] args) {
  int boolInt = 1;
  boolean bool = true;

  if (bool) System.out.println(bool); // imprime true
  if (boolInt) // ERRO: SÛ valores booleanos!
  {
      System.out.println("Olá Mundo!");
  }
  while (i = 5) // ERRO: sÛ expressıes booleanas!
  {
     System.out.println("Isto é um teste")
  }
 }
}
```

Exemplo 2. 4 - Não Equivalência Entre Tipos Lógicos e Inteiros

2.2 – Declaração de Variáveis

Variáveis são alocações de memória nas quais podemos armazenar dados. Para criar uma variável, você precisa dar-lhe um nome e identificar o tipo de informação que ela vai armazenar. Você também poderá atribuir a variável um valor inicial logo no momento de sua criação.

Existem três tipos de variáveis em Java: variáveis de instância, variáveis de classe e variáveis locais. O exemplo abaixo ilustra estas três possibilidades:

```
public class variavel {

 public static int c = 10; // variável de classe
 public int i;  // variável de inst,ncia

 public int func()
  {
   int n = 5;  // variável local
   i = 2*n;
   return (i + c);
  }
}
```

Exemplo 2. 5 – Variáveis locais, de instância e de classe

As variáveis de instância são usadas para definir os atributos de um objeto. As variáveis de classe definem os atributos de uma classe de objetos inteira e aplicam-se a todas as instâncias dela. Já as variáveis locais são usadas dentro de definições de métodos ou mesmo em blocos de instruções menores dentro de um método.

Variáveis podem ser declaradas em qualquer ponto de um programa Java, sendo válidas em no escopo onde foram declaradas e nos escopos internos à estes. Por escopo entende-se o bloco (conjunto de comandos da linguagem) onde ocorreu a declaração da variável. Em Java, os blocos de instruções são delimitados por '{' e '}'. O exemplo abaixo ilustra estas situações:

```
public class testaVisibilidade {

 public static void main(String[] args) {
  int x = 10;  // definição da variável x
  int y = 100; // definição da variável y
  while (x < 100)
   {
     int z = 0; // válido: z visÌvel apenas neste
bloco
    int y = x; // inválido: redefinição da variável y
    x++; // válido: x também é visÌvel neste bloco
   }
 }
}
```

Exemplo 2. 6 - Visibilidade das Variáveis

O nome de uma variável em Java pode ser uma seqüência de um ou mais caracteres alfabéticos e numéricos, iniciados por uma letra ou ainda pelos caracteres '_' *(underscor*e) ou '$' (cifrão). Os nomes não podem conter outros símbolos gráficos, operadores ou espaços em branco, podendo ser arbitrariamente longos embora apenas os primeiros 32 caracteres sejam utilizados para distinguir nomes de diferentes variáveis. É importante ressaltar que as letras minúsculas são consideradas diferentes das letras maiúsculas, ou seja, a linguagem Java é sensível ao caixa empregado, assim temos como exemplos válidos:

a	total	x2	$mine
_especial	TOT	Total	ExpData

Segundo as mesmas regras temos abaixo exemplos inválidos de nomes de variáveis:

1x	Total	geral	numero-minimo	void

A razão destes nomes serem inválidos é simples: o primeiro começa com um algarismo numérico, o segundo possui um espaço em branco e o terceiro contém o operador menos. Mas, por que o quarto nome é inválido? Porque além das regras de formação do nome em si, uma variável não pode utilizar como nome uma palavra reservada da linguagem. As palavras reservadas são os comandos, nomes dos tipos primitivos, especificadores e modificadores pertencentes a sintaxe de uma linguagem.

As palavras reservadas da linguagem Java, as quais não podem ser utilizadas como nome de variáveis ou outros elementos, são:

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

Além destas, existem outras que, embora reservadas, não são utilizadas pela linguagem:

const	*future*	*generic*	*goto*	*inner*
operator	*outer*	*rest*	*var*	*cast*

Algumas destas, tal como o goto, faziam parte da especificação preliminar do *Oa*k, antes de sua formalização como Java. Atualmente, elas não podem ser utilizadas em qualquer programa.

Desta forma para declararmos uma variável devemos seguir a seguinte sintaxe:

```
Tipo nome1 [, nome2 [, nome3 [..., nomeN]]];
```

Ou seja, primeiro indicamos um tipo, depois declaramos uma lista contendo um ou mais nomes de variáveis desejadas deste tipo, onde nesta lista os nomes são separados por vírgulas e a declaração é terminada por ';' (ponto e vírgula). Exemplos:

```
int i;
float total, preço;
byte mascara;
double valorMedio;
```

As variáveis podem ser declaradas individualmente ou em conjunto:

```
char opcao1, opcao2;
```

A declaração anterior equivale as duas declarações abaixo:

```
char opcao1;
char opcao2;
```

Também é possível definirmos um valor inicial para uma variável diretamente em sua declaração como indicado a seguir:

```
int quantidade = 0;
float angulo = 1.57;
```

```
boolean ok = false;
char letra =   c  ;
```

2.3 – Conversões Entre Tipos Numéricos

Java não possui nenhuma restrição quanto à operações envolvendo tipos numéricos diferentes. Por exemplo, a multiplicação de um número do tipo int por um número do tipo double terá como resposta um número do tipo double.

Em geral, qualquer operação binária envolvendo valores numéricos de tipos diferentes serão aceitas e tratadas da seguinte forma:

- Se algum dos operandos for do tipo double, então o outro operando será convertido em um double.
- Caso contrário, se algum dos operandos for do tipo float, o outro operando será convertido em um float.
- Caso contrário, se algum dos operandos for do tipo long, o outro operando será convertido em um long.

Isso funciona de forma análoga para os tipos inteiros: int, short e byte.

Por outro lado, em algumas situações poderá ser necessário considerar um número double como sendo um inteiro. Este tipo de conversão é aceito na linguagem Java, embora alguma informação poderá ser perdida durante este processo. Neste tipo de conversão, em que alguma perda de informação possa ocorrer, é necessário informar o tipo para o qual a conversão será realizada. Esta informação deve estar entre parêntesis e deve preceder o nome da variável que será convertida. Por exemplo:

```
double x = 9.997;
int nx = (int)x;
```

Como a conversão de um tipo ponto-flutuante em um inteiro descarta sua parte fracionária, o valor da variável nx, no exemplo acima é 9. Se você preferir que o número do tipo ponto-flutuante seja arredondado para o inteiro mais próximo, que é a operação mais útil na maior parte dos casos, recomendamos o uso do método `Math.round`:

```
double x = 9.997;
int nx = (int)Math.round(x);
```

Agora a variável nx possui o valor 10. Você ainda precisa explicitar a conversão para int quando você chama o método round. A razão para isso é que o método round retorna um número do tipo long, que só pode ser convertido para int com uma conversão explícita.

O exemplo abaixo ilustra a obrigatoriedade ou não do uso de conversões explícitas na conversão de números do tipo int, float e char. O exemplo também deixa clara a impossibilidade de conversão entre os tipos booleanos em numéricos e vice-versa.

```
public class testaConversoes {
 static public void main (String args[])
 {
  char c = 'a';
  System.out.println("c: " + c);
  int i = c;  // conversão explÌcita desnecessária
  System.out.println("i: " + i);
  float f = c;// conversão explÌcita desnecessária
  System.out.println("f: " + f);
  int j = 101;
  System.out.println("j: " + j);
  char d = (char)j;     //     conversão     explÌcita
obrigatÛria
  System.out.println("d: " + d);
  boolean b = true;
  int k = (int)b; // ERRO: Os tipos são incompatÌveis
 }
}
```

Exemplo 2. 7 – Conversões Numéricas

Uma outra peculiaridade da linguagem Java acontece quando realizarmos algum tipo de operação matemática em tipos primitivos menores que um int (char, byte ou short). Estes valores são "promovidos" para o tipo int antes de a operação matemática ser efetuada e o resultado da operação será do tipo int. Portanto, se você quiser atribuir o valor desta expressão à uma variável cujo tipo seja menor que int, uma conversão explícita deverá ser efetuada. O exemplo abaixo mostra esta promoção ocorrendo no tipo short:

```
void testeShort(short x, short y) {
   x = (short)(x * y);  // conversão obrigatÛria
   x = (short)(x / y);  // conversão obrigatÛria
   x = (short)(x % y);  // conversão obrigatÛria
   x = (short)(x + y);  // conversão obrigatÛria
   x = (short)(x - y);  // conversão obrigatÛria
   x++;  // conversão desnecessária
   x--;  // conversão desnecessária
}
```

Exemplo 2. 8 - A Questão da Promoção

2.4 – Constantes

Em Java, usamos a palavra chave *final* para denotar uma constante. Como por exemplo:

```
public class Constantes {
 static public void main (String args[])
 {
  final double CM_POR_POLEGADA = 2.54; // constante
  double larguraPalel = 8.5;
  double alturaPapel = 11;

  System.out.println("Tamanho em centimetros: " +
     (larguraPalel * CM_POR_POLEGADA) + " por " +
     (alturaPapel * CM_POR_POLEGADA));
 }
}
```

Exemplo 2. 9 - O Uso de Constantes

A palavra chave *final* indica que a associação de valores à variável só poderá ocorrer uma única vez. Geralmente usa-se letras maiúsculas para nomear constantes.

Provavelmente seja mais comum em Java o uso de constantes disponíveis para diversos objetos de uma mesma classe e seus métodos. Estes tipos de constantes também são conhecidas como constantes de classe. Constantes de classe são declaradas com o uso das palavras-chave *static final*. O exemplo seguinte exemplifica o uso de uma constante de classe:

