Programação em Java - Fundamentos

2 - Sintaxe

Citeforma

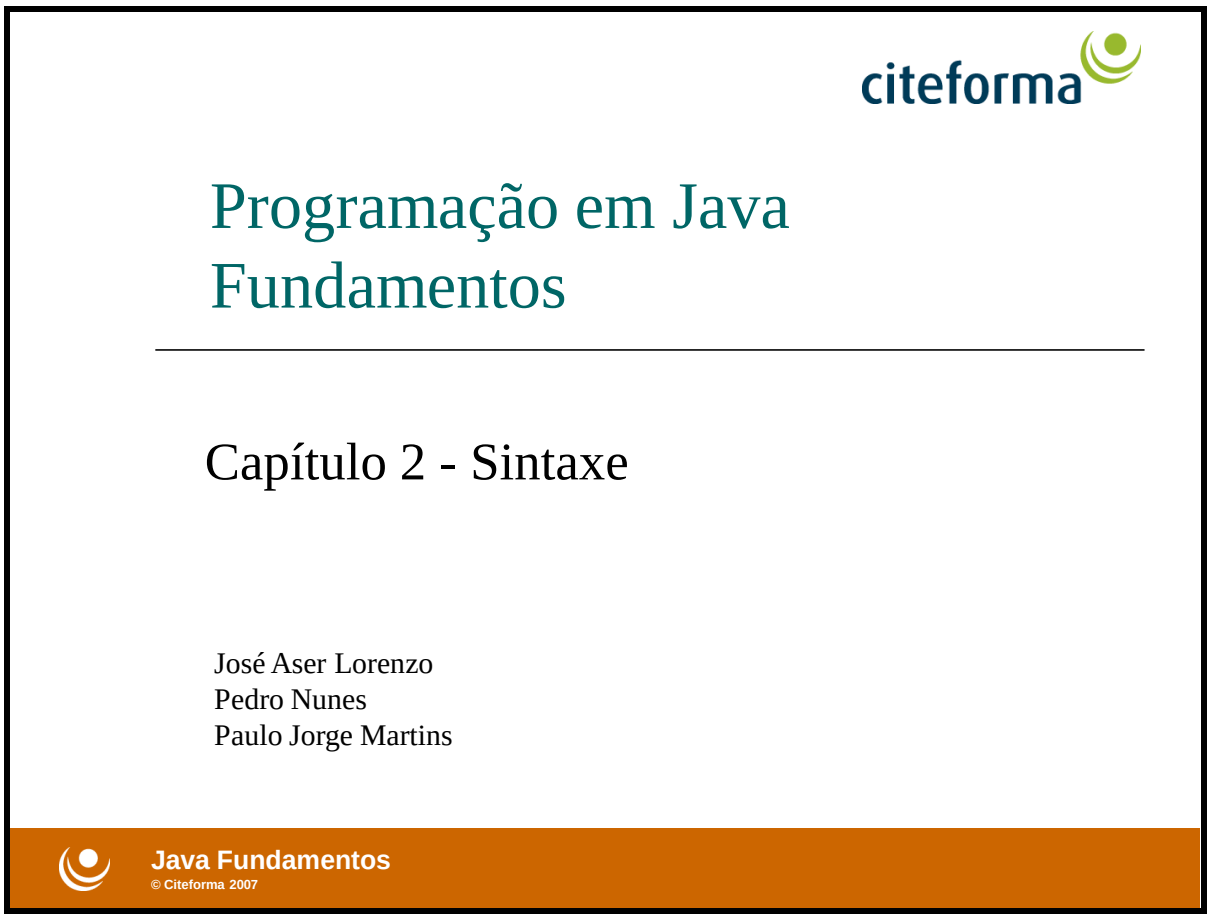
Jose Aser Lorenzo, Pedro Nunes, Paulo Jorge Martins

jose.l.aser@sapo.pt

Março de 2012

Sumário

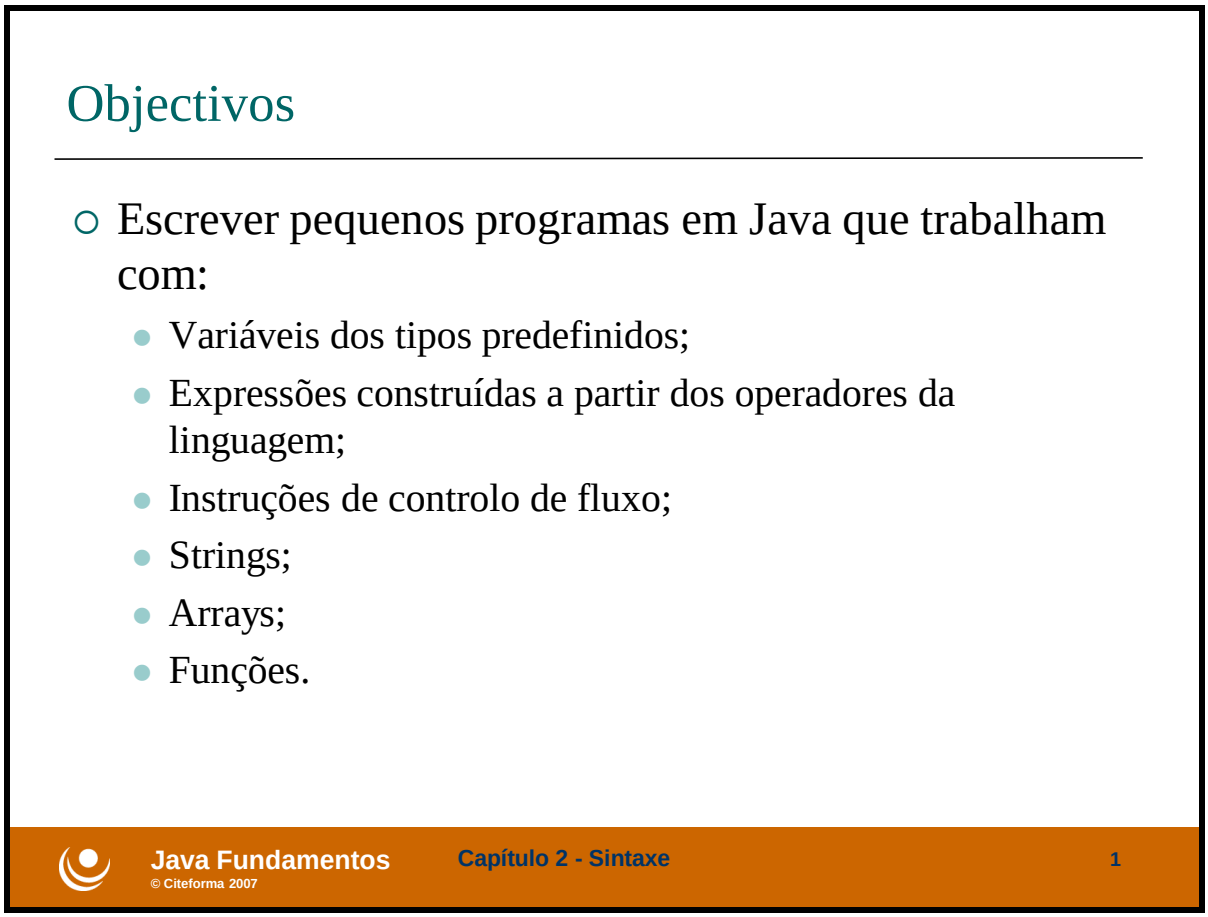
Capítulo 2 - Sintaxe



A sintaxe de uma linguagem natural, como o português, consiste no conjunto de regras gramaticais que definem como as palavras devem ser combinadas para a construção de frases. De forma semelhante, a sintaxe de uma linguagem de programação define um conjunto de regras que, embora menos extensas, são mais rigorosas, e servem para a criação das linhas de código que constituem o programa.

Numa linguagem natural cada elemento da linguagem é classificado como letra, palavra, verbo, sujeito, adjetivo, frase, parágrafo, etc. Numa linguagem de programação também se classificam os seus elementos: programa, funções, instruções, operadores, operando, expressões, etc.

Objetivos

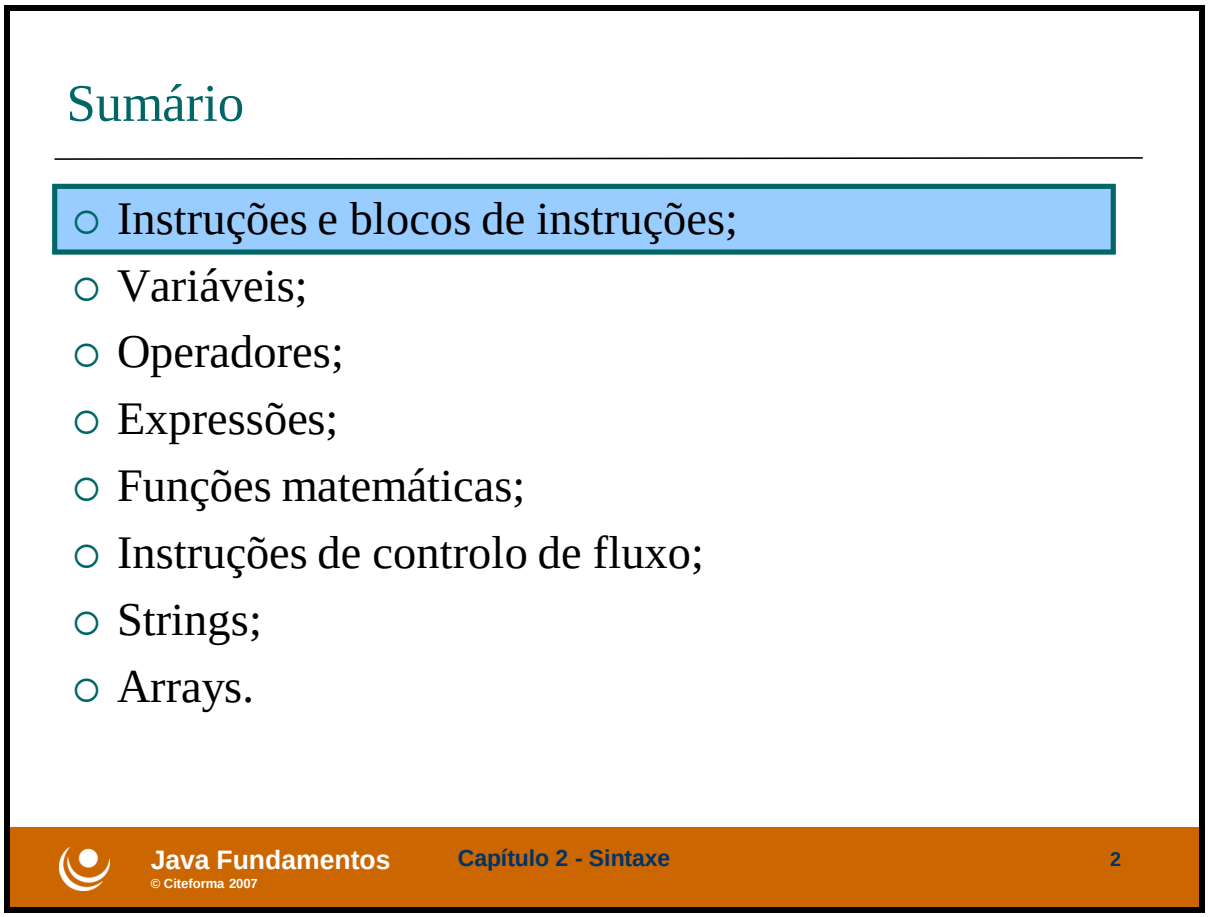


No fim deste capítulo estará apto a escrever pequenos programas em Java, utilizando os tipos de variáveis predefinidos, instruções de controlo de fluxo e funções com parâmetros. Os programadores de outras linguagens estarão aptos a usar Java como uma linguagem baseada em procedimentos.

Ao longo deste capítulo aprenderemos as regras fundamentais definidas pela sintaxe da linguagem Java, assim como algumas recomendações ("*coding standards*") que devemos seguir para tornar o código mais legível.

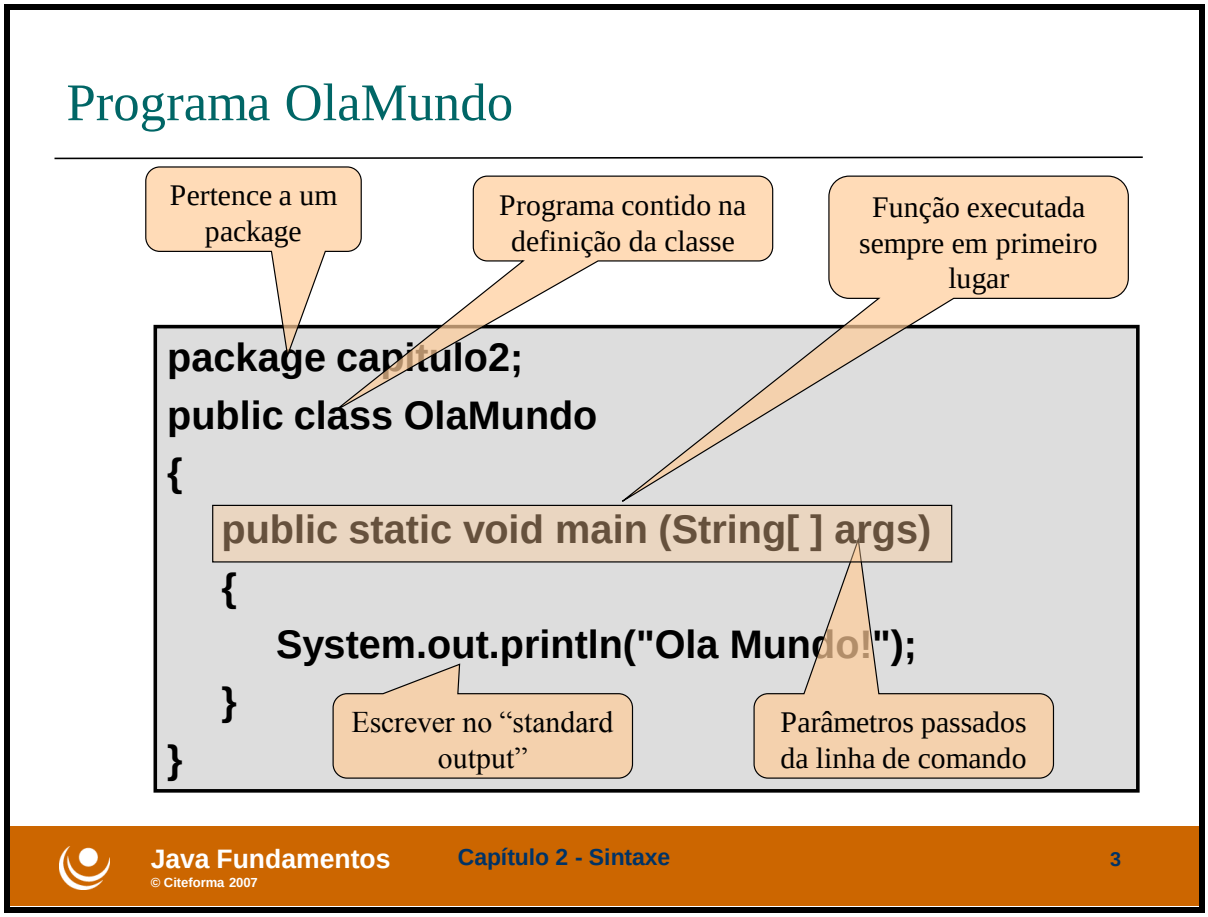
Analisaremos em detalhe cada um dos elementos da linguagem acima enunciados e, no fim de cada tópico, realizaremos exercícios de consolidação.

Instruções e blocos de instruções



Na linguagem Java, as instruções têm de ser terminadas com ponto e vírgula. Podem-se agrupar várias linhas de instruções, entre chavetas, definindo assim um bloco de instruções.

Programa OlaMundo



Vamos começar por escrever um programa que escreve a frase "Olá Mundo". Os passos abaixo indicados devem ser seguidos em todos os exemplos deste manual, variando apenas o nome da classe.

- Abrir o NetBeans;
- Criar um projeto com o nome **JavaFundamentos** (se não foi criado no capítulo anterior):
  - File → New Project → Java → Java Aplication;
  - Nome do projecto: JavaFundamentos;
  - Escolher a diretoria onde o projeto ficará armazenado;
- Criar uma nova classe com as características abaixo:
  - Escolher o projeto JavaFundamentos. Com o botão de contexto do rato:
    - New → Java Class
    - Name: OlaMundo
    - Package: capitulo2 (incluir a classe dentro do package capitulo2 - mais tarde veremos o que são packages);
  - Alterar a linha 12 para a apresentada abaixo:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package capitulo2;

/**
 *
 * @author Aser
 */
public class OlaMundo {
    public static void main (String args[]) {
        System.out.println("Ola Mundo");
    }
}
```

Seguem alguns comentários sobre este programa:-

Um programa Java está contido na definição de uma classe, neste caso **OlaMundo**. O ficheiro que contem a classe deve ter o mesmo nome que a classe, respeitando letras maiúsculas e minúsculas, com a extensão **java**. Todos os exemplos deste capítulo seguem este critério.

Este programa define a função main(), que recebe como parâmetro um array de Strings e é criada com os qualificadores **public** e **static**, não retornando valores (**void**). Esta função será executada em primeiro lugar, quando mandamos executar a classe, e chamará os outros elementos do programa. No final deste capítulo veremos como criar um array de Strings e nos próximos capítulos abordaremos o significado dos qualificadores.

A instrução System.out.println() envia para o "*standard output*" o texto que recebe com parâmetro. **out** é uma variável que pertence à classe System e representa o canal de output. O método println() serve para escrever texto nesse canal de saída.

Os blocos de código são limitados por { e } como na linguagem C.

Os comentários são colocados entre /* e */. Se começar por /** o comentário será lido pelo utilitário javadoc, que produz documentação automaticamente. Todo o texto que venha depois de // e até ao fim da linha é ignorado pelo compilador, pelo que esta sequência também é usado para comentários.

Em Java é aconselhável que todas as classes sejam armazenadas dentro de um package, que neste caso tem o nome capitulo2. Um package é uma biblioteca de classes e é implementado por uma diretoria;

Para compilar esta classe na linha de comando deve introduzir o comando abaixo na diretoria que contem a classe:-

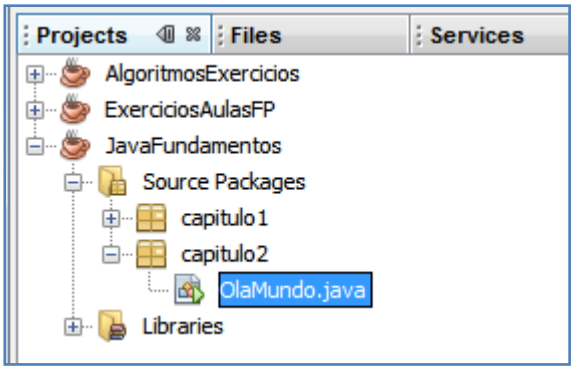
```
capitulo2> javac  OlaMundo.java
```

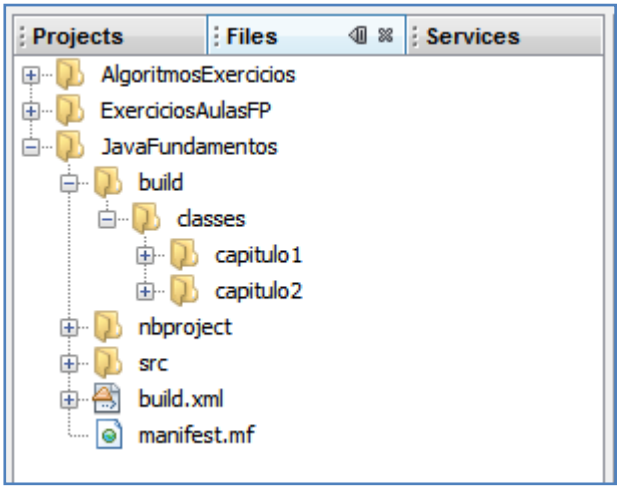
Após a compilação é gerado um ficheiro de *bytecode* com o mesmo nome da classe, neste caso **OlaMundo.class**. Para o executar na linha de comando deve introduzir o comando abaixo na diretoria imediatamente anterior à que contem a classe, neste caso capitulo2:-