```
// ConstantesDeClasse.java
public class ConstantesDeClasse {
 public static final double G = 9.81; // constante

 static public void main (String args[])
 {
   System.out.println(G + " metros por segundo " +
     "ao quadrado");
 }
}
```

Exemplo 2. 10 - Constantes de Classe

2.5 – Comentários

Comentários são informações incluídas em um programa que não serão usadas para gerar código. Elas servem apenas para proveito dos seres humanos. Embora o compilador Java ignore totalmente os comentários ao preparar uma versão executável de um arquivo-fonte, os mesmos, quando bem usados, aumentam sensivelmente a legibilidade de um programa.

Existem três tipos diferentes de comentários em Java e você pode usar cada um deles como quiser. O primeiro modo de incluir um comentário em um programa é precedendo-o com os dois caracteres de barra normal (//). Tudo o que estiver das barras até o final da linha é considerado comentário e ignorado pelo compilador Java, como no comando a seguir:

```
float angulo = 1.57; // define o ângulo em graus
```

Caso seja necessário usar um comentário que ocupe mais de uma linha, pode-se iniciar o comentário com /* e finalizá-lo com */. Tudo o que estiver entre estes delimitadores será considerado comentário, como no seguinte.

```
/* Este programa foi escrito tarde da noite, sob a
influência de medicamento anti-histamínico com
validade expirada. Não damos garantias expressas
ou implícitas de que ele funcione */
```

O último tipo é semelhante ao comentário de múltiplas linhas mas tem o propósito de documentar o programa. Geralmente o comentário de documentação é posicionado imediatamente antes do elemento a ser documentado e tem seu conteúdo extraído automaticamente pelo utilitário javadoc fornecido juntamente com o JDK. Esta ferramenta gera páginas em formato html contendo os comentário organizados da mesma forma que a documentação fornecida juntamente com o JDK.

Aproveitando tal característica do javadoc, usualmente são adicionados *tags* html aos comentários de documentação para melhorar a forma final da documentação produzida com a inclusão de imagens, tabelas, textos explicativos, *links* e outros recursos. Além das *tags* html vários tipos de informação administradas pelo javadoc podem ser adicionadas a estes comentários especiais através de marcadores pré-definidos iniciados com "@" que permitem a criação automática de ligações hipertexto entre a documentação e a formatação padronizada de outros elementos, tais como nome do autor, parâmetros, tipo de retorno, etc. Veja o exemplo a seguir:

```
/** Classe destinada ao armazenamento de arquivos
* ou diretórios.
* <p> Pode ser usada para armazenar árvores de
* diretórios.
* @author Carlos da Silva
* @see java.io.File
*/
public class FileData extends File {
 /** Construtor
 * @param filename nome do arquivo
 */
 public FileData(String filename) {  }
}
```

Exemplo 2. 11 – O uso do Javadoc

Ao salvarmos o exemplo acima em um arquivo chamado "fileData.Java" e digitarmos no diretório em que este arquivo se localiza o seguinte comando:

```
javadoc fileData.java
```

O javadoc irá gerar um arquivo html contendo toda a documentação referente ao arquivo. Embora não seja necessário, tradicionalmente, costuma-se iniciar cada linha intermediária de um comentário com *. Ao processar o arquivo texto, javadoc irá desprezar o * e os espaços em branco iniciais de cada linha comentada. No exemplo, a tag html <p> é utilizada para iniciar um novo parágrafo, formatando o texto do comentário. A única restrição é que não se use os tags de cabeçalho (por exemplo, <h1>) porque o javadoc cria seus próprios cabeçalho.

Existem três tipos de comentários de documentação, cada um correspondente aos elementos que eles precedem: classe, variável e método. O exemplo abaixo ilustra onde devem ser escritos:

```
/** Um comentario de classe */
public class docTeste {
  /** Um comentario de variavel */
  public int i;
  /** Um comentario de metodo */
  public void f() {}
}
```

Exemplo 2. 12 – Comentários de classe, variável e método

Existem vários tags próprios para o javadoc. O tag @see cria links para outras classes pertencentes a documentação. Os tags @author e @version permitem fornecer informações a respeito do autor e da versão da implementação. Os tags @param, @return e @exception permitem incluir informações sobre os parâmetros, o valor de retorno e as exceções que podem ser disparadas pelo método. Por fim, o tag @deprecated permite indicar que o método está desatualizado, já havendo outro método correspondente que deve ser utilizado.

2.6 – Operadores

Operadores são símbolos especiais que recebem um ou mais argumentos e produzem um novo valor. Os operadores de adição (+), subtração (-), multiplicação (*), divisão (/) e atribuição (=) funcionam de forma muito semelhante na maioria das linguagens de programação.

A maioria dos operadores trabalha apenas com os tipos primitivos. As exceções são '=', '==', e '!=', que podem ser aplicados a todos os objetos. A classe String possui ainda suporte aos operadores '+' e '+='.

2.6.1– Operadores Aritméticos

Como na maioria das linguagens de programação, o Java possui vários operadores aritméticos:

Operador	Significado	Exemplo
+	Adição	a + b
-	Subtração	a - b
*	Multiplicação	a * b
/	Divisão	a / b
%	Resto da divisão inteira	a % b
-	Sinal negativo (- unário)	-a
+	Sinal positivo (+ unário)	+a
++	Incremento unário	++a ou a++
--	Decremento unário	--a ou a--

Tabela 2. 4 - Operadores Aritméticos

Estes operadores aritméticos podem ser combinados para formar expressões onde deve ser observada a precedência (ordem convencional) de avaliação dos operadores. Parêntesis podem ser utilizados para determinar uma forma específica de avaliação de uma expressão. A seguir um exemplo de aplicação que declara algumas variáveis, atribui valores iniciais e efetua algumas operações imprimindo os resultados obtidos.

```
// Aritmetica.java
 class Aritmetica {
  public static void main ( Strings args[] )
 {
   short x = 6;
   int y = 4;
   float a = 12.5;
   float b = 7;

   System.out.println ( "x é " + x + ", y é " + y );
   System.out.println ( "x + y = " + (x + y) );
   System.out.println ( "x - y = " + (x - y) );
   System.out.println ( "x / y = " + (x / y) );
   System.out.println ( "x % y = " + ( x % y ) );
   System.out.println ( "a é " + a + ", b é " + b );
   System.out.println ( " a / b = " + ( a / b ) );
 }
}
```

Exemplo 2. 13 – Aplicação Aritmética

Se você executar este aplicativo Java, ele produzirá a seguinte saída :

```
x é 6, y é 4
x + y = 10
x - y = 2
x / y = 1
x % y = 2
a é 12.5, b é 7
a / b = 1.78571
```

2.6.2– Operadores Relacionais

Além dos operadores aritméticos o Java possui operadores relacionais, isto é, operadores que permitem comparar valores literais, variáveis ou o resultado de expressões retornando um resultado do tipo lógico, isto é, um resultado falso ou verdadeiro. Os operadores relacionais disponíveis são:

Operador	Significado	Exemplo
==	Igual	a == b
!=	Diferente	a != b
>	Maior que	a > b
>=	Maior ou igual a	a >= b
<	Menor que	a < b

<=	Menor ou igual a	a <= b

Tabela 2. 5 – Operadores Relacionais

Note que o operador igualdade é definido como sendo um duplo sinal de igual (==) que não deve ser confundido com o operador de atribuição, um sinal simples de igual (=). Um erro extremamente comum em C/C++ é o seguinte:

```
while (x = y)
  { ... }
```

O programador está tentando realizar o teste de igualdade (==), mas na verdade está sendo executada uma atribuição. Em C/C++, o resultado da atribuição sempre retornará true se y for diferente de zero, e teremos provavelmente um loop infinito.

Em Java, diferentemente de C/C++, não é possível a realização de uma atribuição como parte de uma expressão condicional. Tal prática implicará em erro de compilação. A decisão dos projetistas da linguagem Java de abolir esta prática, presente nas linguagens C e C++, talvez tenha sido o fato da mesma constituir um erro comum mesmo para programadores mais experientes.

O operador de desigualdade é semelhante ao existente na linguagem C, ou seja, é representado por "!=". Os demais são idênticos a grande maioria das linguagens de programação em uso.

É importante ressaltar também que os operadores relacionais duplos, isto é, aqueles definidos através de dois caracteres, não podem conter espaços em branco.

A seguir um outro exemplo simples de aplicação envolvendo os operadores relacionais. Como para os exemplos anteriores, sugere-se que esta aplicação seja testada como forma de se observar seu comportamento e os resultados obtidos.