```
> java  capitulo2.OlaMundo
Ola Mundo !
>
```

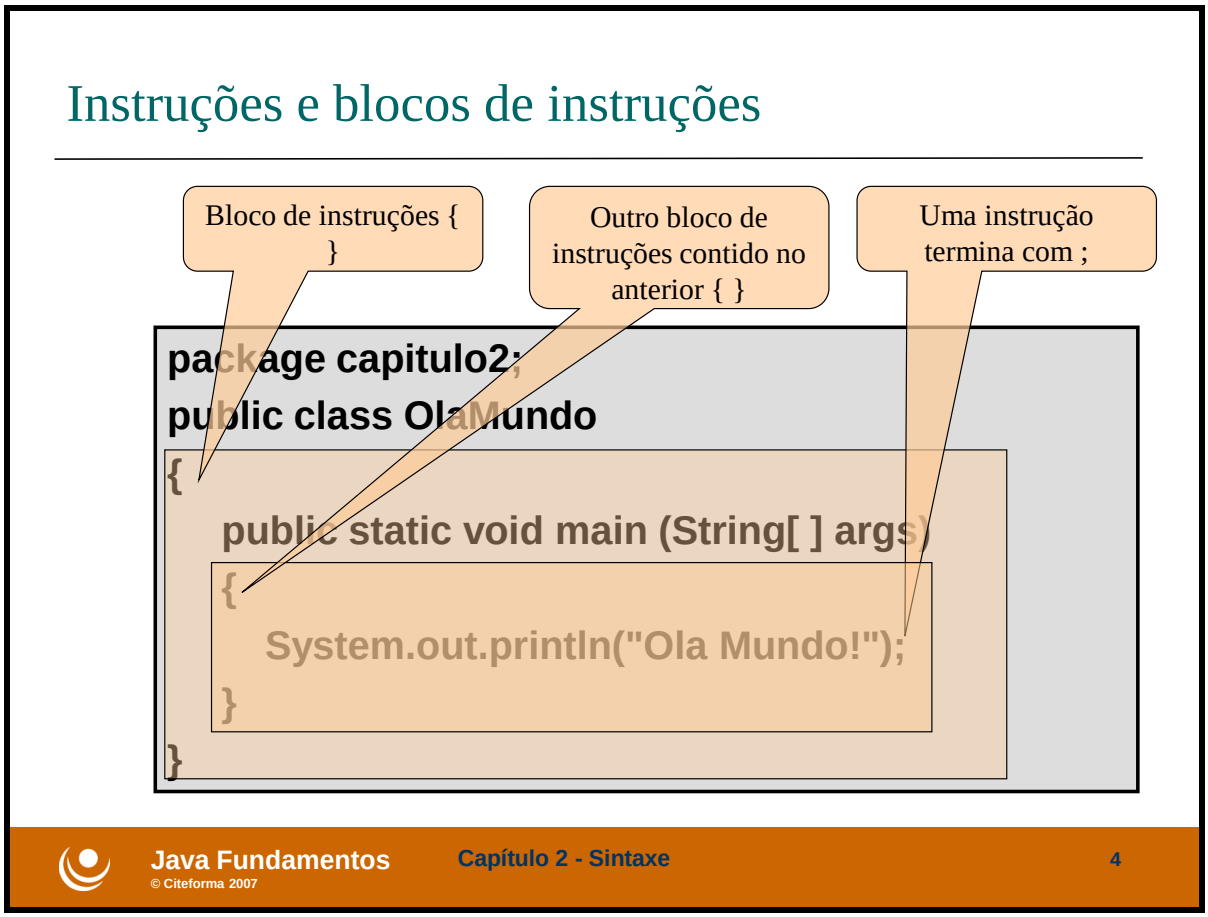
Para compilar e executar esta classe no **NetBeans** faça SHIFT-F6. Se preferir pode usar o menu de contexto do rato e escolher a opção **Run File**. Não faça Run Project pois neste momento não temos o projeto configurado para isso.

Ao contrário da compilação manual, ao compilar no NetBeans o ficheiro com extensão class foi guardado na diretoria JavaFundamentos → Build → Classes → capitulo2. Se nada for dito em contrário o NetBeans guarda os source files na diretoria src e os class na diretoria build. Este comportamento pode ser alterado nas propriedades do projeto. Veja a estrutura de diretorias do projeto recorrendo ao separador file.




Projects
Files
Services
AlgoritmosExercicios
ExerciciosAulasFP
JavaFundamentos
build
classes
capitulo1
capitulo2
nbproject
src
build.xml
manifest.mf

Instruções e blocos de instruções



As instruções em Java terminam sempre com ponto e vírgula (**;**).

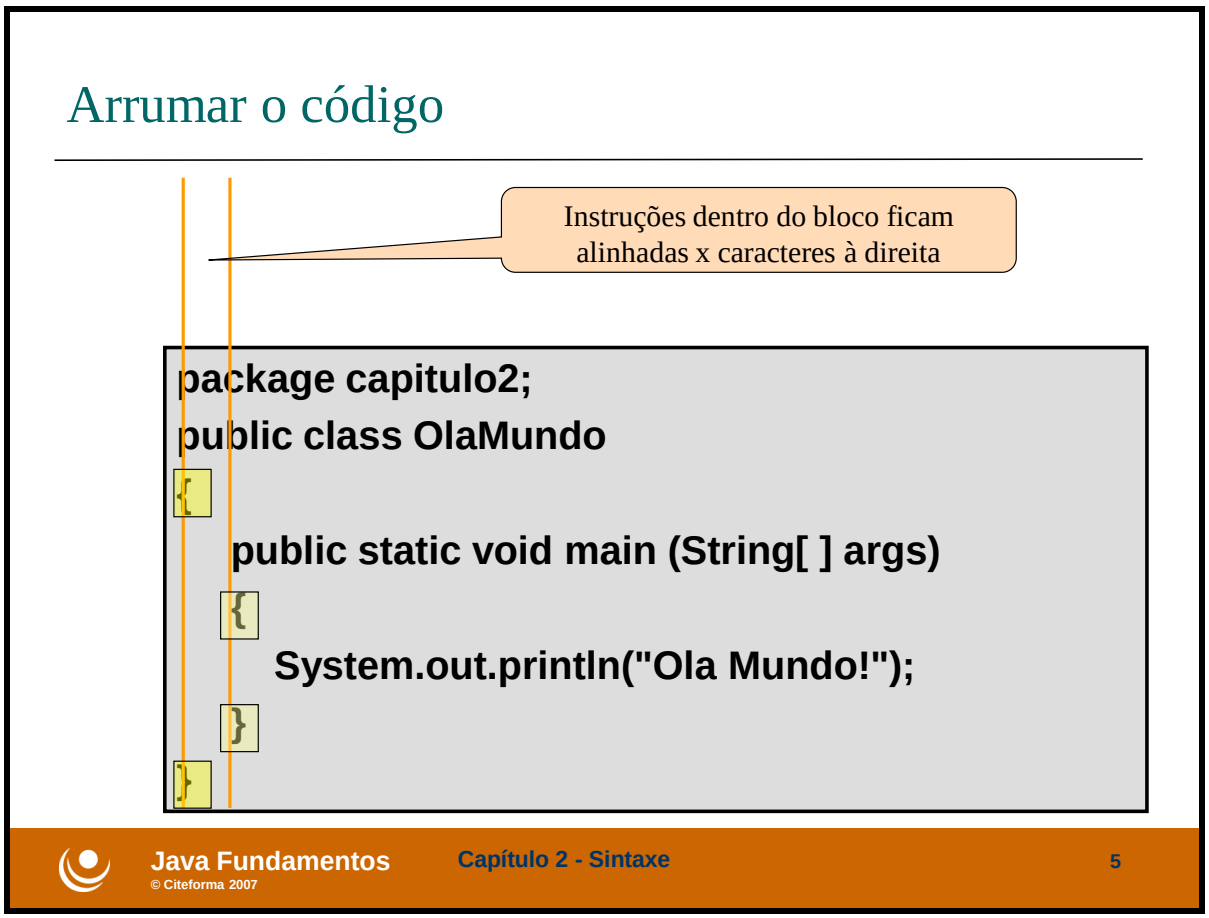
Os blocos de instruções são delimitados por chavetas, '**{**' e '**}**'.

```
{
   int k = 0;
   System.out.println("k tem valor zero");
   k=1;
   System.out.println("k tem valor um");
}
```

Os blocos de instruções podem conter várias instruções e/ou outros blocos de instruções.

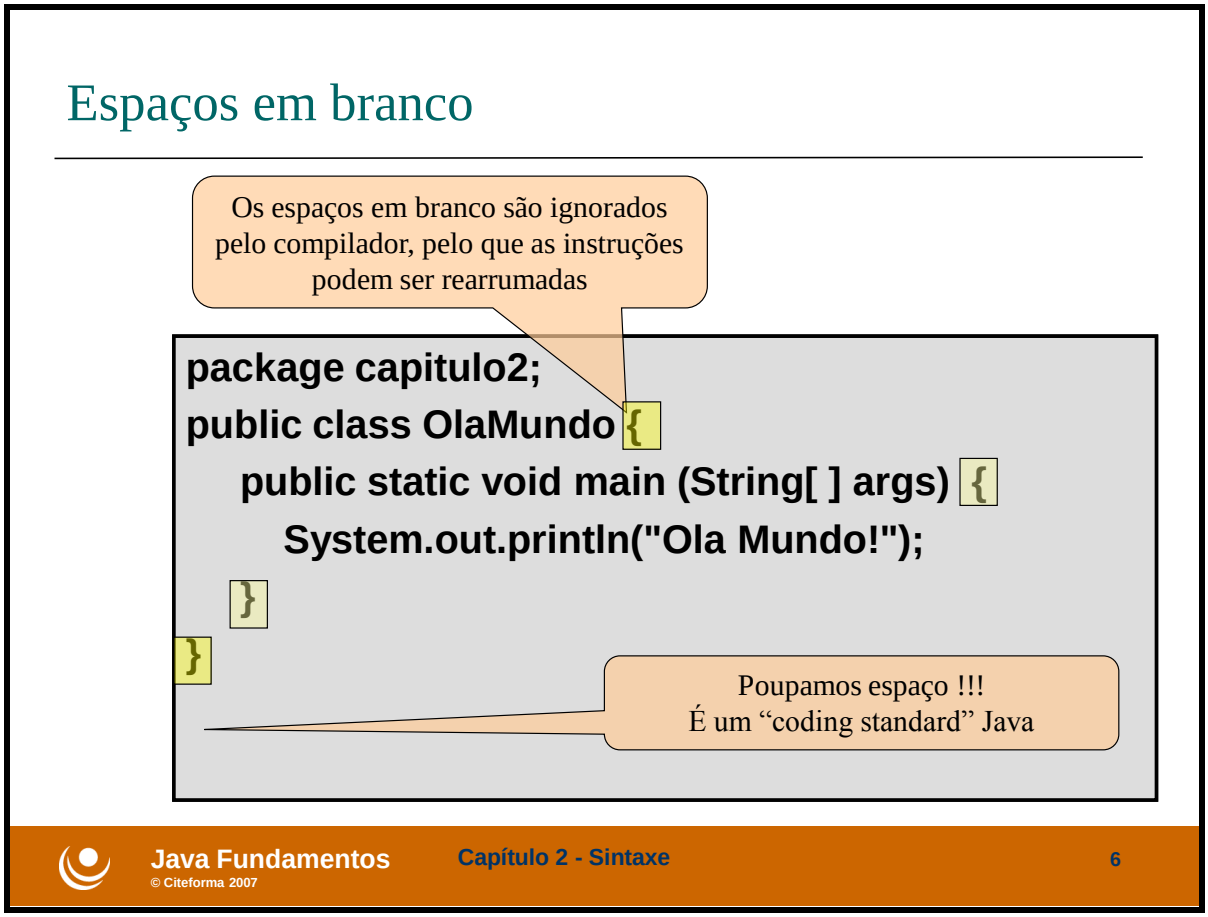
Em alguns casos as chavetas são obrigatórias, mesmo que só contenham uma instrução, por exemplo, para delimitar o início e o fim do corpo de uma classe e o início e o fim do corpo das funções.

Arrumar o código



Existem várias convenções ("*coding standards*") que recomendam formas de arrumar o código e todas elas concordam que, no caso de um sub-bloco, as instruções nele contidas são encostadas à direita 3 ou 4 espaços em branco (ou tabulador), como mostra o exemplo anterior.

Espaços em branco

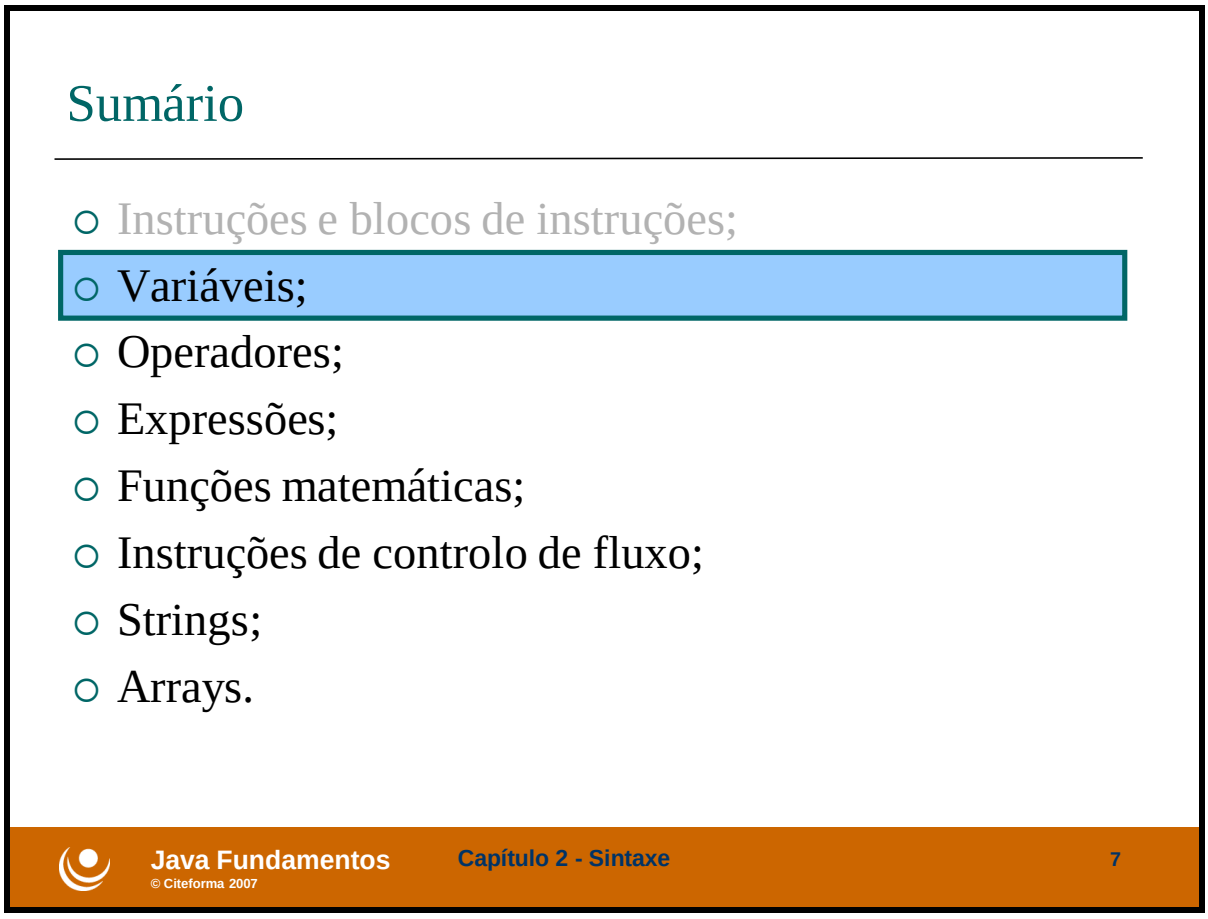


O compilador de Java ignora os espaços em branco, pelo que as instruções acima poderiam ser arrumadas de várias formas.

A Sun definiu um "*coding standard*" que recomenda que se coloque a chaveta de início de bloco na mesma linha que a instrução que o inicia, o que permite poupar espaço. O exemplo abaixo aplica esta formatação ao programa OlaMundo:

```
package capitulo2;
public class OlaMundo {
   public static void main(String[] args) {
      System.out.println("Ola Mundo!");
   }
}
```

Variáveis

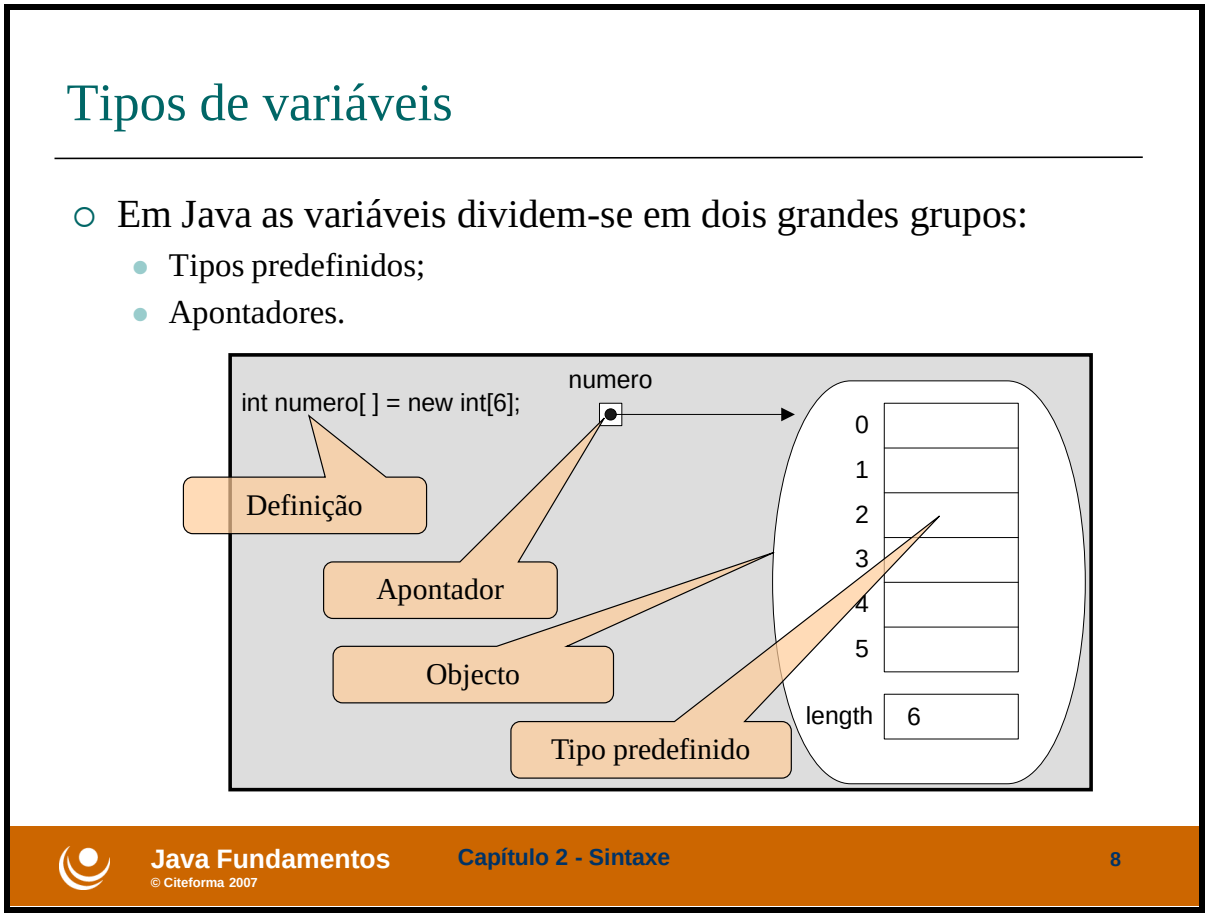


As variáveis são utilizadas nos programas de computador para armazenar dados.

Antes de serem utilizadas têm que ser declaradas, o que consiste na definição de 4 componentes:-

- O tipo da variável;
- O nome da variável;
- O seu ciclo de vida;
- O seu valor inicial.

Tipos de variáveis



As variáveis em Java são obrigadas a ter um tipo de dados (o Java é *Strong Typed*). O tipo define o domínio dos dados que a variável pode receber e as operações que com ela podem ser efetuadas.

O Java suporta duas grandes categorias de tipos de variáveis: os **predefinidos** e os que contêm **apontadores**.

A declaração abaixo define **contador** como uma variável do tipo inteiro. Ela pode armazenar números inteiros, positivos ou negativos, e pode ser utilizada em expressões com os operadores aritméticos, como por exemplo a soma, subtração, divisão ou multiplicação.

```
int contador;
```

Tipos de variáveis predefinidos (“built in”)

Tipos de variáveis predefinidos (“*built in*”)

Tipo	Tamanho e Formato	Descrição
byte	8 bit em complemento de dois	Número inteiro de –128 a 127
short	16 bit em complemento de dois	Número inteiro de –32768 a 32767
int	32 bit em complemento de dois	Número inteiro de -2^{31} a $(2^{31}-1)$
long	64 bit em complemento de dois	Número inteiro de -2^{63} a $(2^{63}-1)$
float	32 bit no formato IEEE 754	Número em virgula flutuante de precisão simples entre $1.5xe^{-45}$ e $3.4xe^{38}$
double	64 bit no formato IEEE 754	Número em virgula flutuante de precisão dupla entre $5.0xe^{-324}$ e $1.7xe^{308}$
char	16 bit no formato UNICODE	Um caracter simples
boolean	true ou false	Valor lógico

Java Fundamentos Capítulo 2 - Sintaxe 9

O tipo **int** é o mais utilizado, mas nele não cabe o número de habitantes do planeta terra. Nesta situação teríamos que usar o tipo **long**. Os tipos **byte** e **short** servem para situações específicas, como seja o tratamento de ficheiros de baixo nível ou a criação de arrays de números onde o consumo de memória se torna um fator crítico.

Um número pode ser representado em hexadecimal usando o prefixo 0x (ex 0xCAFE). Também pode ser representado em octal usando o prefixo 0 (ex: 010).

O tipo **double** é o mais usado para números com parte decimal. Quando o consumo de memória é um fator crítico podemos optar por **float** que ocupa metade do espaço, mas tendo em conta que há perda de precisão.

```
float a = 123456.78f;
float b = 876543.21f;
System.out.println(a+b); // imprime 1000000.0 em vez de 999999.99
```

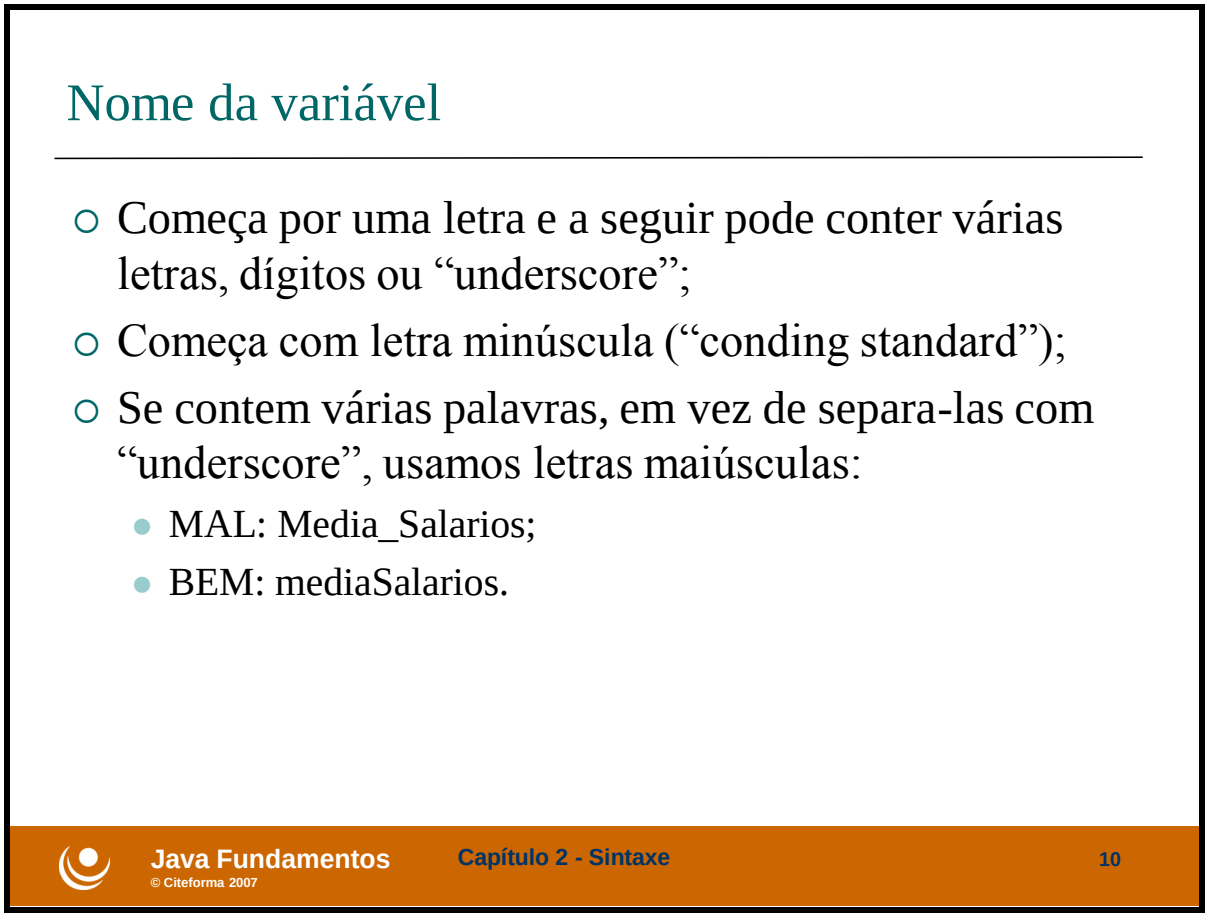
Se a precisão de **long** e **double** não for suficiente o programador pode optar por utilizar as classes **BigInteger** e **BigDecimal**.

UNICODE é um código de codificação de caracteres criado pelo **UNICODE Consortium** (http://unicode.org) para suportar as várias línguas usadas na Terra. Permite a codificação de 65.535 caracteres, embora só estejam definidos os primeiros 38.885. Inclui os caracteres dos alfabetos Japonês, Grego, Russo e Hebreu. Este código inclui os conhecidos ASCII e ISO8859-1 (Latin-1).

Apontadores

Os tipos que contêm apontadores, em vez de armazenarem valores de um dos tipos pré-definidos, guardam um endereço de memória que aponta para um objeto. Por exemplo, um array é referenciado pelo endereço da primeira posição. O programador não pode ver os valores desses endereços, pois é a máquina virtual Java que faz a sua gestão. A forma como se faz o endereçamento de memória varia com as plataformas e por isso a linguagem Java esconde este nível de complexidade. Mais adiante veremos como criar objetos e voltaremos aos apontadores.

Nome da variável



Uma variável é identificada pelo respetivo nome, que deverá ser usado quando se pretende alterar ou usar o conteúdo da variável. Por convenção os nomes das variáveis começam por letras minúsculas e os das classes por maiúsculas.

Os nomes dos identificadores devem:

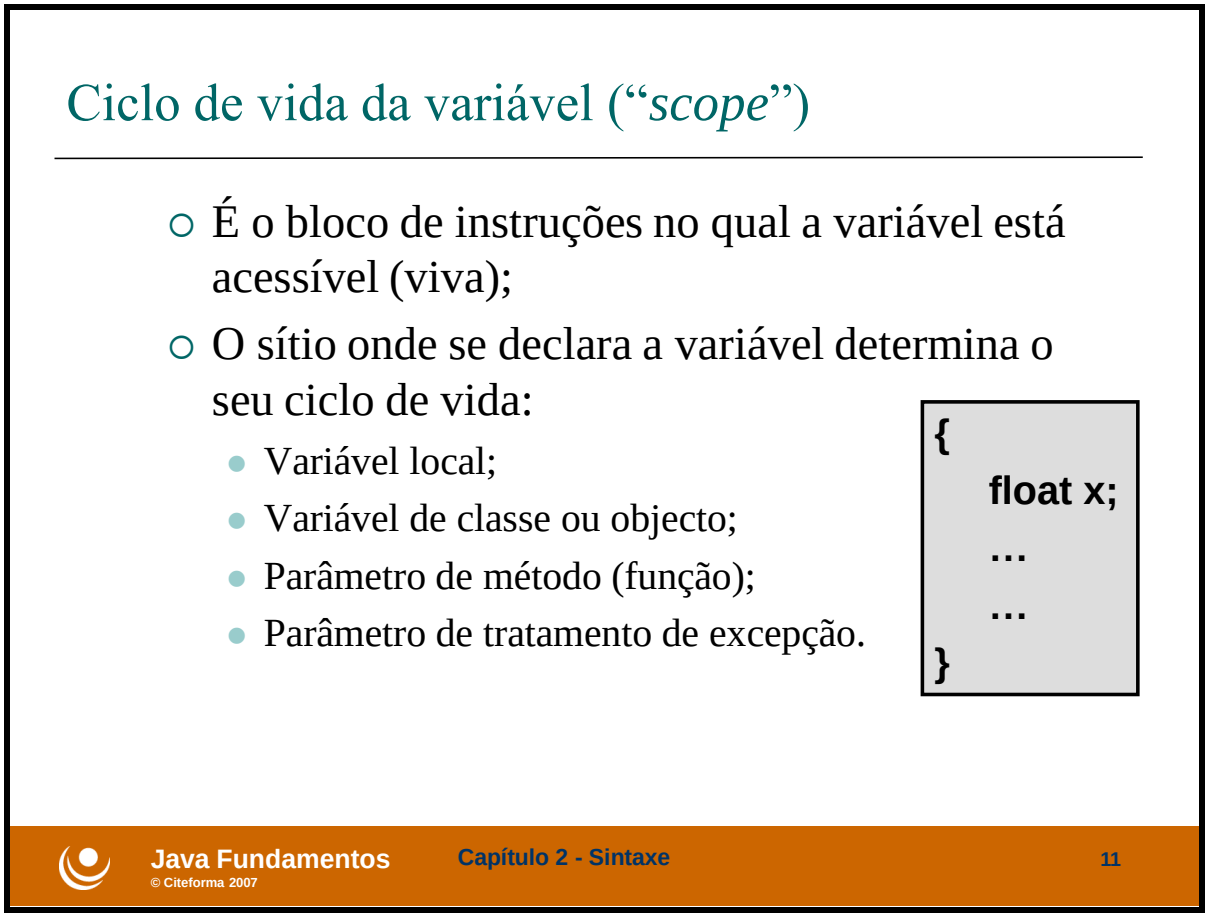
- Acompanhar a regra dos identificadores em Java definida abaixo;
- Não ser igual a uma palavra reservada do Java;
- Não ser igual ao nome de outra variável definida no mesmo bloco de instruções ("*scope*");

Os identificadores podem começar por uma letra, um dollar ($) ou um underscore (_) e a seguir podem conter várias letras, dígitos, dollars ou underscores.

Caso uma variável contenha várias palavras, não é normal em Java separá-las com underscore, mas sim usar maiúsculas, como no próximo exemplo:

```
Mal: Media_Salarios
Bem: mediaSalarios
```

O ciclo de vida da variável



O ciclo de vida de uma variável é definido pelo bloco de instruções no qual ela está acessível. Em inglês este conceito é designado pelo termo "*scope*". Quando o programa entra num bloco de instruções a máquina virtual Java cria as variáveis nele definidas. Estas estas serão destruídas quando o programa sai desse bloco.

O sítio onde a variável é declarada determina o seu ciclo de vida. Assim, a variável pode ser classificada numa das seguintes categorias: **local**, de **classe ou objeto**, **parâmetro de um método**, **parâmetro de tratamento de exceção**.

Uma variável local é declarada nos blocos de instruções dos métodos da classe. O seu ciclo de vida coincide com o bloco de instruções que contém a sua declaração (limitado por { }).

Uma variável de classe está definida no bloco de instruções que constitui a classe e pode ser usada enquanto a classe estiver em memória. Uma variável de objeto é declarada no bloco de instruções que constitui a classe e pode ser usada no contexto de um objeto dessa classe. No próximo capítulo voltaremos a este tema.

As variáveis utilizadas como parâmetros em métodos servem para passar valores para o método. O seu ciclo de vida é o próprio método. As variáveis usadas como parâmetros para o tratamento de exceções têm o mesmo comportamento que os parâmetros dos métodos, pois são válidas no bloco de instruções definido para tratar a exceção.

Valor inicial da variável

Valor inicial da variável

- As variáveis locais, de classe ou de objecto podem receber um valor inicial:

```
{
    int contador = 0;
    int i,j,k,m,n;
    i = j = k = m = n = 0;
}
```

Java Fundamentos
© Citeforma 2007
Capítulo 2 - Sintaxe
12

Das categorias anteriores, aquelas em que faz sentido falar de valor inicial são as variáveis de membro e as locais. O valor inicial deve pertencer ao domínio da variável e pode ser definido do seguinte modo:

```
int contador = 0;
int i,j,k,m,n;
i=j=k=m=n=1;
```

No exemplo anterior a variável contador recebe o valor zero.

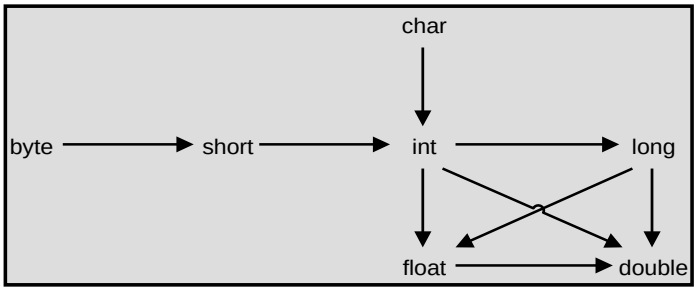
Nas instruções seguintes as variáveis i, j, k, m e n recebem o valor 1.

Por omissão, as variáveis de domínios numéricos recebem o valor zero.

Conversão entre tipos de variáveis

Conversão entre tipos de variáveis

- Conversão automática:



- Conversão manual (cast):

```
float  f = 2.14f;
int  i;
i =  (int)  f;
```

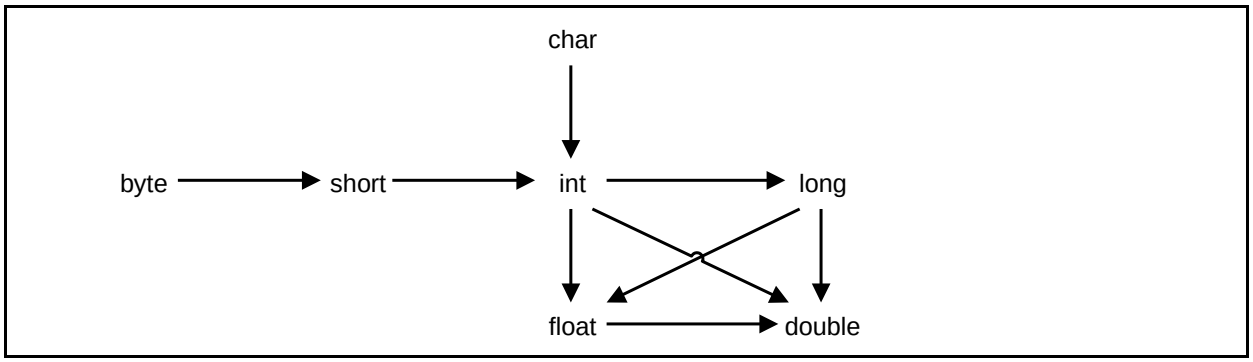
Java Fundamentos
© Citeforma 2007
Capítulo 2 - Sintaxe
13

Em matemática o conjunto dos números inteiros está contido no conjunto dos números reais. Em Java os tipos de dados não têm intersecção e portanto uma variável do tipo int não está contida no tipo double. Isto ocorre porque a representação interna (em memória) de um int é completamente diferente da de um float. Enquanto um int segue uma conversão de decimal para binário e portanto não há erro de arredondamento, num float ou num double é seguida a norma IEE754, que causa erro de arredondamento. Os conteúdos podem ser convertidos de um tipo para outro, existindo conversões automáticas e manuais.

Conversão automática

As conversões de um tipo noutro mais abrangente dão-se automaticamente. Por exemplo, o conteúdo de uma variável int pode ser atribuído a um float, pois é convertido automaticamente de um formato no outro. O mesmo ocorre de um float para um double.

O diagrama abaixo mostra as conversões automáticas:-



Conversão manual: o operador cast

A conversão de um dado pertencente a um tipo abrangente num tipo menos abrangente (mais específico) não se faz de forma automática, pois pode provocar perda de informação.

A linguagem Java não permite a conversão automática (por exemplo) de float para int, pois poderá haver perda de informação ao retirar a parte decimal do float.

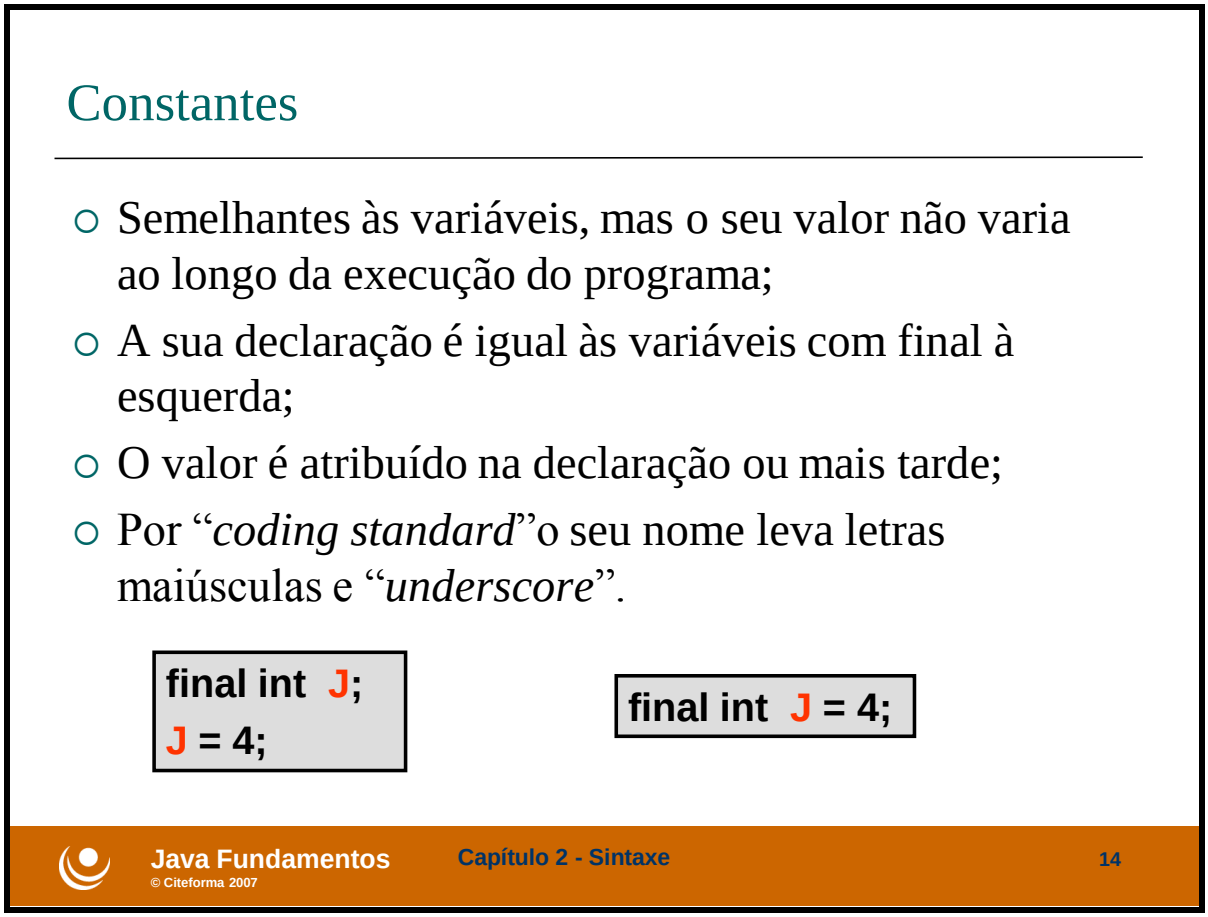
```
float x=1.23f;
int   i=x;        // dá erro, alertando para perda de precisão na conversão.
```

Se o programador efetivamente quiser fazer esta conversão, consciente da possível perda de informação, pode recorrer ao operador **cast**. Este consiste na indicação, entre parêntesis, do tipo de dados para o qual queremos converter o valor.

No exemplo abaixo estamos a fazer um cast de x para inteiro, indicando ao compilador que estamos cientes de que vamos perder a parte decimal de x.

```
float x=1.23f;
int   i=(int)x;   // Só estamos interessados na parte inteira do valor.
```

Constantes



As constantes são semelhantes às variáveis no sentido em que têm um tipo, armazenam um valor, são identificadas por um nome e têm um ciclo de vida.

Enquanto as variáveis podem armazenar vários valores ao longo do seu ciclo de vida, as constantes recebem um valor que se mantem inalterado durante o seu ciclo de vida.

A declaração de uma constante é semelhante à de uma variável, mas precedida pela palavra reservada **final**.

Por convenção o seu nome é colocado em maiúsculas, como no exemplo seguinte:

```
package capitulo2;
public class Constantes01 {
   public static void main(String args[]) {
      int i=3;        // variavel
      final int J; //constante
      J=4;
      J=i;
      J=5;
      System.out.println("i="+i);
      System.out.println("J="+J);
   }
}
```

Quando o nome da constante contiver várias palavras, deveremos separá-las com *underscore*, por exemplo HORAS_DO_DIA.

A execução deste programa produz o erro de compilação listado abaixo, referente à linha 6, pois tenta atribuir um novo valor a uma constante:

```
Constantes01.java:6: Can't assign a value to a final variable: J
              J=i;
              ^
1 error
```

Exercício sobre variáveis

Exercício sobre variáveis

- Exercício 1: desenvolver uma classe que define variáveis de vários tipos, atribui-lhes valores e escreve o seu conteúdo no "*standard output*";
- Exercício 2: desenvolver uma classe que pratica conversões automáticas entre tipos de variáveis;
- Exercício 3: desenvolver uma classe que usa o operador (cast);

Java Fundamentos Capítulo 2 - Sintaxe 15

Exercício 1:

Escrever uma classe que define variáveis de diferentes tipos e escreve os seus valores:

```
package capitulo2;

public class VariaveisEx1 {
   public static void main(String args[]) {
      int a, b, h, o;
      String s;
      a = 2;
      b = 2;
      h = 0x000D; // hexadecimal 13
      o = 011;    // octal 9
      System.out.println("h="+h+"   o="+o);
      // em binário 13...
      System.out.println("h (em binário)=" + Integer.toBinaryString(h));

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

      float f=2.14f;   // f indica que é float
      double d=3e10;   // notacao cientifica
```

```
        char c='a';       // ' limita char
        System.out.println("float="+f);
        System.out.println("double="+d);
        System.out.println("char="+c);
    }
}
```

O resultado da execução desta classe será:-

```
h=13   o=9
h (em binário)=1101
Total=4
float=2.14
double=3.0E10
char=a
```

São declaradas quatro variáveis inteiras e uma String. As variáveis **a** e **b** são inicializadas com um número em notação decimal. A variável **h** é inicializada com um número em notação hexadecimal ($D_{16}=13_{10}$) e a variável **o** recebe um número em octal ($11_8=9_{10}$).

É de notar também a utilização da classe Integer e do método toBinaryString() para a representação de **h** em binário.

Este exemplo usa a variável **out,** que pertencente à classe **System**, que por sua vez está predefinida no conjunto de classes standard da linguagem Java. O método **println**() escreve no canal "*standard output*" e é invocado no contexto da variável **out**, que é uma variável de classe, não sendo por isso necessário criar um objetos da classe **System**.

O operador + é usado com dois objetivos: somar números e concatenar Strings.

O operador = é utilizado para atribuição de valores.

Repare que os números com parte decimal são considerados do tipo **double** se nada for dito em contrário. Se quisermos que sejam do tipo **float** devem ser terminados com a letra **f**. No exemplo anterior, se colocássemos **2.14** em vez de **2.14f** obteríamos um erro de compilação.

As Strings são delimitadas por “” (aspas) enquanto os caracteres são delimitados por ‘ ’ (plicas).

Exercício 2:

Classe que mostra conversões automáticas entre diferentes tipos de variáveis.

```
package capitulo2;

public class VariaveisEx2 {
   public static void main(String args[]) {
      byte b=3;
      short s;
      int i;
      long l;
      float f=3.0f;
      double d;
      char c='a';
      s=b;  // short <-- byte
      System.out.println("short=" + s);
      i=s;  // int   <-- short
      System.out.println("int=" + i);
      i=c;  // int   <-- chart
      System.out.println("int=" + i);
      l=i;  // long  <-- int
      System.out.println("long=" + l);
      f=i;  // float <-- int
      System.out.println("float=" + f);
      f=l;  // float <-- long
      System.out.println("float=" + f);
      d=i;  // double<-- int
      System.out.println("double=" + d);
      d=l;  // double<-- long
      System.out.println("double=" + d);
      d=f;  // double<-- float
      System.out.println("double=" + d);

      // Exemplos sem perda de precisão...
      // float + int = float
      float x=f+i;
      System.out.println("f="+x);
      // float + int = int
      int y = (int)f+i;
      System.out.println("i="+y);

      // Exemplos com perda de precisão... --> Atenção!
      i = 16777217;
      f = i;
      System.out.println(f);  // imprime 1.6777216E7 em vez de 1.6777217E7
    }
}
```

Todas as conversões efetuadas no programa acima, são de um tipo mais específico para um tipo mais abrangente e são por isso automáticas. Nas conversões automáticas não há, regra geral, perdas de precisão.

No entanto algo parece não estar bem, pois o espaço ocupado em memória pelo tipo int e pelo tipo float é o mesmo (32 bits), e o tipo float consegue armazenar valores muito maiores que o tipo int, para além de ainda ter que guardar a parte decimal.

Isto acontece porque a partir de um dado valor (positivo e negativo) o float armazena esses inteiros com menos precisão, como podemos ver no exemplo abaixo.

```
   int i = 16777217;
   float f = i;
   System.out.println(f);  // imprime 1.6777216E7 em vez de 1.6777217E7
```

Apesar da conversão ser automática, também há uma perda de precisão na conversão de inteiro para float, como haverá na conversão de long para double.

Isto não é uma particularidade da linguagem Java, pois em qualquer linguagem de programação, o programador tem de ter algum cuidado ao lidar com conversões de vírgula flutuante (mesmo que automáticas) e arredondamentos.

Exercício 3:

Classe que usa o operador cast.