```
// Relacional.java
public class Relacional {
 static public void main (String args[])
 {
  int a = 15;
  int b = 12;

  System.out.println("a = " + a);
  System.out.println("b = " + b);

  System.out.println("a == b -> " + (a == b));
  System.out.println("a != b -> " + (a != b));
  System.out.println("a < b -> " + (a < b));
  System.out.println("a > b -> " + (a > b));
  System.out.println("a <= b -> " + (a <= b));
  System.out.println("a >= b -> " + (a >= b));
 }
}
```

Exemplo 2. 14 – Aplicação Relacional

2.6.3– Operadores Lógicos

As expressões que resultam em valores booleanos, como operações de comparação, podem ser combinadas para formar expressões mais complexas. Isso é realizado por operadores lógicos. Esses operadores são usados para representar os conectivos lógicos AND, OR, XOR e NOT lógico.

Para a representação do AND são usados os operadores lógicos &. E &&. Quando duas expressões são vinculadas pelos operadores & ou &&, a expressão combinada retornará o valor true apenas se as duas expressões booleanas forem verdadeiras.

A diferença entre & e && reside no trabalho realizado pela linguagem Java na avaliação da expressão combinada. Se for usado o operador &, as expressões dos dois lados desse símbolo são avaliadas, independente do que sejam. Se for usado o operador && e a expressão do lado esquerdo desse símbolo for falsa, a expressão do lado direito não será avaliada.

No caso dos conectivos OR, os operadores lógicos | e || são usados. Quando duas expressões são vinculadas pelos operadores | ou ||, o valor retornado por essa combinação será verdadeiro se pelo menos uma das duas expressões booleanas for verdadeira.

Observe o uso de | em vez de ||. Se for usado o operador |, as expressões dos dois lados desse símbolo são avaliadas. Caso o operador usado seja || e a expressão do lado esquerdo desse símbolo for verdadeira, o resultado retornado será verdadeiro e a segunda expressão não será avaliada.

O conectivo XOR (ou exclusivo) é representado pelo operador lógico ^. Esse operador resulta em um valor true somente se as duas expressões booleanas que ele combina possuírem valores opostos. Caso as duas expressões retornem true ou as duas retornem false, o resultado retornado será false.

O conectivo NOT usa o operador lógico !, seguido de uma única expressão. Ele reverte o valor de uma expressão booleana da mesma maneira que o sinal de subtração reverte o sinal positivo ou negativo em um número.

Por exemplo, se idade < 18 retornar um valor true, !(idade < 18) retornará um valor false.

```
// Logico.java
public class Logico {
 static public void main (String args[])
 {
  System.out.println("True & True = " + (true & true) );
  System.out.println("True && True = " + (true & true) );
  System.out.println("True & False = " + (true & false) );
    System.out.println("True  &&  False  =  "  +  (true  &
false) );
  System.out.println("True | True = " + (true | true) );
  System.out.println("True || True = " + (true || true) );
  System.out.println("True | False = " + (true & false) );
    System.out.println("True  ||  False  =  "  +  (true  &
false) );
  System.out.println("Not True = " + (!true) );
  System.out.println("Not False = " + (!false) );
 }
}
```

Exemplo 2. 15 – Aplicação Lógica

O comportamento dos operadores lógicos pode parecer difícil de ser entendido, quando encontrados pela primeira vez. Você terá muitas oportunidades de trabalhar e se familiarizar com eles nos capítulos subseqüentes.

2.6.4– Operador de Atribuição

Atribuição é a operação que permite definir o valor de uma variável através de uma constante ou através do resultado de uma expressão envolvendo operações diversas. A atribuição de uma variável é uma expressão, pois produz um valor. Devido a essa característica, você pode encadear instruções de atribuição da seguinte maneira:

```
x = y = z = 7;
```

Nessa instrução, todas as três variáveis acabam com o valor 7.

O lado direito de uma operação de atribuição é sempre calculado antes que a atribuição ocorra. Isso torna possível o uso de instruções de atribuição como no exemplo do código a seguir:

```
int x = 5;
x = x + 2;
```

Na expressão x = x + 2, o que primeiro acontece é o cálculo de x + 2. O resultado desse cálculo, no caso 7, é então atribuído a x.

O uso de uma expressão para mudar o valor de uma variável é uma tarefa extremamente comum em programação. Existem vários operadores utilizados estritamente nesses casos. A tabela 6 mostra esses operadores de atribuição e as expressões funcionalmente equivalentes.

Expressão	Significado
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`

Tabela 2. 6 – Operadores de atribuição especiais

2.6.5– Operadores de Incremento e Decremento

Outra tarefa comum é somar ou subtrair 1 de uma variável inteira. Existem operadores especiais para essas expressões, que são chamadas operações de incremento e decremento.

O operador de incremento é ++ e o de decremento é --. Esses operadores são colocados imediatamente antes ou depois de um nome de variável, como no exemplo a seguir:

```
int x = 7;
x++;
```

Nesse exemplo, a instrução x++ incrementa x de 7 para 8.

Os operadores de incremento e decremento podem ser colocados antes ou depois de um nome de variável e isso pode afetar o valor das expressões que os envolvem. Em uma expressão simples, como x++, usar operadores prefixados (antes da variável) ou pós-

fixados (depois da variável) não altera o resultado final. Contudo, quando as operações de incremento e decremento fazem parte de uma expressão maior, a escolha entre operadores prefixados ou pós-fixados é importante. Considere as duas expressões a seguir:

```
int x, y, z;
x = 42;
y = x++;
z = ++x;
```

Essas duas expressões produzem resultados distintos, por causa da diferença entre as operações prefixadas e pós-fixadas. Quando você usa operadores pós-fixados, como em y = x++, y recebe o valor de x antes que ele seja incrementado. Ao se usar operadores prefixados, como em z = ++x, x é incrementado em 1 antes da operação de atribuição. O resultado final deste exemplo é que y é igual a 42, z é igual a 44 e x é igual a 44.

2.6.6– O Operador Ternário

É um operador pouco comum que apesar de possuir três operandos, funciona exatamente como um operador pois produz um valor (diferentemente da estrutura de desvio de fluxo if-else que você verá no próximo capítulo). O oerador ternário de Java possui a seguinte forma:

```
<expressão_booleana> ? <valor_1> : <valor_2>
```

Caso a <expressão_booleana> for *true*, o <valor_1> é avaliado e seu resultado torna-se o resultado produzido pelo operador ternário. Se a <expressão booleana> for *false*, o <valor_2> é quem é avaliado, e o resultado da avaliação torna-se o valor produzido pelo operador.

É claro que você pode usar a estrutura if-else (descrita posteriormente) no lugar do operador ternário, mas este último produz um código mais enxuto. Entretanto, C (a linguagem de onde este operador é originário) orgulha-se de ser uma linguagem enxuta, mas essa caraterística por vezes compromete a legibilidade dos programas. Portanto você deve tomar uma certa cautela ao usar este operador, já que torna-se mais fácil escrevermos códigos ilegíveis.

O exemplo abaixo mostra o uso deste operador:

```
// ternario.java
public class ternario {
 static public void main (String args[])
 {
  int i = 7;
  int j = i < 10 ? (2*i) : (i);  // faz j = 2*i
  System.out.println("i: " + i); // imprime "i: 7"
  System.out.println("j: " + j); // imprime "j: 14"
 }
}
```

Exemplo 2. 16 – Aplicação Lógica

2.6.7– Precedência na Avaliação de Operadores

Numa expressão onde existam diversos operadores é necessário um critério para determinar qual destes operadores será primeiramente processado, tal como ocorre em expressões matemáticas comuns. A precedência é o critério que especifica a ordem de avaliação dos operadores de um expressão qualquer. Na Tabela 7 temos relacionados os níveis de precedência organizados do maior (Nível 1) para o menor (Nível 15). Alguns dos operadores colocados na tabela não estão descritos neste texto.

Nível	Operadores
1	.(seleção) [] ()
2	++ -- ~ *instanceof* *new* *done* - (unário)
3	* / %
4	+ -
5	<< >> >>>
6	< > <= >=
7	== !=
8	&
9	^
10	|
11	&&
12	||
13	?:
14	= += -= *= /=
15	,

Tabela 2. 7 - Precedência dos Operadores em Java

2.7 – Tipos Compostos (Objetos)

Freqüentemente, há a necessidade dos programadores modelarem conjuntos de dados que não são homogêneos. Por exemplo, informações a respeito de um estudante universitário poderiam incluir o nome, o número de matrícula, a idade, a média das notas e assim por diante.

Linguagens como C e Pascal usam o conceito de registros para permitir a criação de tipos compostos pelo programador. Um registro é um agregado de dados, possivelmente heterogêneos cujos elementos individuais são identificados por nomes.

O Java não possui estruturas do tipo registro, e seus tipos compostos (inclusive arrays e strings) são objetos. Imagine um objeto como uma variável especial que pode armazenar dados (atributos), Cada objeto pode armazenar internamente outros objetos, ou seja, você pode criar um objeto que possui como atributos outros objetos.

Cada objeto é uma instância de uma classe, aonde classes definem conjuntos de objetos com características e comportamento comuns. Analogamente, assim como variáveis primitivas possuem tipos, objetos possuem classes. O exemplo abaixo ilustra a definição da classe coordenadas e o uso de seus objetos:

```
public class coordenadas {
 public int x;
 public int y;
 public int z;

  static public void main (String args[])
  {
   coordenadas cord = new coordenada();
   cord.x = 10;
   cord.y = 15;
   cord.z = 18;
  }
}
```

Exemplo 2. 17 – Definição de Objetos

Cada linguagem de programação possui seus próprios meios de manipulação de dados. Em alguns casos, o programador deve estar constantemente atento para o tipo de manipulação que irá realizar. Em diversas linguagens de programação, são comuns, por parte dos programadores, dúvidas do tipo: irei manipular o objeto diretamente ou estarei lidando com algum tipo de representação indireta (um ponteiro em C ou C++) que deve ser tratada com uma sintaxe especial ?

Em Java, tudo isto é simplificado. Uma única sintaxe é usada em todos os casos e o no caso dos objetos, o identificador que você manipula é na realidade uma referência a um objeto.

Imagine a seguinte cena: uma televisão (o objeto) com o seu controle remoto (a referência). Enquanto você tiver posse desta referência, você possuirá, de certa forma, uma conexão com a televisão. E quando você realizar alguma ação como "mudar de canal" ou "aumentar o volume", o que você estará manipulando será a referência, que por sua vez modificará o objeto. Se você quiser se mover pela sala e ainda controlar a televisão, você levará o controle remoto (referência) com você, e não a televisão.

Por outro lado, o controle remoto pode existir no quarto, sem necessariamente existir uma televisão. O fato de você ter uma referência não assegura que necessariamente você terá sempre um objeto associado a ela.

Por exemplo, para manipular uma cadeira de caracteres, criamos uma referência à objetos da classe String:

```
String s;
```

Até aqui foi criada apenas a referência, portanto nenhum objeto está associado a ela. Qualquer tentativa de manipular a referência implicará em algum erro (em tempo de execução), já que s não está associada a nenhum objeto. Portanto, uma prática segura é sempre iniciar a referência com algum objeto, no momento de sua criação:

```
String s = "Olá Mundo!";
```

O exemplo acima usa uma característica especial da classe string: strings podem ser inicializadas por cadeias de caracteres delimitadas por aspas. Normalmente são usadas formas mais gerais na inicialização de objetos, mas este assunto será melhor tratado em um capítulo subseqüente.

2.7.1– Atribuição de Valores Primitivos e de Objetos

Um detalhe interessante da linguagem Java é que atribuição de valores primitivos é realizada por valor, ou seja, é realizada uma cópia do valor original, e esta cópia é que é atribuída á variável. Por exemplo imagine a operação A = B, na qual A e B são tipos primitivos da linguagem. Note que será realizada uma cópia do conteúdo da variável B para a variável A. Com isso, as duas variáveis passam a existir no programa de forma independente. Portanto qualquer alteração posterior no conteúdo da variável B não afetará a variável A. Isso é o que um programador esperaria que ocorresse na maioria das situações.

O mesmo não ocorre durante a atribuição de objetos. Em Java, quando são realizadas atribuições envolvendo objetos, o que acaba sendo manipulado na verdade são referências a estes objetos.
Imagine duas variáveis C e D referenciando dois objetos distintos (figura 2.1-a). Ao executarmos o comando C = D, passamos a ter as duas variáveis C e D referenciando um mesmo objeto antes somente referenciado por D (figura 2.1-b).

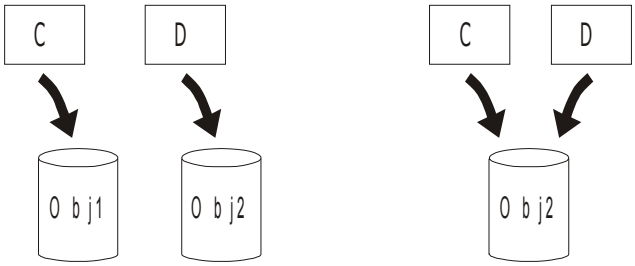

Figura 2.1-a **Figura 2.1-b**

Figura 2. 1 - Atribuição de Objetos

O exemplo seguinte irá demonstrar esta situação:

```
//Atribuicao.java
class Numero {
 int i;
}
public class Atribuicao {
 public static void main(String[] args) {
  Numero n1 = new Numero();
  Numero n2 = new Numero();
  n1.i = 9;
  n2.i = 47;
  System.out.println("1: n1.i: " + n1.i +", n2.i: " +
                     n2.i);
  n1 = n2;
  System.out.println("2: n1.i: " + n1.i +", n2.i: " +
                     n2.i);
  n1.i = 27;
  System.out.println("3: n1.i: " + n1.i +", n2.i: " +
                     n2.i);
 }
}
```

Exemplo 2. 18 – Atribuição de Objetos por Referência

No exemplo acima, n1 e n2 são iniciados com duas instâncias da classe Numero. Inicialmente são atribuídos diferentes valores para o atributo 'i' de n1 e n2. Depois, n2 é atribuído a n1, e as duas variáveis passam a referenciar um mesmo objeto. Como o ocorrido em várias linguagens de programação, você esperaria que n1 e n2 fossem independentes durante todo o tempo. Mas como n2 foi atribuído a n1, e a atribuição nestes casos é feita por referência, isso não ocorreu e esta é a saída que este programa gera:

```
1: n1.i: 9, n2.i: 47
2: n1.i: 47, n2.i: 47
3: n1.i: 27, n2.i: 27
```

2.7.2– Strings

Strings são seqüências de caracteres. Java não possui um tipo primitivo para a representação de strings, mas possui a classe String definida em sua biblioteca padrão. Isto significa que em Java Strings são objetos e cada cadeia de caracteres delimitada por aspas duplas é uma instância da classe String:

```
String e = ""; // uma string vazia
String saudacao = "Olá!";
```

Java, como a maioria das linguagens de programação, permite o uso do operador + para concatenar duas strings.

```
String cumprimento = "Bom dia";
String pessoa = "Renata";
String msg = cumprimento + " " + pessoa + "!";
```

O código acima faz a variável msg ter valor igual a "Bom dia Renata!".

Quando uma string é concatenada a um valor que não é uma string, este valor é então convertido em uma string. Como veremos em um capítulo futuro, qualquer objeto pode ser convertido em uma string. O exemplo abaixo mostra a concatenação de uma string com uma expressão numérica:

```
System.out.println("15 + 31 = " + (15+31) );
```

O código acima possui alguns passos que são realizados implicitamente: primeiramente é resolvida a expressão numérica (15+31), que têm como retorno o inteiro 46. Depois disso, o resultado da expressão (no caso o inteiro 46) é convertido de inteiro para string e só após isso é concatenado à string `"15 + 31 = "`. Portanto o resultado do comando acima é a saída:

```
15 + 31 = 46
```

Strings são constantes, ou seja, não é possível modificarmos o valor de uma string depois de sua criação. Portanto, a string `"Olá!"` conterá sempre os caracteres `'O','l','á','!'`. E estes valores não podem ser modificados. O que pode ser modificado é o conteúdo da variável `saudacao`, que poderá referenciar a uma outra string.

2.8 Exercícios

1. Edite e compile cada um dos exemplos deste capítulo. Corrija os exemplos onde ocorre erro de compilação. Agora execute todos os exemplos. Em seguida, altere, compile e os execute novamente.
2. Faça um programa que imprima um valor de cada tipo primitivo.
3. Inclua comentários no arquivo do segundo exercício e gere a documentação utilizando javadoc.
4. Crie um arquivo para a classe abaixo. Compile e execute-o.

```
class shortExplo
{
  public static void main ( String[] args )
  {
    short valor = 32;
    System.out.println(valor + " é um short!");
  }
}
```

   Edite o programa alterando 32 para 356. Compile e execute. Repita o procedimento agora com 35000. Altere novamente o programa modificando o tipo de valor para int. Compile e execute. Repita o procedimento alterando o tipo de valor para double. Substitua também 35000 por um valor fora dos limites de double (voce pode usar a notação científica para isso). O que acontece?
5. Crie um arquivo para a classe abaixo. Compile e execute-o.

```
class charExplo
{
  public static void main ( String[] args )
  {
    char ch = 'A' ;
    System.out.println("Um caracter: " +  ch );
  }
}
```

   Mude 'A' no programa por 'AA' e compile-o. Agore mude 'AA' por "A".
   Compile novamente.
6. Crie um arquivo para a classe abaixo e o compile. Explique o que ocorreu. Faça a correção devida e execute-o.

```
class byteExplo
{
  public static void main ( String[] args )
  {
    byte a, b, c;
    b = 10;
    c = 15;
    a = b + c;
    System.out.println("Um byte: " +  a );
  }
}
```

7. Considere o seguinte exemplo:

```
class incrExplo
{
  public static void main ( String[] args )
  {
    int a, b, c;
    b = c = 10;
    a = b++ + b++;
    System.out.println("valor de a = " +  a );
    System.out.println("valor de b = " +  b );
    a = ++c + ++c;
    System.out.println("valor de a = " +  a );
    System.out.println("valor de c = " +  c );
    a = 10;
    b = 5;
    boolean x = a > b || b++ > 3;
    System.out.println("valor de x = " +  x );
    System.out.println("valor de b = " +  b );
    a = 1;
    b = 5;
    x = a > b || b++ > 3;
    System.out.println("valor de x = " +  x );
    System.out.println("valor de b = " +  b );
    c = 5;
    b = 3;
    a = (b = 3 + c) + b;
    System.out.println("valor de a = " +  a );
    System.out.println("valor de b = " +  b );
  }
}
```

Tente dizer quais valores serão impressos. Depois, crie um arquivo, compile e execute. Os resultados coincidiram? Se não, o que foi que aconteceu?