```
package capitulo2;

public class VariaveisEx3 {
   public static void main(String args[]) {
      float f;
      int i;
      f=2.14f;
      i=f;
      System.out.println("f="+f);
      System.out.println("i="+i);
   }
}
```

A compilação deste programa produz um erro na linha 7:-

```
VariaveisEx3.java:7: possible loss of precision
found   : float
required: int
        i=f;
1 error
```

Podemos evitar o erro de compilação, explicitando a conversão, com um **cast** para int.

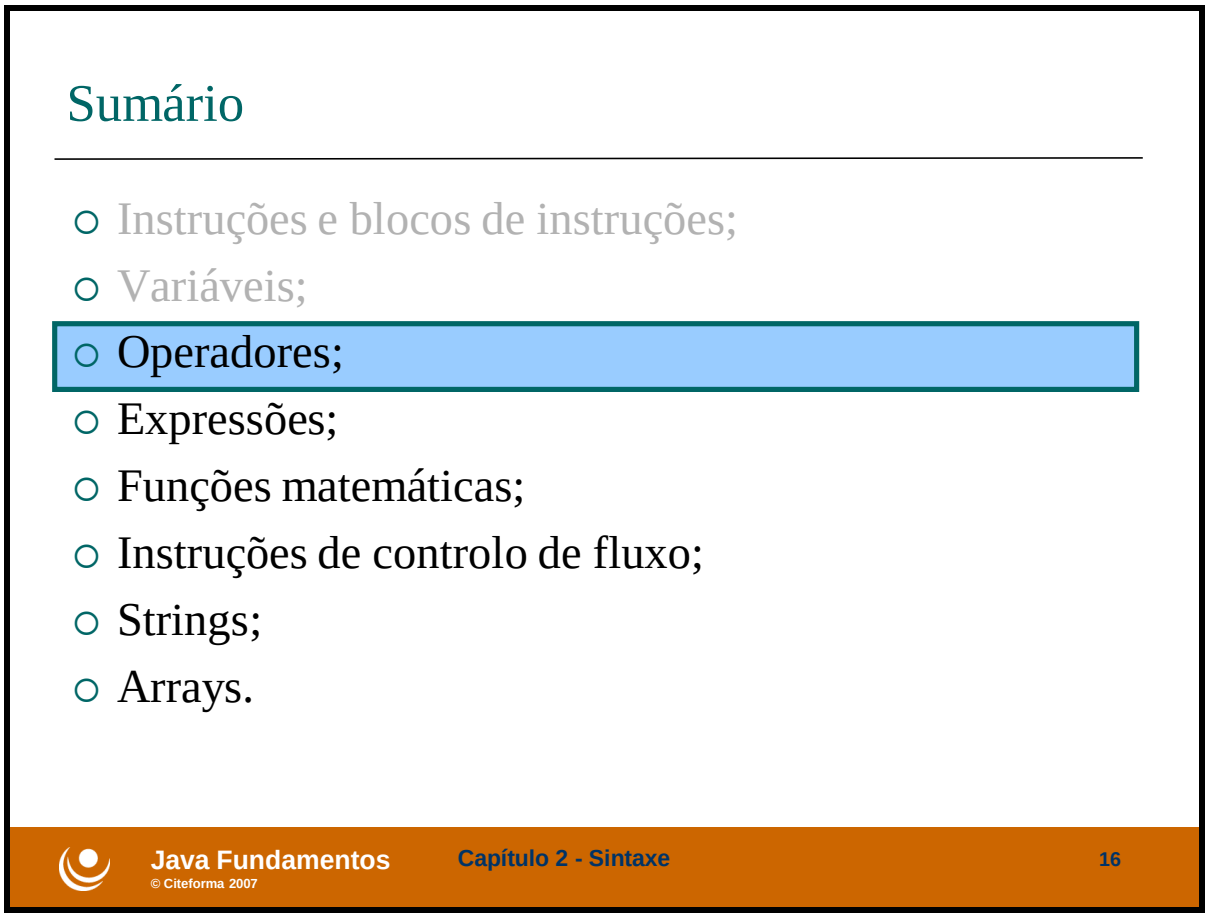
```
package capitulo2;
public class VariaveisEx3 {
    public static void main(String args[]) {
        float f;
        int i;
        f=2.14f;
        i= (int) f;
        System.out.println("f="+f);
        System.out.println("i="+i);
    }
}
```

Este programa produz o seguinte resultado:-

```
f=2.14
i=2
```

Repare que a conversão truncou a parte decimal do número.

Operadores



O valor retornado por uma operação depende do tipo dos operandos envolvidos e isto permite dividir os operadores da linguagem Java nos seguintes grupos:

- Aritméticos;
- Relacionais e lógicos;
- Bit a bit;
- De atribuição;

Operadores aritméticos

Operadores aritméticos

- Binários

Operador	Sintaxe	Descrição
+	op1 + op2	Calcula a soma de op1 com op2
-	op1 - op2	Calcula a diferença entre op1 e op2
*	op1 * op2	Calcula o produto de op1 com op2
/	op1 / op2	Calcula a divisão de op1 por op2
%	op1 % op2	Calcula o resto da divisão de op1 por op2

- Unários

Operador	Sintaxe	Descrição
++	++op	Incrementa op uma unidade; avalia o valor de op;
++	op++	Avalia o valor de op; incrementa op uma unidade;
--	--op	Decrementa op uma unidade; avalia o valor de op
--	op--	Avalia o valor de op; decrementa op uma unidade;

Java Fundamentos Capítulo 2 - Sintaxe 17

Os operadores efetuam operações sobre um ou dois operandos. Os operadores que necessitam de um operando são designados **unários**, enquanto os que requerem dois operandos são **binários**.

Na linguagem Java o tipo de dados resultante de uma operação aritmética depende do tipo dos operandos. O tipo resultante é sempre o mais abrangente dos tipos envolvidos na operação. Assim a soma, subtração, multiplicação ou divisão de um tipo int com um tipo float resulta no tipo mais abrangente, neste caso o float.

Quando queremos manter o tipo original, abdicando da conversão para o tipo mais abrangente, temos que recorrer ao operador cast.

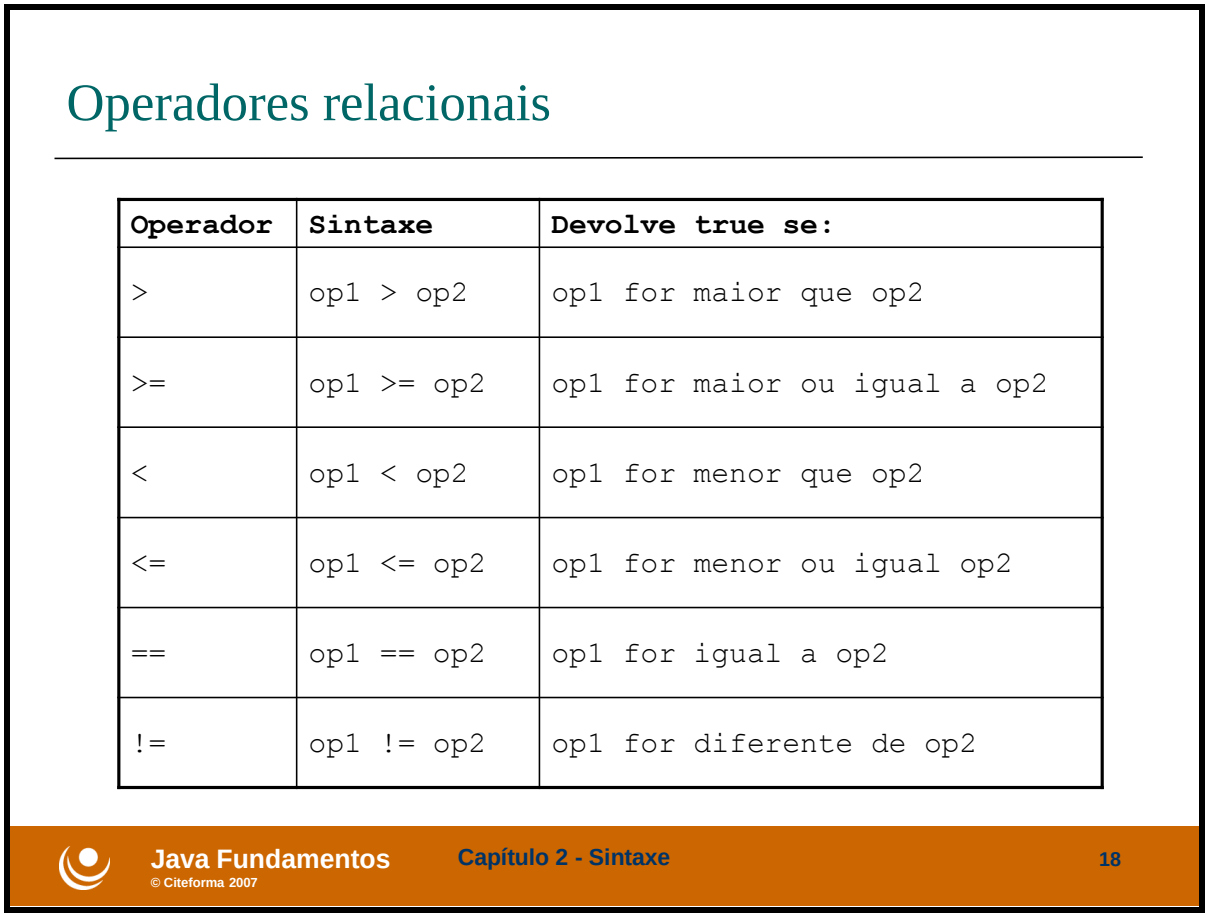
Operadores aritméticos unários

O símbolo usado pelo operador aritmético subtração (–) é também usado como operador unário para inverter o sinal do de um número.

O Java disponibiliza ainda os operadores **aritméticos compactos** ++ e - -, que permitem o incremento ou o decremento do valor da variável. Estes operadores unários podem ser utilizados em notação **pré fixada** ou **pós fixada**, o que significa que o operador pode aparecer antes ou depois do operando, respetivamente:

```
Pré fixada: ++op
Pós fixada: op++
```

Operadores relacionais



Os operadores relacionais comparam dois valores e determinam a relação entre eles, devolvendo **true** ou **false**. Por exemplo o operador **!=** devolve **true** se os dois operandos forem diferentes.

Operadores lógicos

Operadores lógicos

Operador	Sintaxe	Devolve true se:
&&	op1 && op2	op1 e op2 são ambos true (AND)
||	op1 || op2	ambos ou um deles é true (OR)
!	!op1	op1 é false (NOT)

Para efetuar operações lógicas o Java dispõe dos operadores **&&**, **||** e **!**. Além destes é possível efetuar operações lógicas com os operadores bit a bit **&** e **|**. Estes últimos não otimizam a avaliação de expressões lógicas em tempo de execução, pelo que só se recomenda a sua utilização em operações bit a bit. A otimização de expressões lógicas permite:

- Num AND se o primeiro operando é **false** o resultado é **false**, independentemente do valor do segundo operando. Por esse motivo nesta situação o segundo operando não é avaliado;
- Num OR se o primeiro operando é **true**, o resultado é **true**, independentemente do valor do segundo operando que nesta situação não chega a ser avaliado.

Operadores bit a bit

Operadores bit a bit

Operador	Sintaxe	Descrição:
>>	op1 >> op2	Os bits de op1 são deslocados à direita op2 unidades
<<	op1 << op2	Os bits de op1 são deslocados à esquerda op2 unidades
>>>	op1 >>> op2	Os bits de op1 são deslocados à direita op2 unidades não considerando o bit de sinal como bit significativo (não há <<< !)
&	op1 & op2	AND bit a bit
|	op1 | op2	OR bit a bit
^	op1 ^ op2	XOR bit a bit
~	~op1	Inverter o valor dos bits

Java Fundamentos Capítulo 2 - Sintaxe 20

Os operadores bit a bit permitem efetuar operações sobre os bits que representam os números armazenados em variáveis inteiras do tipo byte, short, char, int ou long. O tipo char é também um tipo inteiro, com a particularidade de não ter sinal.

Os operadores *shift* (<<, >>, >>>) servem para deslocar um determinado número de bits para a esquerda ou para a direita, envolvendo ou não o bit mais à esquerda (bit de sinal) no caso de inteiros com sinal. Não existe o operador <<<.

Na base decimal retirar ou acrescentar zeros à direita corresponde respetivamente a uma divisão ou multiplicação por 10. Na base binária retirar ou acrescentar zeros à direita corresponde respetivamente a uma divisão ou multiplicação por 2.

```
int   mbytes = 1;              // 1 kByte = 2^10 bytes e 1 Mbyte = 2^20 bytes
long  bytes  = mbytes << 20;   // acrescenta 20 bits zero à direita
System.out.println(bytes);     // imprime 1048576 = 1024*1024
```

No exemplo acima estes operadores são usados para converter rapidamente um valor de megabytes em bytes.

Operadores de atribuição

Operadores de atribuição

- Operador de atribuição normal =
- Operadores de atribuição compactos:

Operador	Sintaxe	É equivalente a:	
+=	op1 += op2	op1 = op1 + op2	
-=	op1 -= op2	op1 = op1 - op2	⌦
*=	op1 *= op2	op1 = op1 * op2	
/=	op1 /= op2	op1 = op1 / op2	⌦
%=	op1 %= op2	op1 = op1 % op2	⌦
&=	op1 &= op2	op1 = op1 & op2	
|=	op1 |= op2	op1 = op1 | op2	
^=	op1 ^= op2	op1 = op1 ^ op2	
<<=	op1 <<= op2	op1 = op1 << op2	⌦
>>=	op1 >>= op2	op1 = op1 >> op2	⌦
>>>=	op1 >>>= op2	op1 = op1 >>> op2	⌦

Java Fundamentos

Capítulo 2 - Sintaxe 21

A tabela acima mostra todos os operadores de atribuição compactos e os seus equivalentes em sintaxe não compacta. Os operadores assinalados com ⌦ **não gozam da propriedade comutativa**, pelo que a ordem dos fatores faz diferença.

O operador **=** serve para atribuir o valor da **expressão** à sua direita à variável que está à sua esquerda. Pode ser usado para inicializar uma variável ou para alterar seu valor. Já foi utilizado em exemplos anteriores.

```
int x, y, z;
x = 3789;       //recebe um literal
x = x + 1;      //contador – incremento unitario
x = x + y;      //acumulador – ao seu valor antigo é adicionado outra quantidade
x = y + 31 / z  //recebe outra expressão qualquer
```

À semelhança da linguagem C, o Java fornece um conjunto de operadores de atribuição com **sintaxe compacta,** que nos permitem efetuar, numa única instrução, uma operação aritmética, lógica ou bit a bit, juntamente com uma atribuição. As duas instruções abaixo são equivalentes:-

```
i = i + 2;
i += 2;
```

Exercícios sobre operadores

Exercícios sobre operadores

- Exercício 1 – programa que usa os operadores aritméticos simples (binários);
- Exercício 2 – programa que usa os operadores aritméticos compactos (unários);
- Exercício 3 – programa que usa os operadores relacionais;
- Exercício 4 – Verificar a optimização na avaliação de expressões lógicas com os operadores lógicos.

Java Fundamentos Capítulo 2 - Sintaxe 22

Exercício 1

Programa que usa os operadores aritméticos simples (binários).

```
package capitulo2;
public class OperAritmeticos {
    public static void main(String args[]) {
        int i=3, j=4;
        float a=2.14f, b=3.52f;
        System.out.println("i="+i+",j="+j);
        System.out.println("i+j<>"+i+j); //nao soma pois cancatena strings
        System.out.println("i+j="+(i+j));
        System.out.println("i-j="+(i-j));
        System.out.println("i*j="+(i*j));
        System.out.println("j/i="+(j/i));
        System.out.println("j%i="+(j%i));
        System.out.println("a="+a+",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("b/a="+(b/a));
        System.out.println("b%a="+(b%a));
    }
```

```
}
```

A execução deste programa produz o seguinte resultado:

```
i=3,j=4
i+j<>34
i+j=7
i-j=-1
i*j=12
j/i=1
j%i=1
a=2.14,b=3.52
a+b=5.66
a-b=-1.3799999
a*b=7.5328
b/a=1.6448597
b%a=1.3799999
```

No exemplo anterior **b%a** significa o resto da divisão inteira entre **b** e **a**. O quociente é 1, o resto é 1.38.

A linguagem Java estende o operador + para permitir a concatenação de Strings. Veja as linhas de código abaixo e os respetivo s resultados:

```
   System.out.println("i+j<>"+i+j); //nao soma pois concatena strings
   System.out.println("i+j="+(i+j));//soma
```

```
i+j<>34
i+j=7
```

O operador + “adiciona” strings para serem escritas pela função println(). No primeiro exemplo é feita a conversão automática dos números i e j para String, antes de escrever o resultado. Não é calculada a sua soma. No segundo caso a utilização de parêntesis curvos força a adição dos números antes da conversão do seu resultado para String.

Exercício 2

Programa que usa os operadores aritméticos compactos (unários).

```
package capitulo2;
```

```
public class OperAritmComp {
    public static void main(String args[]) {
        int i=3,j=3,k=3,m=3;
        System.out.println("i="+i+" j="+j+" k="+k+" m="+m);
        System.out.println("i++="+(i++)+" i="+i);
        System.out.println("++j="+(++j)+" j="+j);
        System.out.println("k--="+(k--)+" k="+k);
        System.out.println("--m="+(--m)+" m="+m);
    }
}
```

A sua execução produz o seguinte resultado:

```
i=3 j=3 k=3 m=3
i++=3 i=4
++j=4 j=4
k--=3 k=2
--m=2 m=2
```

Repare que as variáveis são alteradas sem recorrer a uma instrução de atribuição.

Exercício 3

Programa que usa os operadores relacionais.

```
package capitulo2;
public class OptimizacaoLogica {
    public static void main(String[] args) {
        int b=0;
        System.out.println("Resultado="+(2>3 && 4/b>0));
        System.out.println("Se chegar aqui, e' porque nao fiz");
        System.out.println("a divisao por zero");
    }
}
```

Exercício 4

Verificar a otimização na avaliação de expressões lógicas com os operadores lógicos.

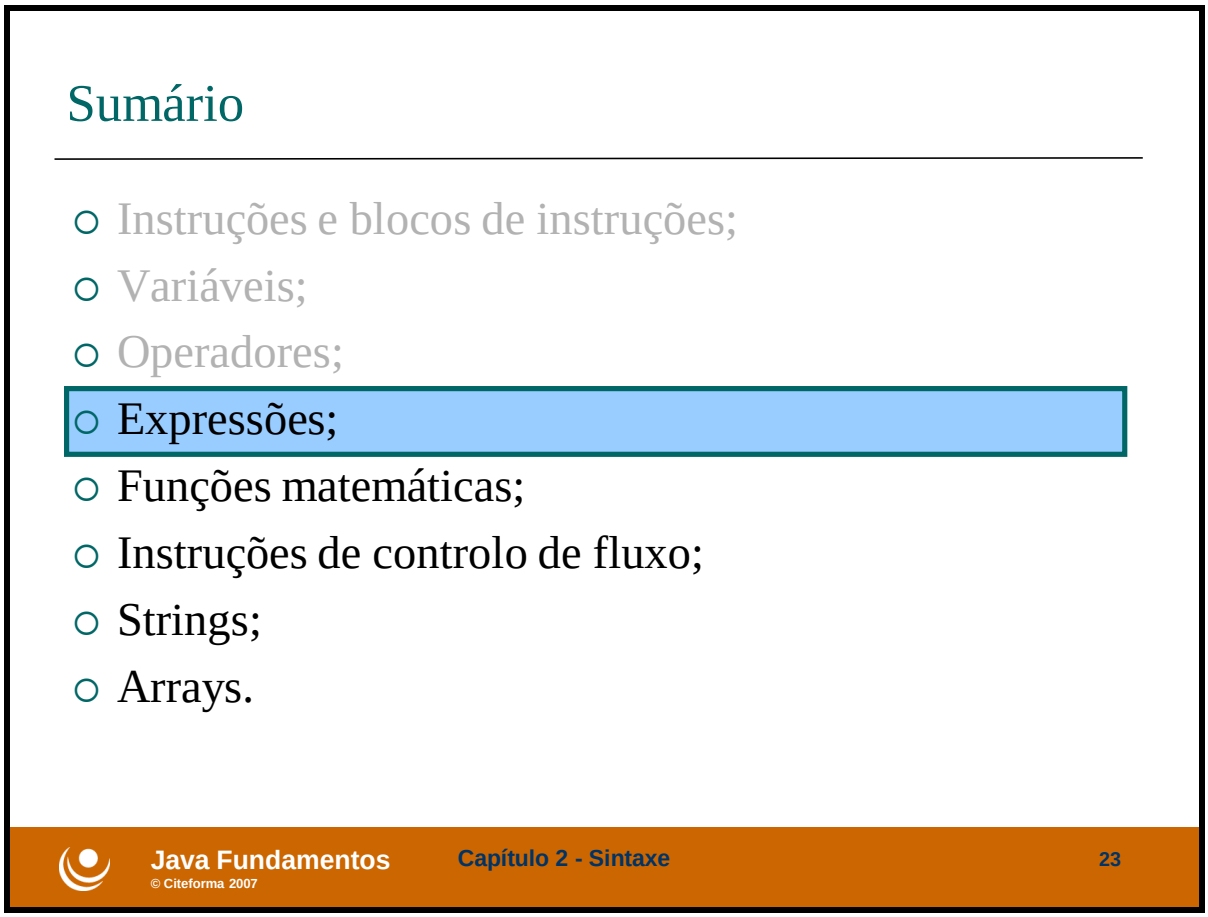
```
package capitulo2;
public class OptimizacaoLogica {
    public static void main(String[] args) {
        int b=0;
        System.out.println("Resultado="+(2>3 & 4/b>0));
        System.out.println("Se chegar aqui, e' porque nao fiz");
        System.out.println("a divisao por zero");
    }
}
```

```
java.lang.ArithmeticException: / by zero
        at OptimizacaoLogica.main(OptimizacaoLogica.java:6)
        Exception in thread "main" Process Exit...
```

O exemplo acima ilustra as **vantagens da otimização na avaliação de expressões lógicas**, sendo a mais **imediata a redução do tempo de execução**. O programa não efetua a divisão por zero, porque só é avaliado o primeiro operando.

O mesmo exemplo com o operador **&** dá um erro em tempo de execução **porque ambos operandos são avaliados**.

Expressões



As expressões executam o trabalho de um programa Java, pois são calculadas pelo computador e devolvem um valor, que depois será utilizado para atribuir um valor a uma variável ou para auxiliar a controlar o fluxo de execução do programa.

Uma expressão é constituída por uma série de variáveis, operadores ou chamadas a funções, colocados de acordo com a sintaxe da linguagem Java, e que produz como resultado um valor de um determinado tipo.

Como os operadores devolvem um valor podem ser utilizados em expressões, como por exemplo:-

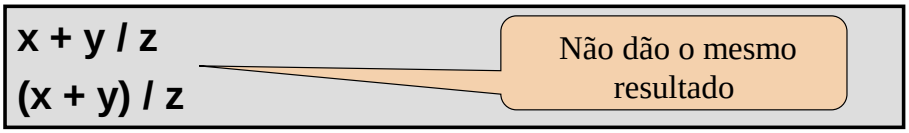
```
contador++;
```

O tipo do valor retornado por uma expressão depende dos elementos nela utilizados. A expressão acima retorna um valor do mesmo tipo da variável **contador**, pois o operador **++** retorna um valor do tipo dos seus operandos.

O que é uma expressão

O que é uma expressão?

- É constituída por uma série de variáveis, operadores, chamadas a métodos ou constantes;
- Devolve um valor;
- Os operadores nela utilizados são avaliados de acordo com uma lista de prioridades;
- Os parêntesis alteram a ordem natural de prioridades.



A ordem na qual uma expressão é avaliada pode influenciar o seu resultado.

No exemplo abaixo, como a multiplicação é comutativa, a ordem de avaliação é indiferente:

```
x * y * z       //da' o mesmo que
(x * y) * z     //da' o mesmo que
x * (y * z)
```

Na expressão abaixo obtemos resultados diferentes conforme se faça primeiro a soma ou a divisão:

```
x + y / z           //pode não dar o mesmo que
(x + y) / z
```

O uso de parêntesis em expressões como a anterior permite indicar ao compilador como queremos que a expressão seja avaliada. Quando são omitidos os parêntesis o compilador tem regras de prioridade bem definidas para avaliar a expressão.

As duas expressões seguintes são equivalentes, porque as regras de prioridade dizem que a divisão é feita antes da soma:

```
x + y / z      //da' o mesmo que
x + (y / z)
```

A utilização de parêntesis é aconselhada nas expressões complexas, pois facilita a sua leitura e interpretação pelas pessoas que leem o programa. Caso os parêntesis sejam omitidos, o compilador vai interpretar a expressão executando primeiro os operadores com maior prioridade. Quando são encontrados dois operadores com a mesma prioridade, é executado primeiro o que estiver à esquerda.

Prioridade dos operadores nas expressões

Prioridade dos operadores nas expressões #1/2

Prioridade	Grupo de operadores	Operadores
15	Pós Fixos	[], ., (params), expr++, expr--
14	Pré fixos	++expr, --expr, +expr, -expr
13	Criação ou conversão	new, (tipo)expr
12	Multiplicativos	*, /, %
11	Aditivos	+, -
10	Deslocamento bit a bit	<<, >>, >>>
9	Relacionais	<, >, <=, >=, instanceof
8	Igualdade	==, !=
7	AND bit a bit	&
6	XOR bit a bit	^
5	OR bit a bit	|
4	AND condicional	&&
3	OR condicional	||
2	if compacto	condicao?instrucao1:instrucao2
1	Atribuição	=, +=, -=, *=, /=, %=, |=, ^=, &=, <<=, >>=, >>>=

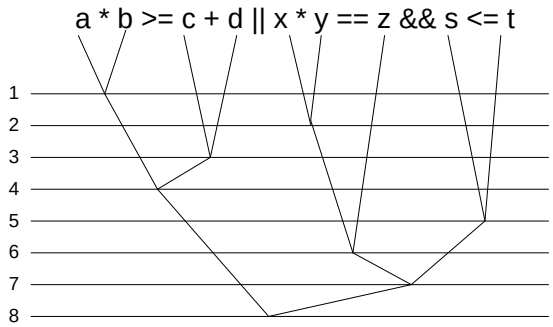
A tabela acima descreve as prioridades associadas a cada operador. O valor 15 representa a maior prioridade.

Prioridade dos operadores nas expressões #2/2

a*b>=c+d||x*y==z&&s<=t

É avaliada como:

(((a*b)>=(c+d))||((x*y)==z))&&(s<=t)



A expressão do exemplo acima é avaliada da esquerda para a direita, tendo em conta que os operadores aritméticos têm maior prioridade face aos operadores relacionais, e que estes últimos têm maior prioridade face aos operadores lógicos.

Funções matemáticas

Sumário

- Instruções e blocos de instruções;
- Variáveis;
- Operadores;
- Expressões;
- Funções matemáticas;
- Instruções de controlo de fluxo;
- Strings;
- Arrays.

Funções matemáticas

Função	Descrição
Math.sqrt()	Raiz quadrada
Math.sin()	Seno
Math.con()	Coseno
Math.tan()	Tangente
Math.asin()	Arco cujo seno é x
Math.atan()	Arco cuja tangente é x
Math.exp()	Exponencial de base e
Math.log()	Logaritmo de base e
Math.pow(x,a)	x^a
Math.toDegrees()	Conversão de radianos em graus
Math.toRadians()	Conversão de graus em radianos

Java Fundamentos 28
© Citeforma 2007
Capítulo 2 - Sintaxe

A classe Math contém um conjunto de funções matemáticas das quais se listam algumas na tabela acima. Na classe Math estão também definidas duas constantes: Math.PI e Math.E.

Exercício

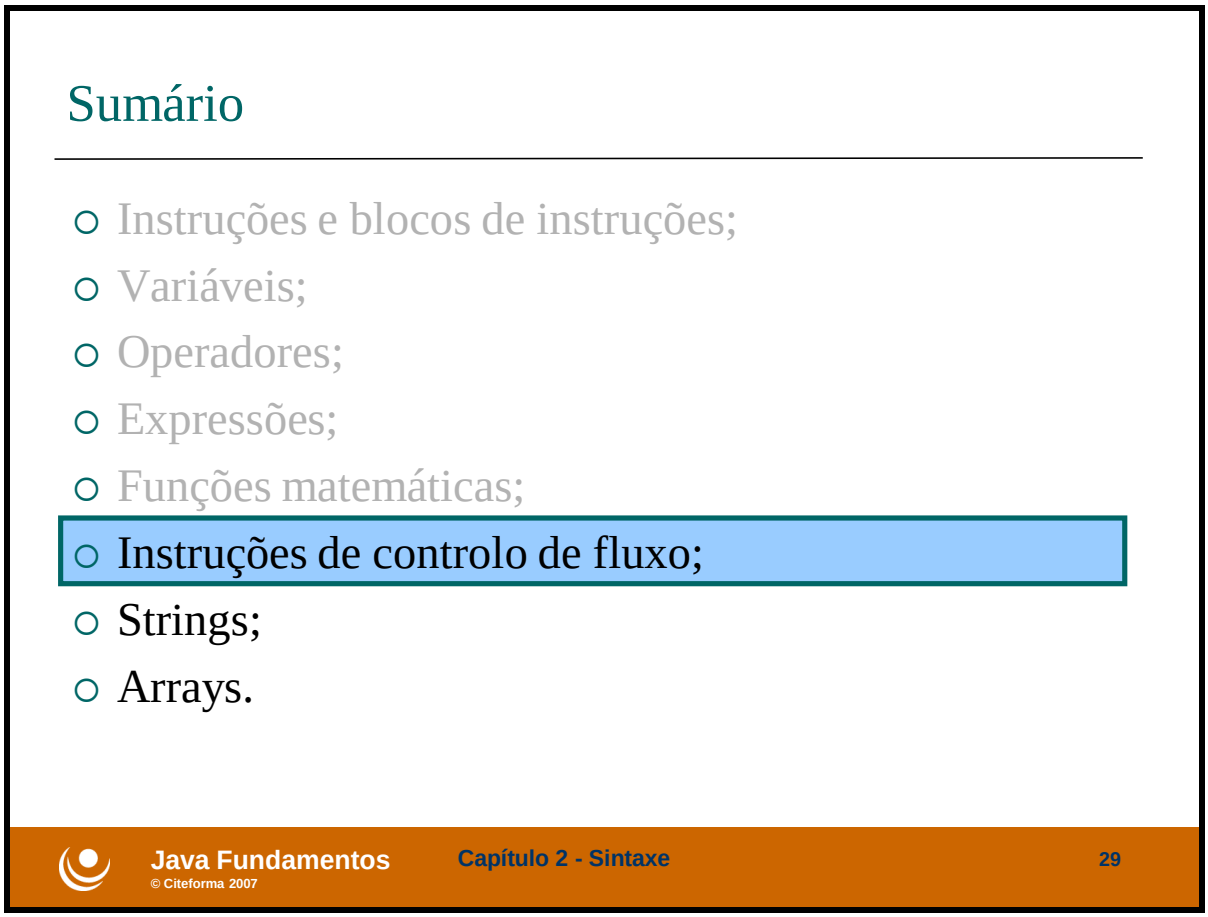
O programa abaixo mostra como se usam estas funções:

```
package capitulo2;
public class FuncoesMatematicas01 {
    public static void main(String args[]) {
        double d1=4, d2;
        d2 = Math.sqrt(d1);
        System.out.println(d2);
        System.out.println(Math.sin(Math.PI/2.0));
    }
}
```

Este programa produz o seguinte resultado:-

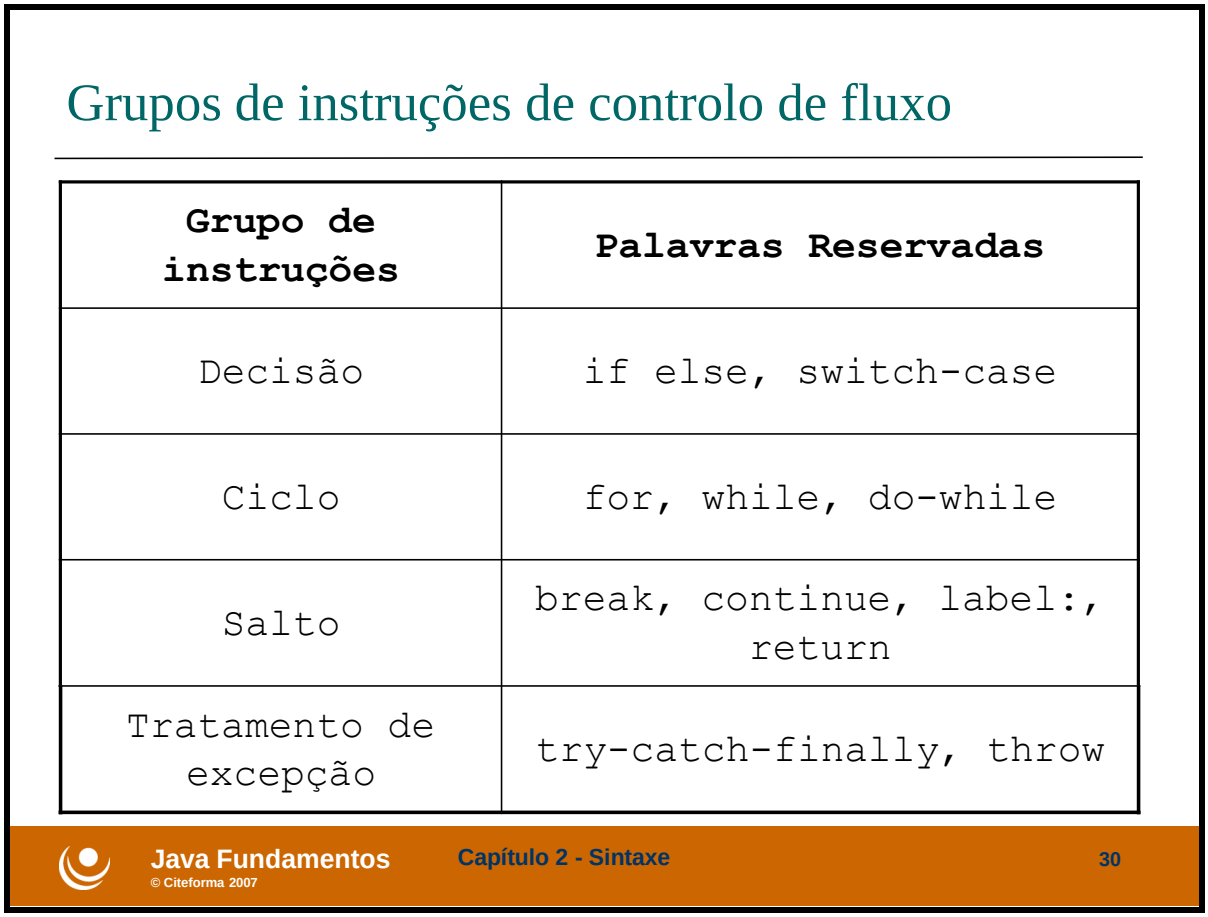
```
2.0
1.0
```

Instruções de controlo de fluxo



Os exemplos apresentados até aqui têm um único fluxo de execução. As instruções de controlo de fluxo permitem criar fluxos de execução alternativos e/ou repetitivos (iterativos).

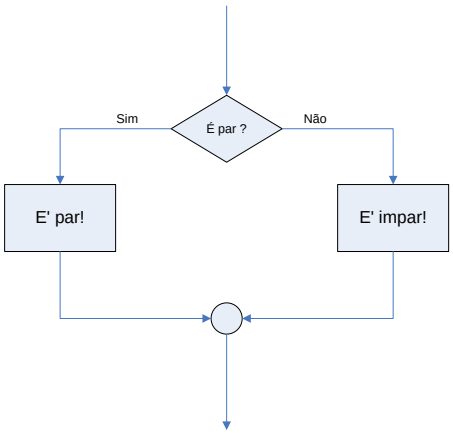
Grupos de instruções de controlo de fluxo



As instruções de controlo de fluxo podem ser agrupadas como indicado no quadro acima.

Decisão: if

Decisão: if #1/2




```
int n=12;
if (n % 2 == 0) {
    System.out.println("E' Par");
} else {
    System.out.println("E' Impar");
}
System.out.println("Acabei !");
```

Java Fundamentos
© Citeforma 2007
Capítulo 2 - Sintaxe
31

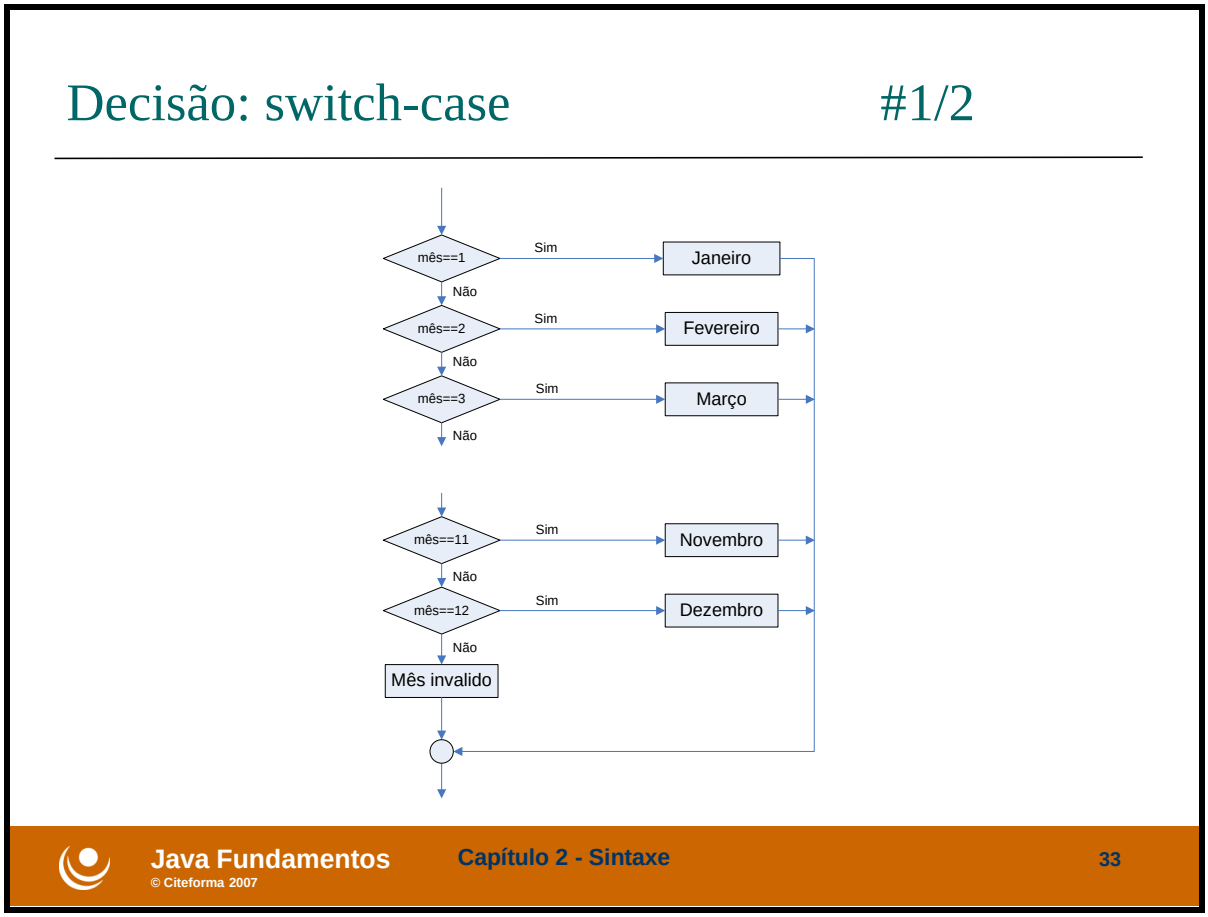
A instrução **if** permite criar dois caminhos alternativos de execução. Um dos caminhos será executado se a condição do if for verdadeira, sendo executado o outro caso seja falsa.

O bloco de instruções que serão executadas caso a condição seja verdadeira estão limitadas por { } e aparecem logo a seguir à condição, que terá que estar limitada por parêntesis curvos. O outro bloco aparece depois a palavra reservada **else**. A sintaxe da linguagem Java permite não colocar a instrução **else**, o que significa que caso a condição seja falsa nada é executado.

Decisão: if encadeado

O exemplo acima utiliza uma sequência de instruções if encadeadas.

Decisão: switch-case



A instrução switch-case é herdada da linguagem C e comporta-se como um conjunto de instruções if encadeadas. A sintaxe do **switch** facilita a leitura do código mas apenas permite comparações com igualdade entre números inteiros ou variáveis char. O tipo **long** não pode ser usado no **switch**.

Decisão: switch-case #2/2

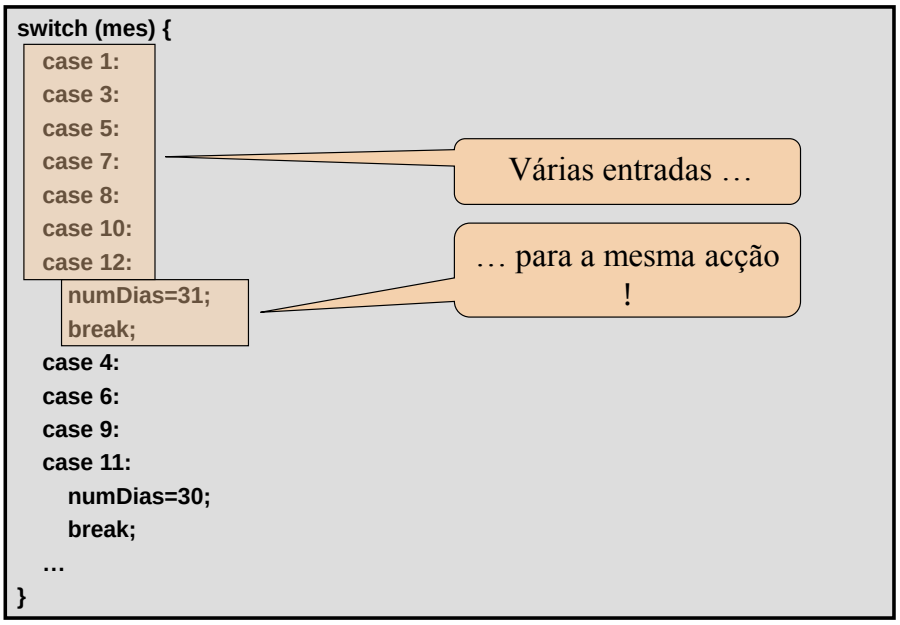
```
int mes=3;
switch (mes) {
   case 1:
      System.out.println("Janeiro");
      break;
   case 2:
      System.out.println("Fevereiro");
      break;
   . . .
   case 12:
      System.out.println("Dezembro");
      break;
   default:
      System.out.println("Nao e' um mes valido");
      break;
}
```

Java Fundamentos Capítulo 2 - Sintaxe 34

O exemplo acima mostra uma instrução **switch-case**. A instrução **break** define quando termina o bloco de instruções definidas para uma entrada.

O bloco de instruções **default** é executado quando nenhum dos outros blocos é executado. Isto acontece quando o valor da variável está fora dos valores colocados nos **case**.

Decisão: switch-case – várias entradas uma saída



O exemplo acima determina o número de dias do mês, o que obriga a ter em conta o ano, pois para o mês de Fevereiro é necessário verificar se o ano é bissexto.

O processo de verificação se o ano é bissexto, baseia-se nos múltiplos de 4, excluindo os que são múltiplos de 100, mas incluindo os que são múltiplos de 400. Isto é válido para os anos do calendário Gregoriano superiores a 1752, altura da Bula Papal que fez um acerto do calendário.

O exemplo usa uma instrução **switch-case** que permite várias entradas para o mesmo conjunto de ações.

Exercícios sobre instruções de controlo de fluxo - Decisão

Exercícios sobre instruções de controlo de fluxo - Decisão

- Exercício 1 – usar if-else encadeados para obter nota qualitativa em função de nota quantitativa;
- Exercício 2 – usar switch-case para obter o nome do mês tendo o seu número (1..12) ;
- Exercício 3 – usar switch-case para obter o número de dias do mês, tendo em conta os anos bissextos.

Java Fundamentos Capítulo 2 - Sintaxe 36
© Citeforma 2007

Exercício 1

O programa abaixo converte uma nota quantitativa em nota qualitativa usando um conjunto de instruções if encadeadas:

```
package capitulo2;
public class IfElseIf {
    public static void main(String args[]) {
        int nota=13;
        String s;
        if (nota>=18) {
            s="Muito Bom";
        } else {
            if (nota>=14) {
                s="Bom";
            } else {
                if (nota>=10) {
                    s="Suficiente";
                } else {
                    if (nota>=7) {
                        s="Mediocre";
                    } else {
```

```
                        s="Mau";
                    }
                }
            }
        }
        System.out.println("Nota qualitativa="+s);
    }
}
```

As instruções e sub instruções foram arrumadas para que o alinhamento vertical evidencie a sua hierarquia. A variável nota recebe 13, pelo que o resultado produzido será:-

```
Nota qualitativa=Suficiente
```

Exercício 2

O programa abaixo converte o número do mês no respetivo nome. Para isso utiliza uma instrução switch-case:

```
package capitulo2;
public class Switch01 {
    public static void main(String args[]) {
        int mes=3;
        switch (mes) {
            case 1:
                System.out.println("Janeiro");
                break;
            case 2:
                System.out.println("Fevereiro");
                break;
            case 3:
                System.out.println("Marco");
                break;
            case 4:
                System.out.println("Abril");
                break;
            case 5:
                System.out.println("Maio");
                break;
            case 6:
                System.out.println("Junho");
                break;
            case 7:
                System.out.println("Julho");
                break;
            case 8:
                System.out.println("Agosto");
                break;
            case 9:
                System.out.println("Setembro");
                break;
            case 10:
                System.out.println("Outubro");
                break;
            case 11:
```

```
                System.out.println("Novembro");
                break;
            case 12:
                System.out.println("Dezembro");
                break;
            default:
                System.out.println("Nao e' um mes valido");
                break;
        }
    }
}
```

A variável mês recebe o valor 3 pelo que o programa produz o seguinte resultado:-

```
Marco
```

Exercício 3:

O programa abaixo determina o número de dias do mês a partir do ano e número do mês, tendo em conta os anos bissextos.

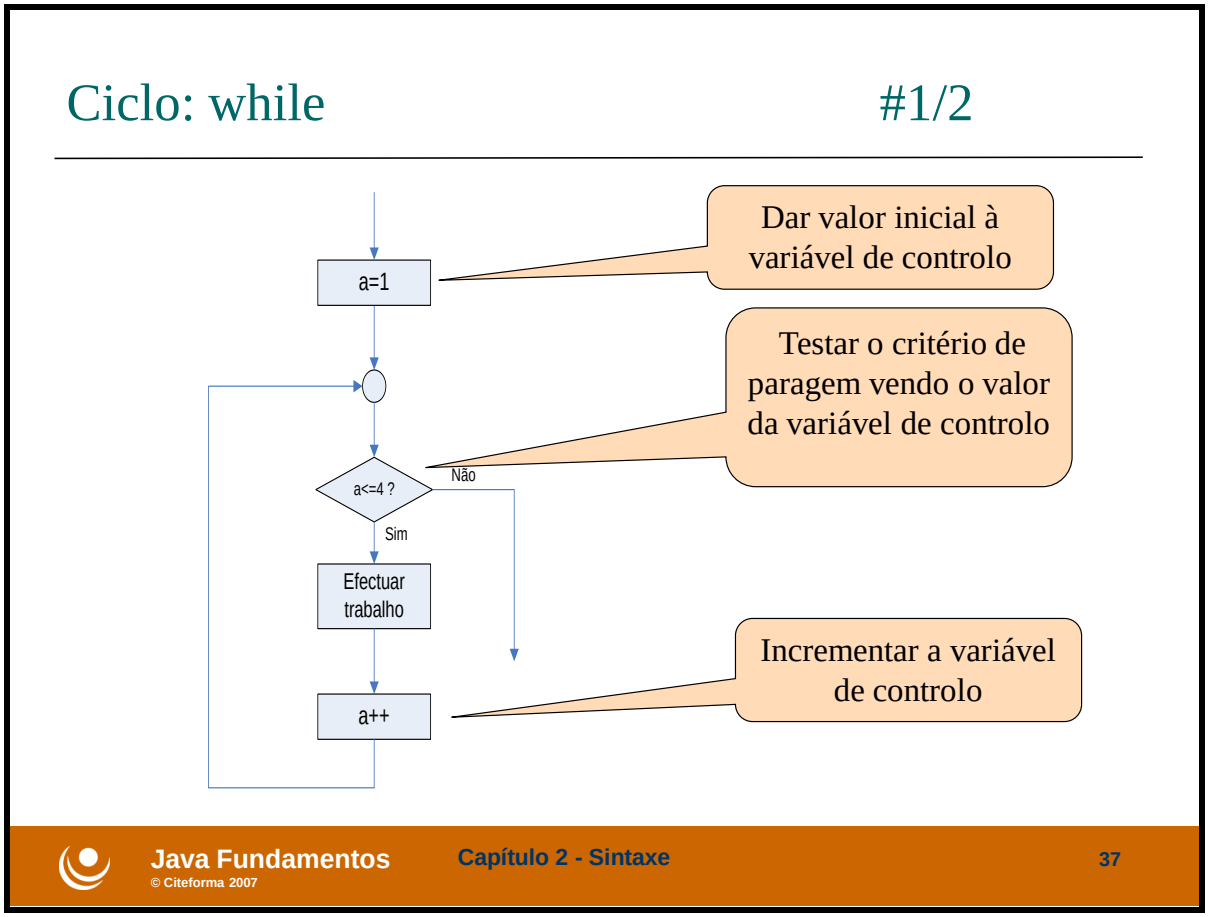
```
package capitulo2;
public class Switch02 {
    public static void main(String args[]) {
        int mes=2, ano=2100, numDias=0;
        switch (mes) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                numDias=31;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                numDias=30;
                break;
            case 2:
                /*so funciona quando o ano e' >= 1752 */
                if ((ano%4==0) && ((ano%100!=0) || (ano%400==0))) {
                    numDias=29;
                }else{
                    numDias=28;
                }
                break;
            default:
                System.out.println("Nao e' um mes valido");
                numDias=0;
                break;
        }
        if (numDias != 0) {
            System.out.println("Numero de dias do mes="+numDias);
        }
    }
```

```
}
```

A colocação da instrução break permite que vários pontos de entrada executem o mesmo código. O programa anterior produz o seguinte resultado:

```
Numero de dias do mes=28
```

Ciclo: while



A instrução **while** permite definir um bloco de instruções cuja execução se repete enquanto a condição é verdadeira.

A repetição de instruções é controlada por uma variável de controlo, cujo ciclo de vida tem 3 momentos que estão assinalados na figura anterior:

- Atribuir um valor inicial à variável de controlo, o que é feito fora do ciclo;
- **Testar o valor da variável de controlo**. Se a expressão lógica devolver **true** o fluxo de execução entra no ciclo. Se devolver false sai do ciclo;
- **Incrementar a variável de controlo**, o que é feito antes do novo teste à expressão lógica. Este incremento deve ser convergente, ou seja, deve conduzir a um valor falso na expressão lógica. Se não for convergente teremos um ciclo infinito, o que não é desejável;

Um ciclo while pode ser construído usando variações desta estrutura, o que o torna uma instrução muito flexível.

Ciclo: while #2/2

```
package capitulo2;
public class While01 {
   public static void main(String[] args) {
      int soma=0, a=1;
      while (a<=4) {
         soma += a;
         System.out.println(soma+"  "+a);
         a++;
      }
      System.out.println("Soma="+soma);
   }
}
```

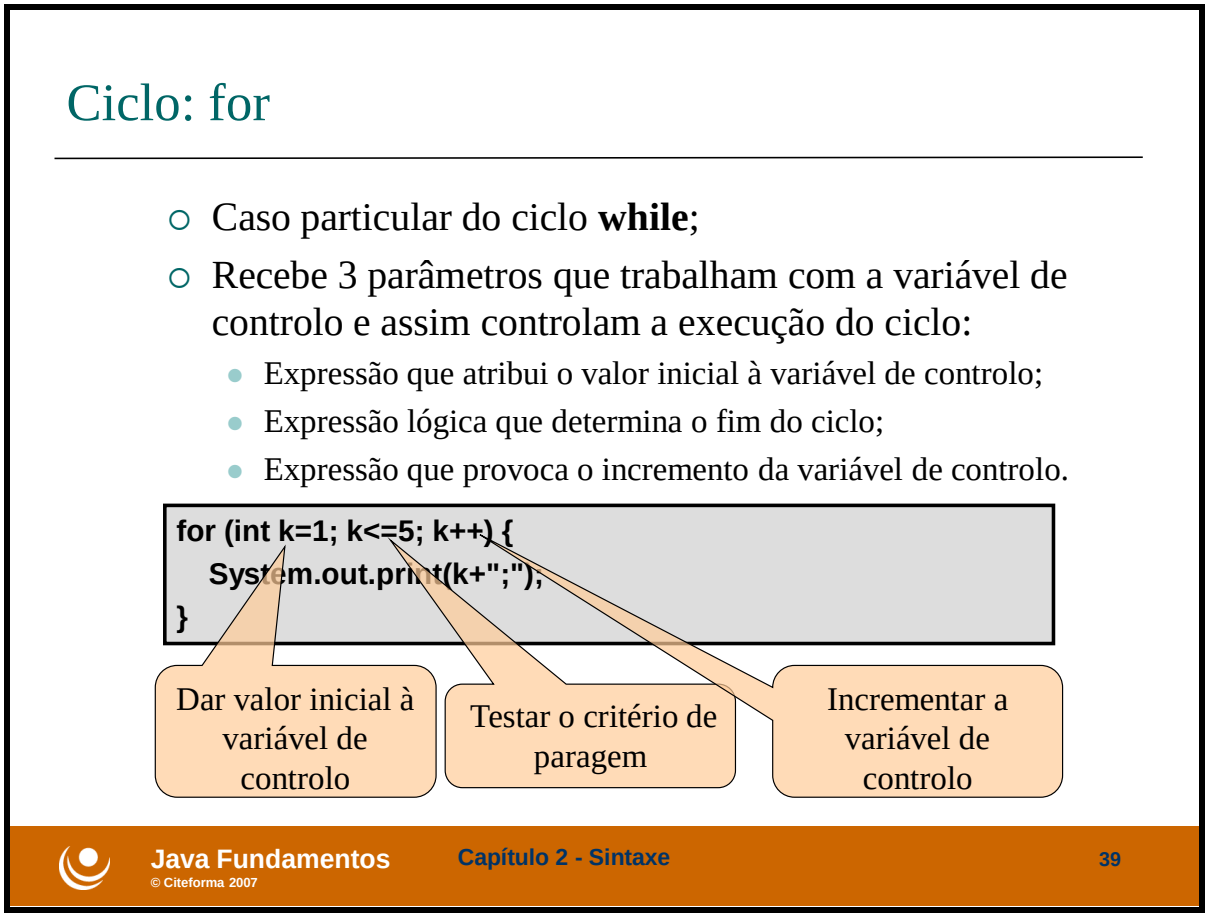
Dar valor inicial à variável de controlo

Testar o critério de paragem vendo o valor da variável de controlo

Incrementar a variável de controlo

Acima apresenta-se a implementação em linguagem Java do ciclo anterior, recorrendo à instrução while.

Ciclo: for



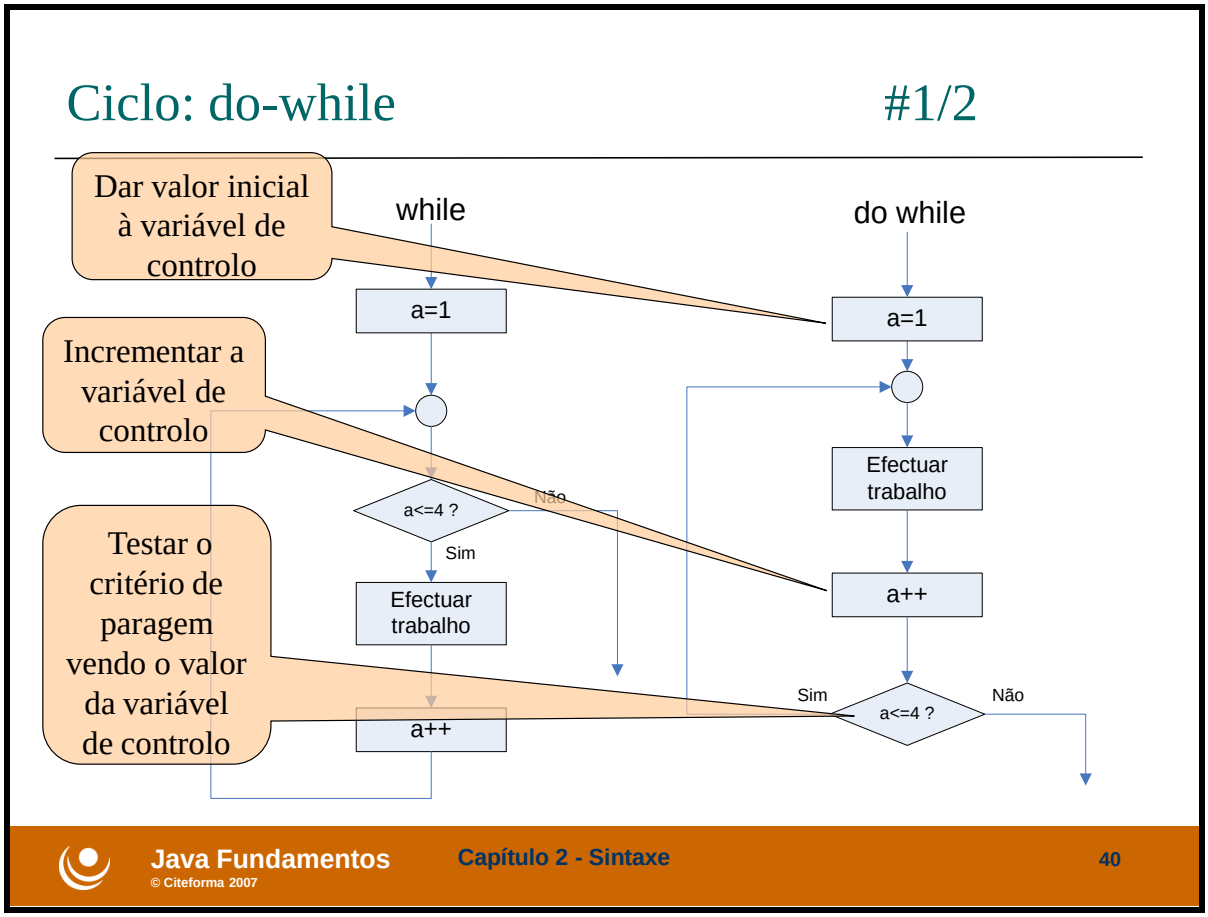
O ciclo **for** pode ser considerado um caso particular do ciclo **while**, pois corresponde-lhe um fluxograma igual ao que foi apresentado no ponto anterior.

O ciclo **while** é mais flexível que o ciclo **for**, pois pode ter ligeiras variações na sua estrutura, ao contrário do segundo que é rígido. No entanto, a definição do ciclo **for** poupa trabalho ao programador e aumenta a clareza do código, pois recebe 3 parâmetros que correspondem aos 3 momentos importantes na vida da variável que controla a execução do ciclo:

- Expressão que atribui o valor inicial à variável de controlo;
- Expressão lógica que determina o fim do ciclo;
- Expressão que provoca o incremento da variável de controlo.

Nota: O primeiro passo (atribuir o valor inicial à variável de controlo) é sempre executado, mesmo que a condição que determina o fim de ciclo seja falsa.

Ciclo: do-while



A instrução **do-while** permite construir ciclos semelhantes ao **while** mas com uma diferença: a condição que testa o fim do ciclo é feita depois de processado o bloco de instruções, o que implica que esse bloco seja executado pelo menos uma vez, enquanto no ciclo while pode nunca ser executado, pois o teste é feito no início.

O fluxograma apresentado acima mostra essas diferenças.

Ciclo: do-while #2/2

- O ciclo do-while é muito usado nas validações de entrada de dados;
- No exemplo abaixo só sai do ciclo quando o ano for superior a 1752:

```
int ano;
do {
    String s = javax.swing.JOptionPane.showInputDialog(
                "Qual o ano?");
    ano = Integer.parseInt(s);
} while (ano<=1752);
```

Java Fundamentos Capítulo 2 - Sintaxe 41

O exemplo acima utiliza um ciclo **do-while** para fazer uma validação de input. O ano é introduzido recorrendo a uma JOptionPane que permite a leitura de uma String. Esta é convertida num número que será comparado com o limite 1752. Caso seja um ano anterior é pedido ao utilizador que reintroduza o ano.

As String serão abordadas mais à frente neste capítulo.

Os objetos Swing pertencem ao package – javax.swing (parte gráfica do Java).

Exercícios sobre instruções de controlo de fluxo - Ciclo

Exercícios sobre instruções de controlo de fluxo - Ciclo

- Exercício 1 – usar um ciclo while para obter a soma dos 4 primeiros números inteiros;
- Exercício 2 – usar um ciclo for para demonstrar os argumentos do ciclo;
- Exercício 3 – usar um ciclo for para obter o factorial de um número inteiro inferior ou igual a 12. Para os superiores a variável factorial tem que ser do tipo double;
- Exercício 4 – usar um ciclo do-while para validar se o valor introduzido pelo utilizador é um ano superior a 1752. O valor é pedido usando uma Dialog do Swing.

Java Fundamentos Capítulo 2 - Sintaxe 42
© Citeforma 2007

Exercício 1

O exercício acima determina a soma de 4 números consecutivos começando em 0:

```
package capitulo2;

public class While01 {
   public static void main(String[] args) {
      int soma=0, a=1;
      while (a<=4) {
         soma += a; //acumula 'soma' e incrementa 'a' na mesma instrucao
         System.out.println(soma+"  "+a);
         a++;
      }
      System.out.println("Soma="+soma);
   }
}
```

Este programa produz o seguinte resultado:

```
1  1
3  2
6  3
10  4
Soma=10
```

Exercício 2

Este programa mostra o output de vários ciclos for encadeados, mostrando diferentes alternativas para construir o ciclo:

```
package capitulo2;

public class For01 {
   public static void main(String[] args) {
      long k;
      for (k = 1; k <= 5; k++)
         System.out.print(k + ";");
      System.out.println(".");
      for (k = 1; k++ <= 5; )     // ++k
         System.out.print(k + ";");
      System.out.println(".");
      k = 1;
      for ( ; ++k <= 5; )         // k=1 e ++k
         System.out.print(k + ";");
      System.out.println(".");
    }
}
```

Este programa produz o seguinte resultado:-

```
1;2;3;4;5;.
2;3;4;5;6;.
2;3;4;5;.
```

É de notar que é possível omitir o primeiro e o terceiro dos argumentos do ciclo **for:**

- O **primeiro** desde que a **variável seja inicializada previamente;**
- O **terceiro** desde que a variável e controlo seja alterada noutra instrução dentro do ciclo.

Exercício 3

A função fatorial devolve um número inteiro e cresce muito depressa. O próximo exemplo calcula o fatorial de números sucessivamente maiores, guardando o resultado numa variável inteira e numa variável long.

```
package capitulo2;

public class For02a {
   public static void main(String args[]) {
      int i, n=17;
      int factI = 1;
      long factL = 1;
      // calculo do fatorial onde de uma variável inteira...
      // em que o máximo só vai até 12...
      for (i = 1; i<=n; i++) {
         factI *= i;
         factL *= i;
         System.out.println(i + "!=" + factI + " \t " + factL);
      }
   }
}
```

A execução deste programa produz o seguinte resultado:

```
1!=1    1
2!=2    2
3!=6    6
4!=24   24
5!=120          120
6!=720          720
7!=5040         5040
8!=40320        40320
9!=362880       362880
10!=3628800     3628800
11!=39916800    39916800
12!=479001600           479001600
13!=1932053504          6227020800
14!=1278945280          87178291200
15!=2004310016          1307674368000
16!=2004189184          20922789888000
17!=-288522240          355687428096000
```

O output mostra números iguais até ao fatorial de 12, sendo diferentes daqui para a frente. O fatorial de 13 é um número demasiado grande para o domínio suportado por uma variável int, o que deveria gerar um erro de "*overflow*". Este programa mostra que o "*overflow*" dessa variável não é detetado e portanto não é emitido nenhum alerta, o que **gera resultados errados**.

O próximo exemplo mostra uma técnica que podemos usar para corrigir esta situação. A linguagem Java disponibiliza uma constante que guarda o maior valor inteiro. Se os cálculos

forem feitos com uma variável que tem maior precisão que a pretendida, conseguimos detetar o "*overflow*" fazendo a comparação com o maior inteiro disponível:

```
package capitulo2;

public class For02b {
   public static void main(String args[]) {
      int i, n=20;
      float fact = 1.0f;
      for (i = 1; i <= n; i++) {
         fact *= i;
         // maneira de contornar o valor máximo de um inteiro...
         if (fact > Integer.MAX_VALUE) {
            System.out.println(i + "!=" +" Erro!!! Máximo inteiro excedido");
         } else {
            System.out.println(i + "!=" + (int) fact);
         }
      }
   }
}
```

A execução deste programa produz o seguinte resultado:

```
1!=1
2!=2
3!=6
4!=24
5!=120
6!=720
7!=5040
8!=40320
9!=362880
10!=3628800
11!=39916800
12!=479001600
13!= Erro!!! Máximo inteiro excedido
14!= Erro!!! Máximo inteiro excedido
15!= Erro!!! Máximo inteiro excedido
16!= Erro!!! Máximo inteiro excedido
17!= Erro!!! Máximo inteiro excedido
```

Exercício 4

O programa abaixo pede ao utilizador um ano e vai determinar se é bissexto:

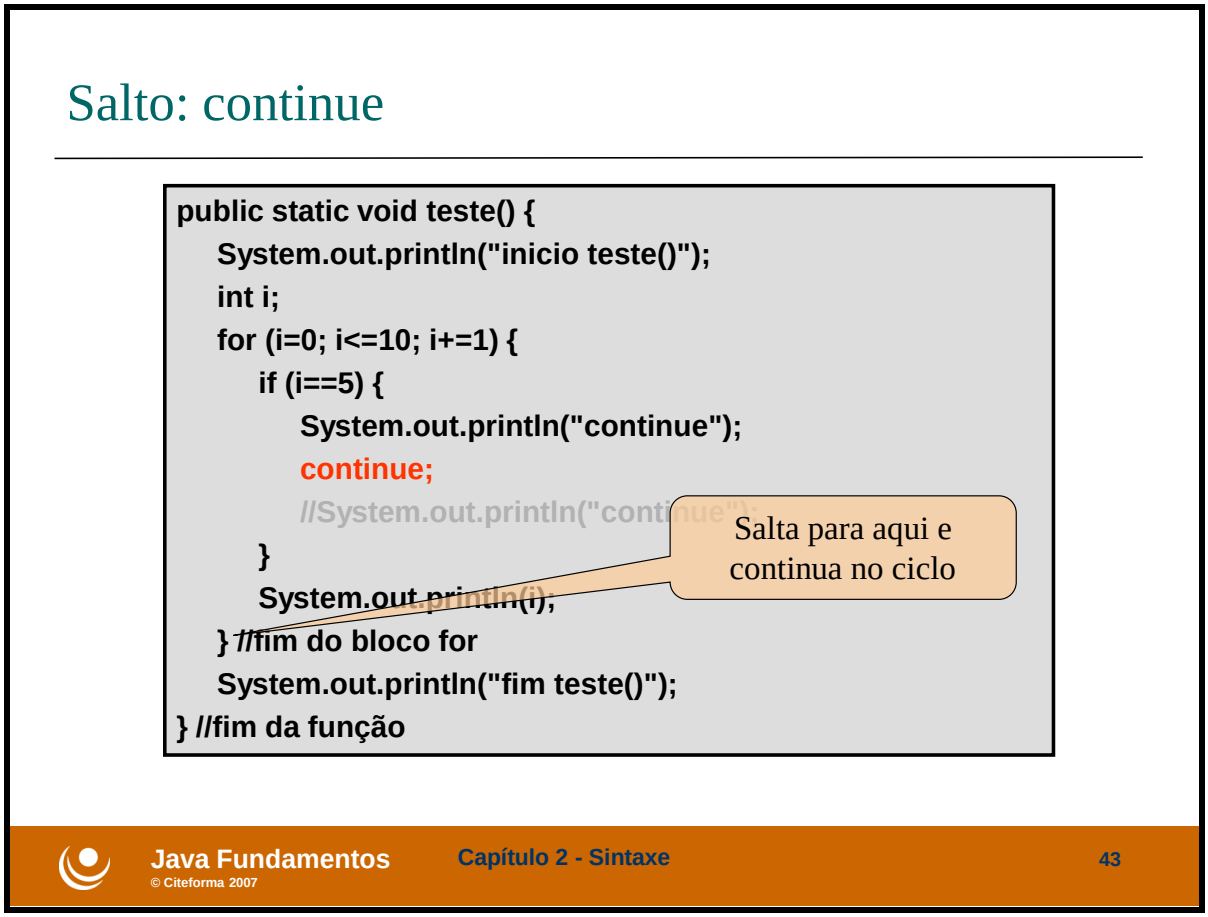
```
package capitulo2;

public class DoWhile01 {
   public static void main(String args[]) {
      int ano;
      do {
```

```
         String s = javax.swing.JOptionPane.showInputDialog("Qual o ano?");
         ano = Integer.parseInt(s);
      } while (ano<=1752);
      // e' bissexto se for divisivel por 4, exceto  os divisiveis por 100
      // mas contando com os divisiveis por 400
      // 1900 nao e' bissexto, mas 2000 e' bissexto
      if ((ano%4==0) && ((ano%100!=0) || (ano%400==0))) {
         System.out.println(ano+" e' bissexto!");
      } else {
         System.out.println(ano+" nao e' bissexto!");
      }
   }
}
```

O ciclo **do-while** é muito utilizado nas validações de entrada de dados, pois primeiro é feita uma leitura de um valor, que em seguida é validado, repetindo-se a leitura se o teste falhar.

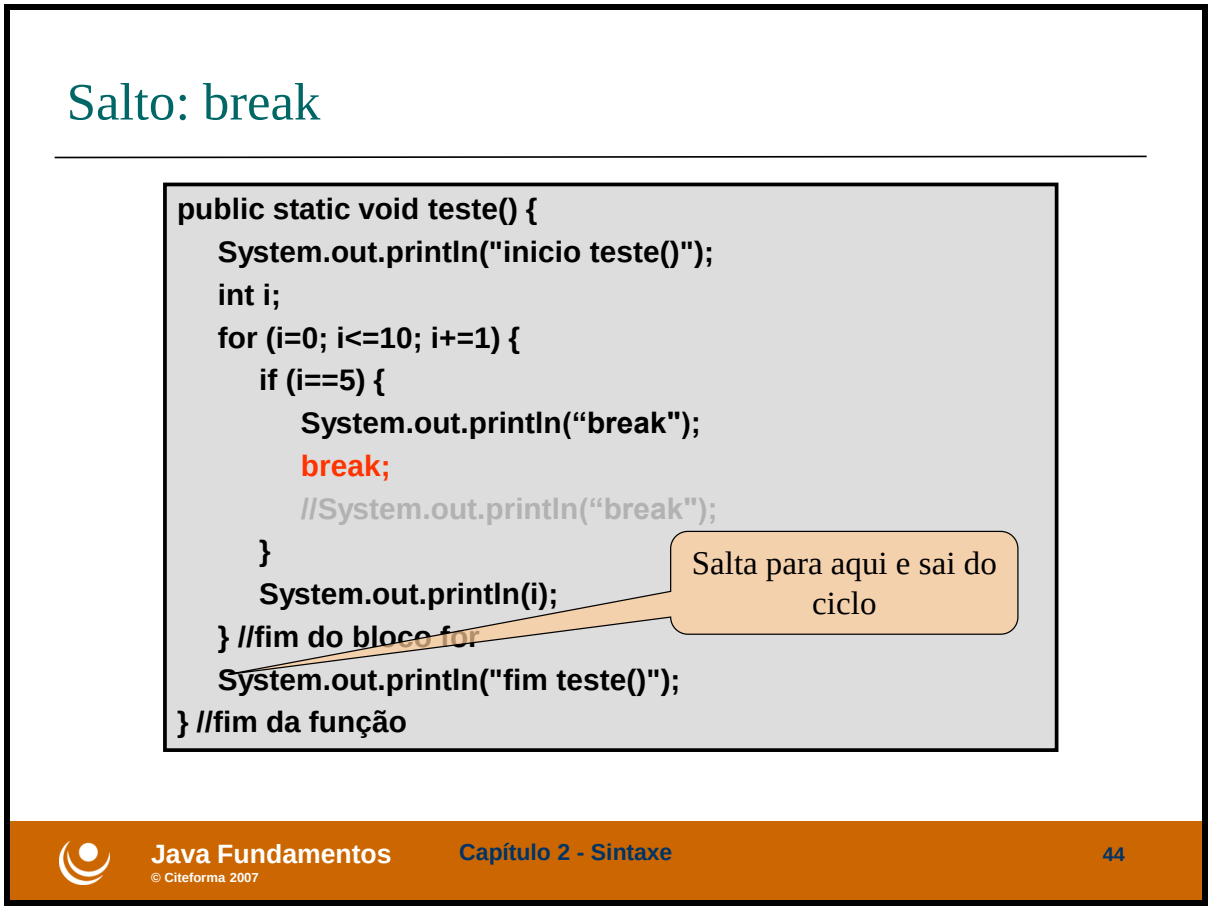
Este programa utiliza o método javax.swing.JOptionPane.showInputDialog() que serve para abrir uma caixa de diálogo, pedindo ao utilizador um dado através de uma pergunta. Este método pertence ao Java package javax.swing (parte gráfica do Java).

Salto: continue



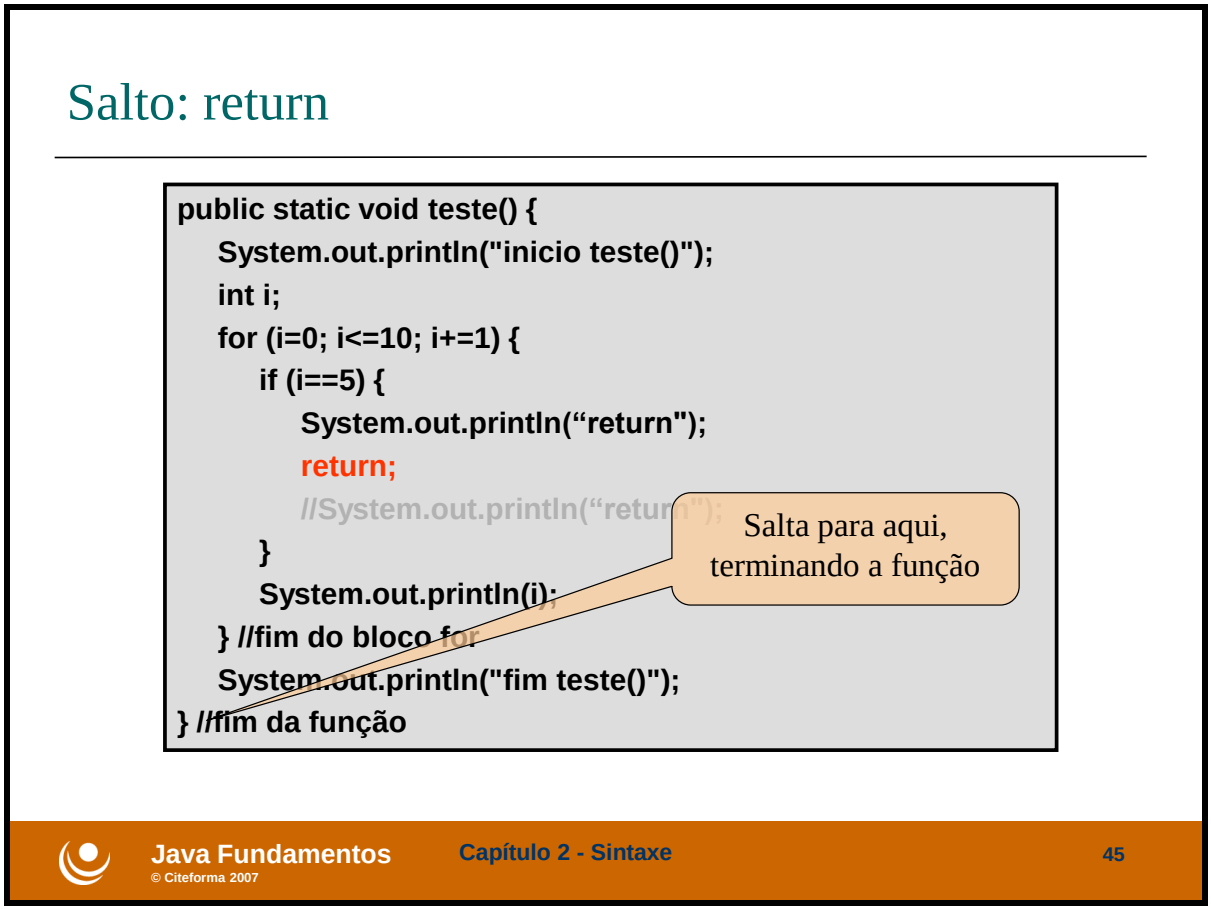
A instrução continue é utilizada para evitar a execução de uma iteração num ciclo. É considerada uma instrução de salto pois força o fluxo de execução a saltar para o fim do bloco corrente.

Salto: break



A instrução **break** é utilizada para forçar o programa a sair de um ciclo, continuando na primeira instrução imediatamente a seguir ao bloco que constitui o ciclo.

Salto: return



A instrução **return** é utilizada para forçar o programa a sair da execução da função atual, regressando à função "chamadora".

Exercícios sobre instruções de controlo de fluxo - Salto

Exercícios sobre instruções de controlo de fluxo - Salto

- Exercício 1 – desenvolver um método main que invoca uma função que tem um ciclo **for** com um contador. A execução deste ciclo é interrompida a meio com **continue**.
- Exercício 2 – Alterar o programa anterior para interromper o ciclo com **break**;
- Exercício 3 – Alterar o programa anterior para interromper o ciclo com **return**;

Java Fundamentos Capítulo 2 - Sintaxe 46
© Citeforma 2007

Os próximos três exercícios servem para evidenciar as diferenças entre as instruções **continue**, **break** e **return**. Para isso recorrem à utilização da função teste(), que é invocada dentro da função main().

Exercício 1

```
/* Continue: salta para a iteracao seguinte no ciclo! Ignora instruccoes
 *     dentro do ciclo e abaixo do continue. */
package capitulo2;
public class Continue01 {
    public static void teste() {
        System.out.println("inicio teste()");
        int i;
        for (i=0; i<=10; i+=1) {
            if (i==5) {
                System.out.println("continue");
                continue;
```

```
                // System.out.println("continue");
            }
            System.out.println(i);
        } // fim do bloco for
        System.out.println("fim teste()");
    }

    public static void main(String args[]) {
        System.out.println("inicio programa");
        teste();
        System.out.println("fim programa");
    }
}
```

A sua execução produz o seguinte resultado:

```
inicio programa
inicio teste()
0
1
2
3
4
continue
6
7
8
9
10
fim teste()
fim programa
```

A instrução **continue** provoca o salto para o fim do bloco **for**, não executando a instrução System.out.println() nele contida e continuando para a iteração seguinte.

A instrução **System.out.println("continue")** está comentada. Se retirar o comentário obterá um erro de compilação, resultante do facto de este código nunca poder ser executado ("*unreachable*"), visto que a instrução **continue** força o salto por cima dele.

Exercício 2

Este exemplo é equivalente ao do ponto anterior, com as modificações assinaladas a amarelo (nome da classe e break):

```
/* Break: Força a saida do ciclo! */
package capitulo2;

public class Break01 {
    public static void teste() {
```

```
        System.out.println("inicio teste()");
        int i;
        for (i=0; i<=10; i+=1) {
            if (i==5) {
                System.out.println("break");
                break;
                // System.out.println("break");
            }
            System.out.println(i);
        }
        System.out.println("fim teste()");
    }

    public static void main(String args[]) {
        System.out.println("inicio programa");
        teste();
        System.out.println("fim programa");
    }
}
```

A sua execução produz o seguinte resultado:

```
inicio programa
inicio teste()
0
1
2
3
4
break
fim teste()
fim programa
```

A instrução break provoca um salto para a primeira instrução imediatamente a seguir ao bloco, abortando as iterações que faltam no ciclo **for**.

Exercício 3

O exemplo apresentado a seguir é equivalente ao do ponto anterior, com as modificações assinaladas a amarelo (nome da classe e return):

```
/* Return: Sai da funcao. Nao executa a ultima instrucao */
package capitulo2;

public class Return01 {
    public static void teste() {
        System.out.println("inicio teste()");
        int i;
        for (i=0; i<=10; i+=1) {
            if (i==5) {
                System.out.println("return");
                return;
                // System.out.println("return");
```

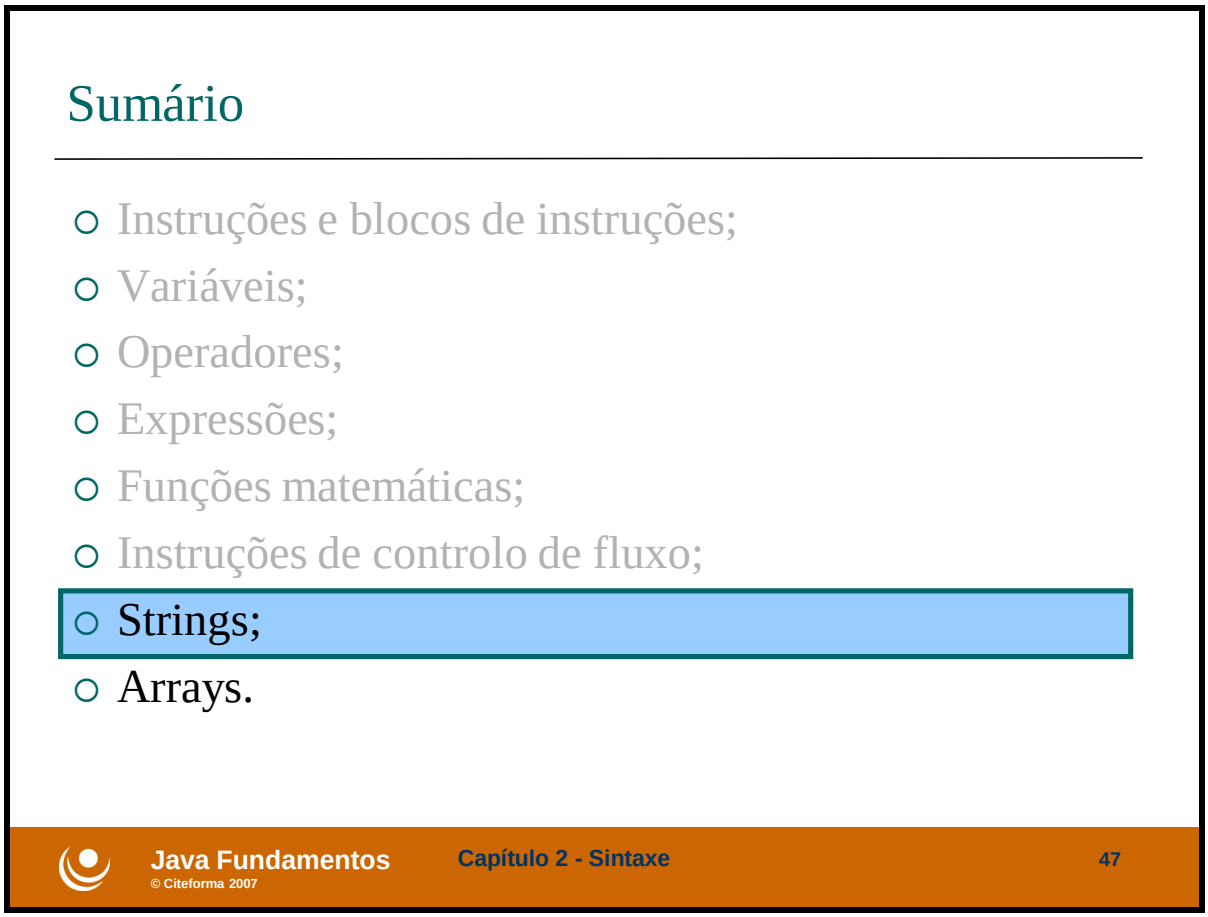
```
            }
            System.out.println(i);
        }
        System.out.println("fim teste()");
    }

    public static void main(String args[]) {
        System.out.println("inicio programa");
        teste();
        System.out.println("fim programa");
    }
}
```

A sua execução produz o seguinte resultado:

```
inicio programa
inicio teste()
0
1
2
3
4
return
fim programa
```

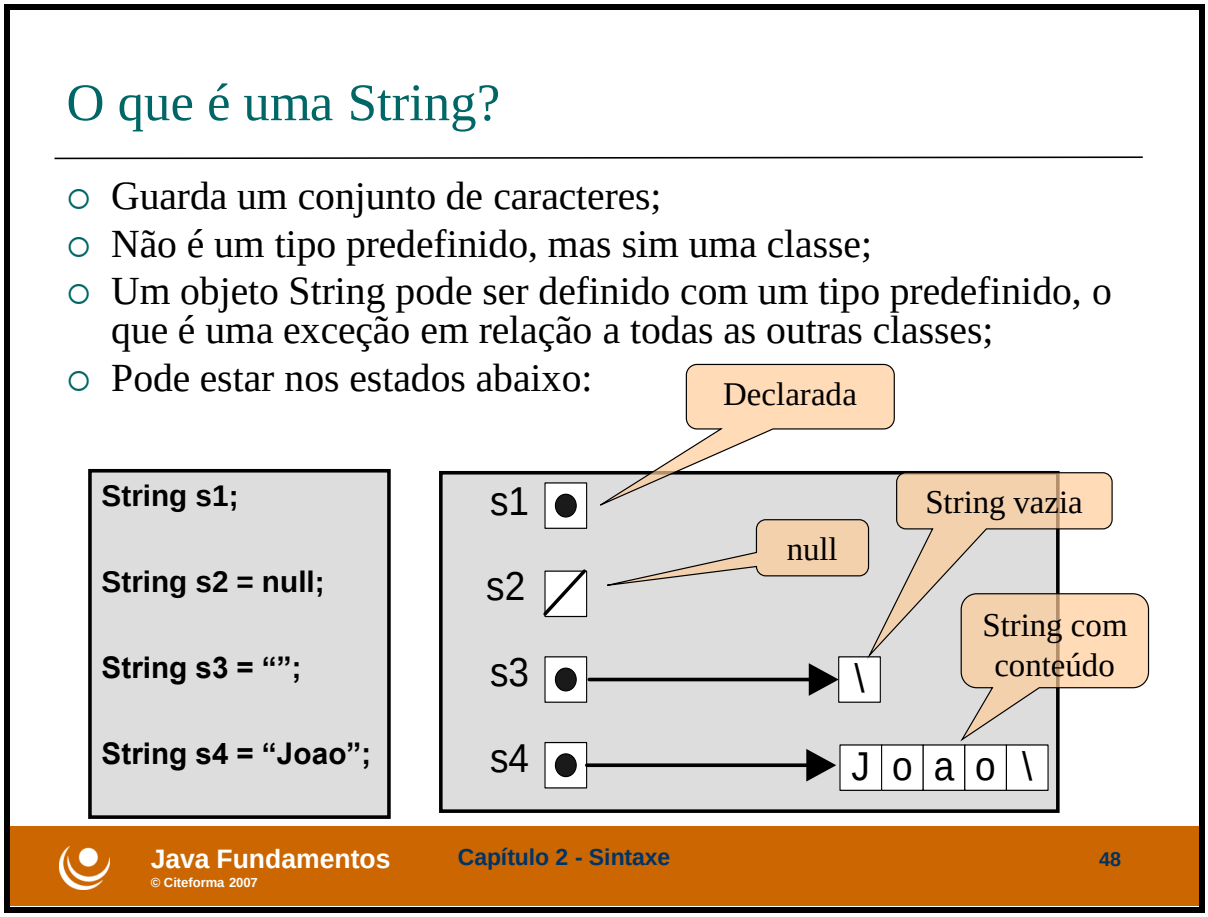
A instrução return força a saída da função teste(), regressando à função main(). Este salto é ainda maior que o dos dois exemplos anteriores.

Strings



Em Java não existe um tipo predefinido para armazenar cadeias de caracteres, sendo fornecida a classe String. Um objeto desta classe pode ser definido de forma semelhante a um tipo predefinido, o que é uma exceção em relação à criação de objetos de todas as outras classes.

O que é uma String?



Uma String pode estar num dos seguintes estados:

- Declarada
- Nula
- Vazia
- Com conteúdo

Métodos da classe String

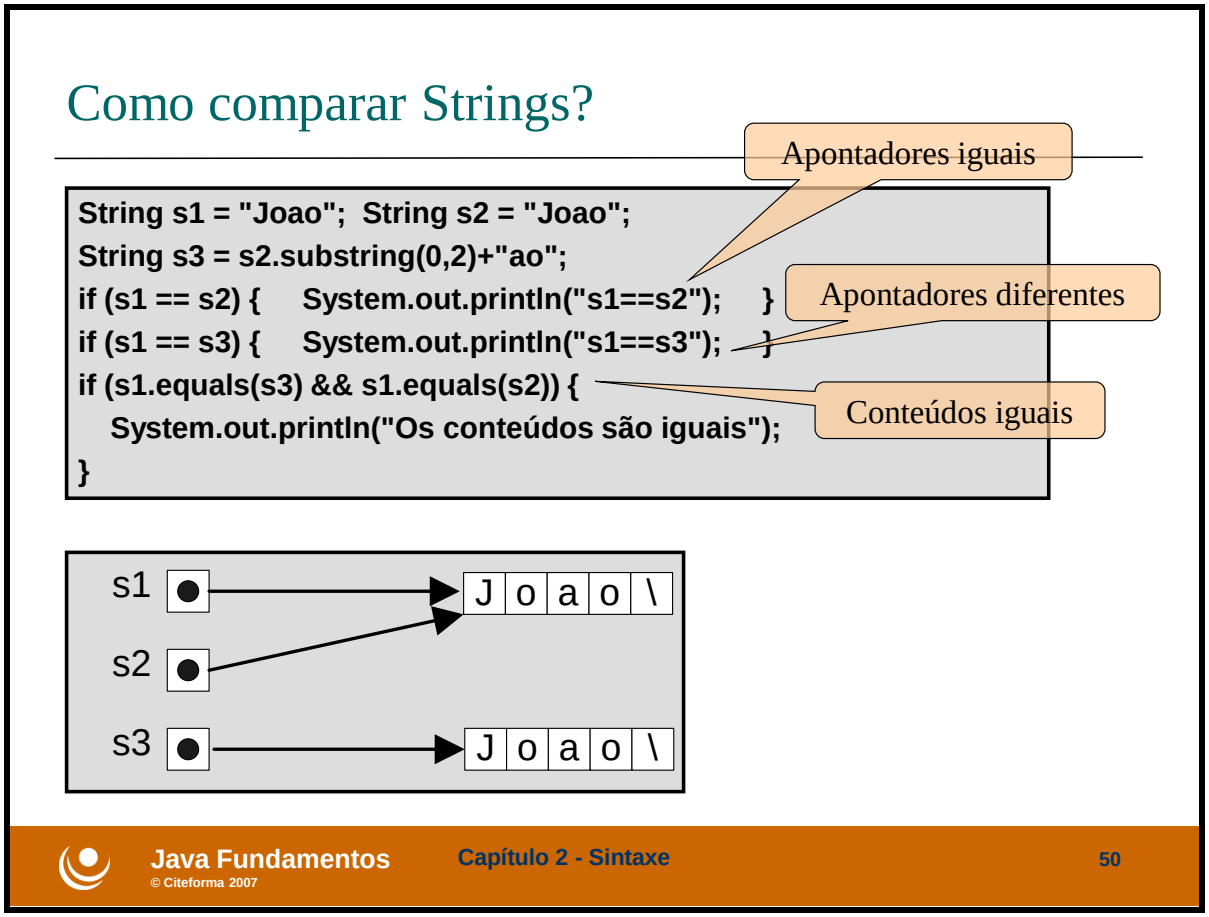
Métodos da classe String

Nome	Descrição
length()	Devolve o número de caracteres que formam a string
substring(n) substring(n,m)	Devolve uma substring
charAt(n)	Devolve o carácter que está na posição n da string original
equals() equalsIgnoreCase()	Permite comparar o texto de duas Strings
compareTo()	Permite ordenar Strings por ordem alfabética
endsWith()	Valida se a string termina com uma sequência de caracteres
indexOf()	Pesquisa um carácter ou uma substring dentro de outra
replace()	Substitui uma substring por outra dentro de uma String maior.
toLowerCase() toUpperCase()	Converte os caracteres da string em minúsculas ou maiúsculas
trim()	Retira os espaços em branco à esquerda e à direita

Java Fundamentos Capítulo 2 - Sintaxe 49

Alguns dos métodos da classe String são descritos no quadro acima.

Como comparar Strings?



Os objetos do tipo String são imutáveis o que significa que uma vez criados não podem ser alterados. Cada vez que uma String é modificada, por exemplo com s = s + “aaa”, é criada uma nova String e atribuída ao apontador s, sendo a String inicial destruída pelo “*garbage collector*”.

O operador de comparação == pode dar resultados inesperados quando usado com Strings.

Para reduzir a ocupação de memória, sempre que o compilador deteta que duas Strings têm o mesmo conteúdo, são automaticamente partilhadas pelos respetivos apontadores, como mostra a figura acima. Quando o compilador não consegue detetar que são iguais, como na String s3 do nosso exemplo, então recebe um novo endereço.

Analisando este diagrama compreendemos que a primeira comparação (s1==s2) seja verdadeira, já que os dois apontadores apontam para o mesmo sitio. Isto aconteceu por otimização da máquina virtual Java. No caso de (s1==s3) é falso porque os apontadores apontam para endereços diferentes. No entanto os conteúdos dasSstrings são idênticos.

Para comparar os conteúdos, em vez dos apontadores, devemos usar os métodos **equals()** ou **equalsIgnoreCase()**.

No capítulo 3, no ponto “Utilização do método troca com tipos pré-definidos” abordamos novamente o armazenamento de Strings em memória.

Exercícios sobre Strings

Exercícios sobre Strings

- Exercício 1 – programa que atribui valores iniciais a 3 Strings e compara os respectivos conteúdos. Verificar as optimizações da linguagem.

Exercício 1:

```
package capitulo2;

public class String01 {
    public static void main(String args[]) {
        String s1 = "Joao";
        String s2 = "Joao";
        String s3 = s2.substring(0,2)+"ao";

        System.out.println(s3);
        if (s1 == s2) {
            System.out.println("s1==s2");
        } else {
            System.out.println("s1!=s2");
        }

        if (s1 == s3) {
            System.out.println("s1==s3");
        } else {
            System.out.println("s1!=s3");
        }
```

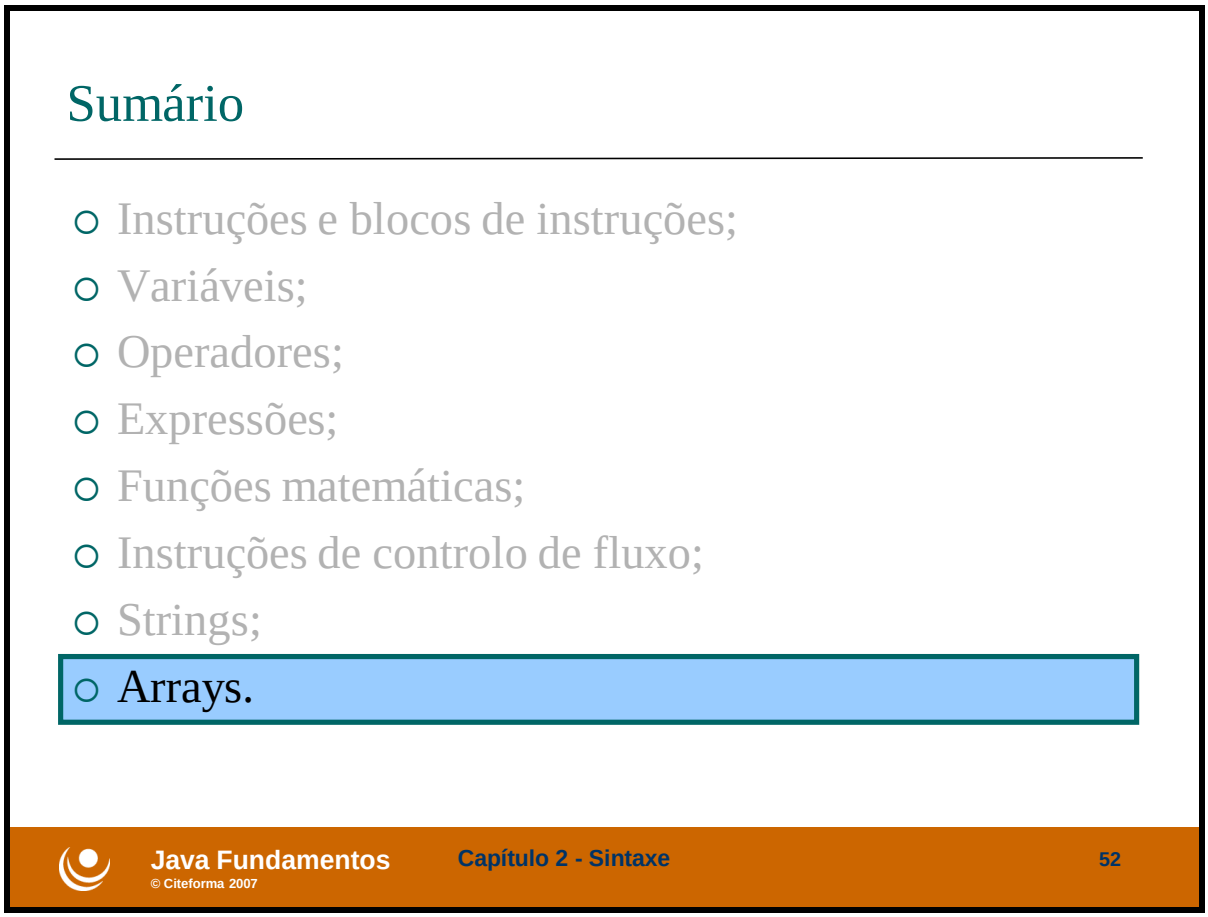
```
        if (s1.equals(s3) && s1.equals(s2)) {
            System.out.println("Os conteúdos são iguais");
        } else {
            System.out.println("Os conteúdos são diferentes");
        }

        // É melhor por "Paulo".equals(s1)
        if (s1.equals("Paulo")) {
            System.out.println("São iguais");
        } else {
            System.out.println("São diferentes");
        }
    }
}
```

A sua execução produz o seguinte resultado:

```
Joao
s1==s2
s1!=s3
Os conteúdos são iguais
São diferentes
```

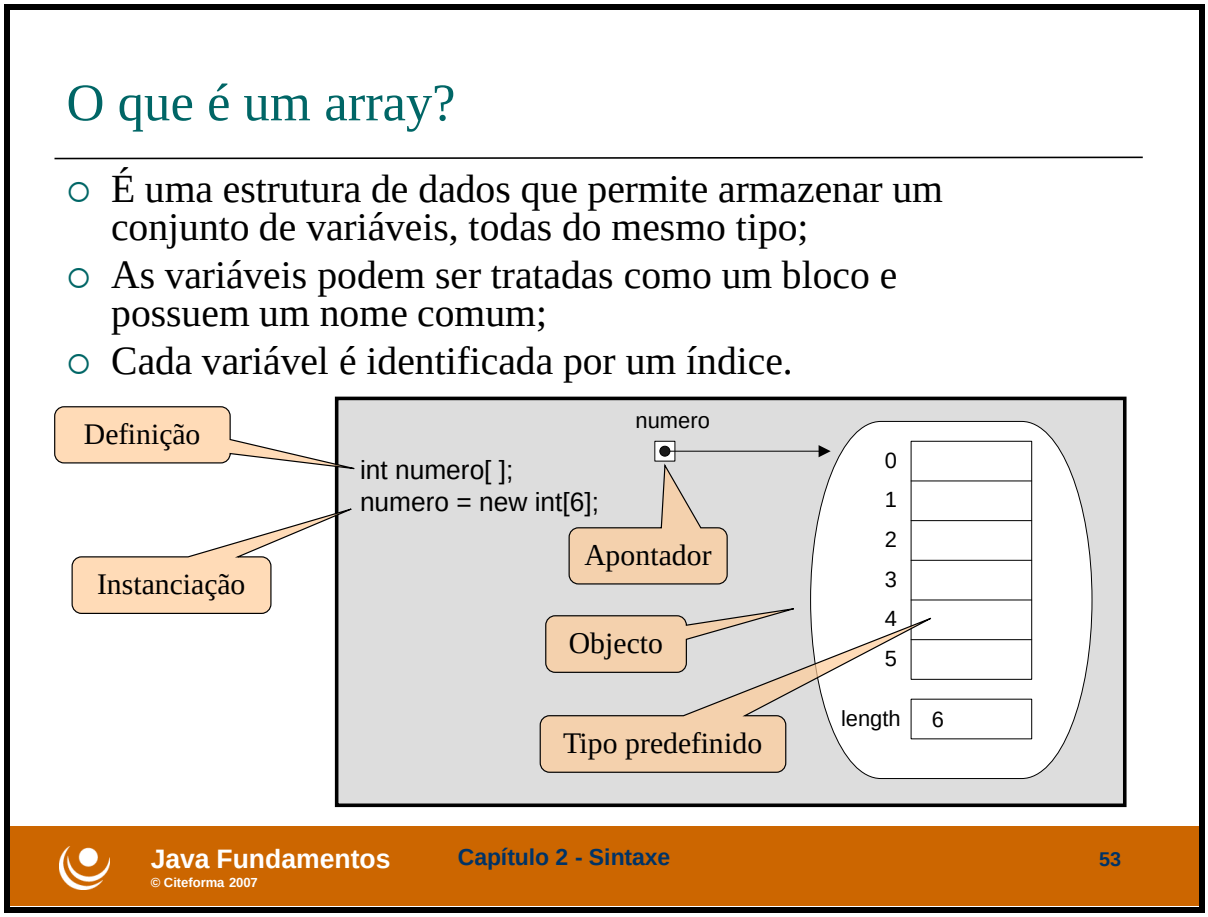
Para compreender o que sucedeu é preciso distinguir entre o apontador (s) e o objeto apontado por esse apontador. A expressão s1==s2 valida se ambos apontadores apontam para o mesmo endereço, mas não compara os conteúdos dos objetos apontados (referenciados).

Arrays



Um array é uma estrutura de dados que permite o armazenamento de um conjunto de variáveis, todas do mesmo tipo, que podem ser tratadas como um bloco e possuem um nome comum. Os elementos dos arrays são identificados por um índice, que é um número inteiro que começa em zero.

O que é um array?



O exemplo acima ilustra como criar e usar um array com uma dimensão, com 6 números inteiros. Os índices vão de 0 a 5 sendo o 0 a referência para o primeiro elemento.

Ao contrário do C, em Java os arrays são automaticamente inicializados com o valor 0. O tipo char é igualmente um tipo inteiro (apenas não tem sinal), razão pela qual os elementos de um array de caracteres são também automaticamente inicializados com zero (\u0000) e não espaço.

Para definir um array com valores pré-definidos indicamos entre chavetas os valores de cada elemento, separados por vírgula:-

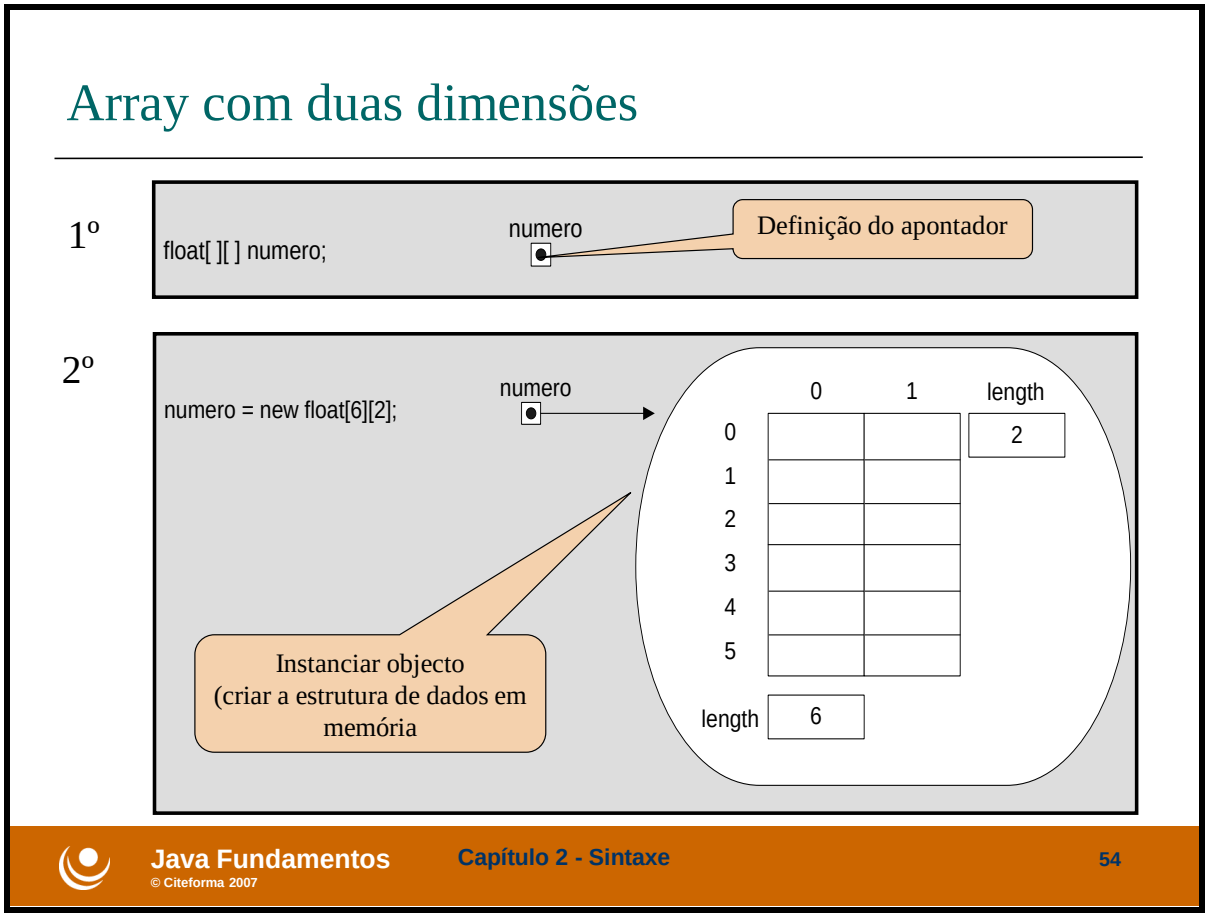
```
int[] numero = { 11, 22, 33, 44, 55, 66 };
```

Note que não precisamos de indicar o comprimento do array (nem devemos), pois o compilador determina-o contando o número de valores iniciais (neste caso 6).

Para indicar os valores iniciais de um array do tipo caracter temos de indicar os seus elementos entre plicas:

```
char[] vogal = { 'a', 'e', 'i', 'o', 'u'};
```

Array com duas dimensões



A figura acima ilustra a criação de um array com duas dimensões (matriz) de números float. Possui 6 linhas e 2 colunas.

Na prática o Java possui apenas arrays de uma dimensão, cujos elementos podem ser outros arrays.

Podemos também definir os seus valores iniciais usando a técnica abaixo:

```
int[][] numero = { {10, 20}, {11, 21}, {12, 22}, {13, 23}, {14, 24}, {15, 25} };
```

Como percorrer um array?

Como percorrer um array?

- Com uma dimensão:

```
for (int i=0;  i<numero.length;  i++) {
   System.out.println("numero[" + i + "]=" + numero[i] );
}
```

- Com duas dimensões:

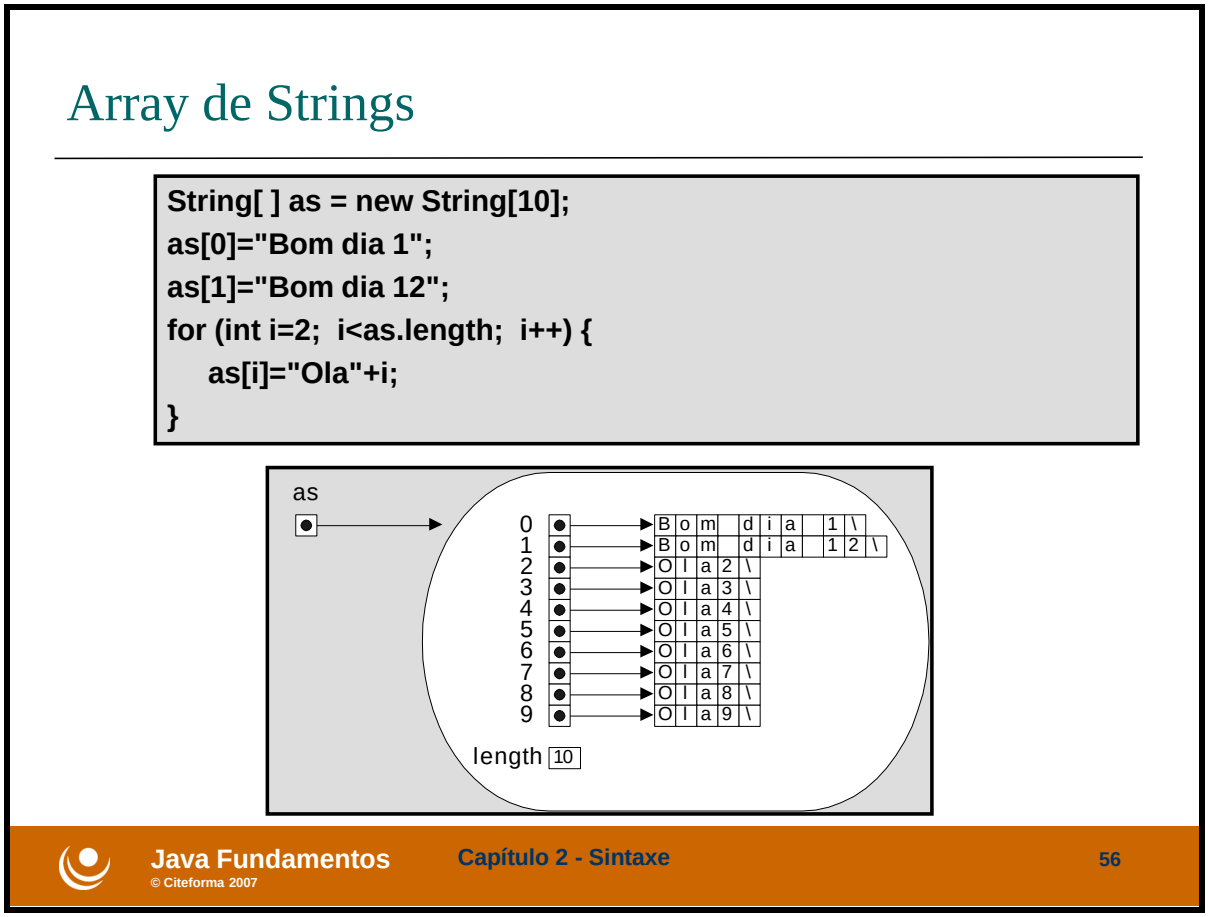
```
for (int i=0; i<numero.length; i++)    {
    for (int j=0; j<numero[0].length; j++) {
        System.out.print("numero["+i+"]["+j+"]="+numero[i][j]+"  \t");
    }
    System.out.println(" ");
}
```

Java Fundamentos Capítulo 2 - Sintaxe 55
© Citeforma 2007

O ciclo for é uma instrução muito conveniente para percorrer arrays, pois permite inicializar o índice, definir a condição de paragem e incrementar o índice, tudo numa só linha.

Para percorrer um array necessitamos de um número de índices igual à sua dimensão. No caso de um array bidimensional (matriz) precisamos de um índice para indicar a linha e outro para indicar a coluna dentro dessa linha.

Array de Strings



Apesar de String não ser um tipo primitivo, a linguagem Java permite criar um objetos da classe String indicando diretamente o seu conteúdo entre aspas. Assim a criação de um array de Strings é idêntica à criação de um array de um tipo primitivo.

Podemos definir os seus valores iniciais usando a técnica abaixo:

```
String[] nome = { "João", "Pedro", "Manuel", "José" };
```

Exercícios sobre arrays

Exercícios sobre arrays

- Exercício 1 – programa que cria e percorre um array com uma dimensão;
- Exercício 2 – programa que cria e percorre um array com duas dimensões;
- Exercício 3 – programa que cria e percorre um array de Strings;
- Exercício 4 – programa que lê os parâmetros importados via linha de comando.

Java Fundamentos Capítulo 2 - Sintaxe 57

Exercício 1

Criar e percorrer um array com uma dimensão:

```
/* Criacao de array de integer;
 * Reparar na variavel i declarada no for e na propriedade ARRAY.length */
package capitulo2;

public class Array01 {
    public static void main(String[] args)     {
        final int MAX=6;
        int[] numero = new int[MAX];
        numero[0]=3;
        numero[1]=4;
        numero[2]=0;
        numero[3]=-3;
        numero[4]=1;
        numero[5]=5;
        System.out.println("Array com "+MAX+" posicoes");
        for (int i=0; i<numero.length; i++) {
            System.out.println("numero["+i+"]="+numero[i]);
        }
    }
```

```
}
```

A sua execução produz o seguinte resultado:

```
Array com 6 posicoes
numero[0]=3
numero[1]=4
numero[2]=0
numero[3]=-3
numero[4]=1
numero[5]=5
```

Exercício 2

Criar e manipular um array de duas dimensões com números float:

```
/* Criacao de array de float com duas dimensoes;
 * Repare na variavel i, declarada no for e na propriedade
 *   ARRAY.length com duas dimensões                          */
package capitulo2;

public class Array02  {
    public static void main(String[] args)  {
        final int LINHA=6;
        final int COLUNA=2;
        float[][] numero=new float[LINHA][COLUNA];
        numero[0][0]= 1.0f;
        numero[0][1]= 2.0f;
        numero[1][0]= 3.0f;
        numero[1][1]= 4.0f;
        numero[2][0]= 5.0f;
        numero[2][1]= 6.0f;
        numero[3][0]= 7.0f;
        numero[3][1]= 8.0f;
        numero[4][0]= 9.0f;
        numero[4][1]=10.0f;
        numero[5][0]=11.0f;
        numero[5][1]=12.0f;
        System.out.println("Array-->numero["+LINHA+"]["+COLUNA+"]");
        for (int i=0; i<numero.length; i++)     {
            for (int j=0; j<numero[0].length; j++) {
                System.out.print("numero["+i+"]["+j+"]="+numero[i][j]+"  \t");
            }
            System.out.println(" ");
        }
    }
}
```

A sua execução produz o seguinte resultado:

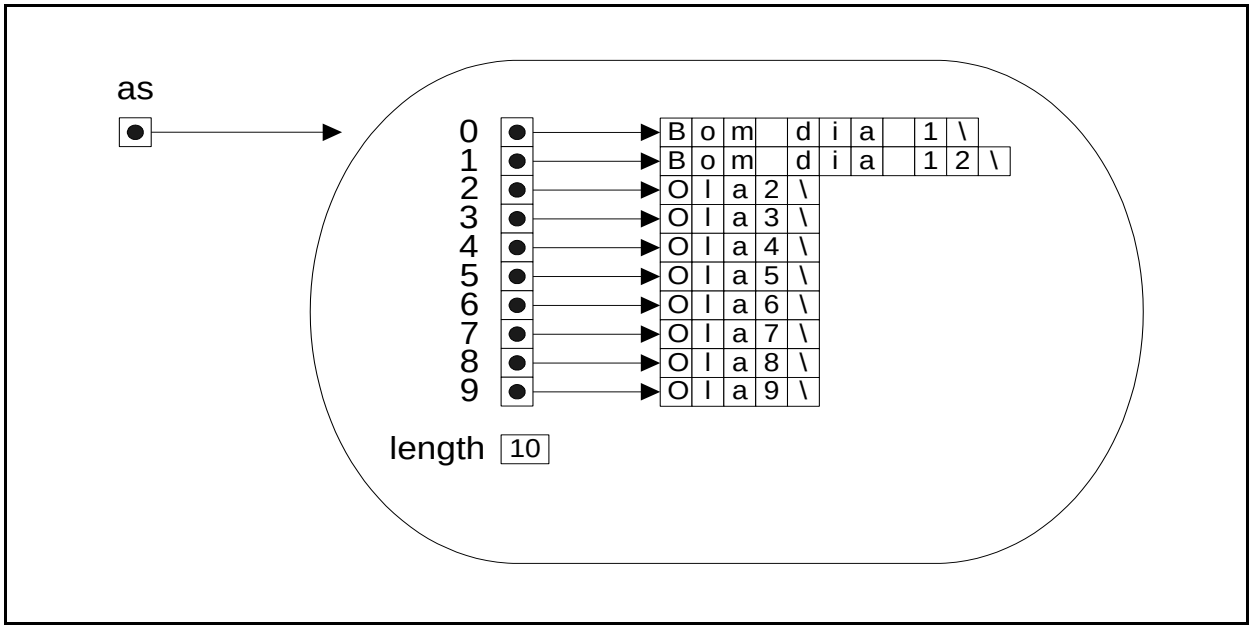
```
Array-->numero[6][2]
numero[0][0]=1.0    numero[0][1]=2.0
numero[1][0]=3.0    numero[1][1]=4.0
numero[2][0]=5.0    numero[2][1]=6.0
numero[3][0]=7.0    numero[3][1]=8.0
numero[4][0]=9.0    numero[4][1]=10.0
numero[5][0]=11.0   numero[5][1]=12.0
```

Exercício 3

Criar e percorrer um array de Strings:

```
/* Criacao de um array de strings.
 * Repare no comprimento variavel das Strings. */
package capitulo2;

public class ArrayString01 {
    public static void main(String[] args) {
        String[] as = new String[10];
        as[0]="Bom dia 1";
        as[1]="Bom dia 12";
        for (int i=2; i<as.length; i++) {
            as[i]="Ola"+i;
        }
        for (int i=0; i<as.length; i++) {
            System.out.println("as["+i+"]="+as[i]+"="+as[i].length());
        }
    }
}
```

O array criado neste programa tem o seguinte aspeto:



A execução deste programa produz o seguinte resultado:

```
as[0]=Bom dia 1=9
as[1]=Bom dia 12=10
as[2]=Ola2=4
as[3]=Ola3=4
as[4]=Ola4=4
as[5]=Ola5=4
as[6]=Ola6=4
as[7]=Ola7=4
as[8]=Ola8=4
as[9]=Ola9=4
```

Exercício 4

Ler os parâmetros importados da linha de comando:

```
package capitulo2;

public class ArrayString02 {
    public static void main(String args[]) {
        for (int i=0; i<args.length; i++) {
            System.out.println("args["+i+"]="+args[i]);
        }
    }
}
```

Para testar o programa podemos chamá-lo na linha de comando usando os parâmetros abaixo:-

```
java capitulo2.ArrayString02 aa bb cc dd
args[0]=aa
args[1]=bb
args[2]=cc
args[3]=dd
```

Para testar este programa no NetBeans é necessário:-

- Ir às propriedades do projeto e, nos parâmetros de execução (RUN time parameters), definir a classe acima como “main class”;
- Adicionar os parâmetros de entrada (aa bb cc dd) ao campo “Command line Arguments”;
- Executar o projeto.

Sumário

- Instruções e blocos de instruções;
- Variáveis;
- Operadores;
- Expressões;
- Funções matemáticas;
- Instruções de controlo de fluxo;
- Strings;
- Arrays.