

# **Estruturas de Programação JAVA**

**Por  
Gilberto Hiragi  
Março/2006**

## **Estruturas de Programação em Java**

Utilizaremos aplicações do tipo *console* para demonstrar as estruturas fundamentais que envolvem a programação em Java. A entrada de dados será feita através da linha de comando e estaremos desta forma fazendo uso da string de argumentos recebida pelo método *main*, já a saída será através do console utilizando o método *System.out.println*.

Java é similar a C++ em muitos aspectos e aqueles que tem um bom conhecimento em C ou C++ tem uma vantagem inicial considerável, principalmente em relação às estruturas fundamentais.

O primeiro detalhe a ser observado é que Java é sensível a maiúsculas e minúsculas, abaixo temos exemplos de identificadores distintos dentro de Java:

Numero, numero e NUMERO

Todos os identificadores acima são diferentes, podendo apontar para valores distintos, mas o principal é estar atento para não se enganar com relação à caixa alta e baixa durante a codificação. Veja que isso é válido para todos os objetos: nome de variáveis, métodos, classes entre outros.

Um outro detalhe importante é que o nome da classe pública deve ser o mesmo do nome do arquivo onde ela esta sendo definida, portanto inclusive aqui deve ser respeitada a regra das letras.

Abaixo temos um programa simples:

### **FirstPrg.java**

```
package estruturas;  
// Nosso primeiro programa  
public class FirstPrg{  
    public static void main(String[] args){  
        System.out.println ("2 + 2 = " + (2+2));  
    }  
}
```

O primeiro detalhe aqui é relativo à declaração de uma classe:

*public class FirstPrg*

Esta classe tem escopo global (*public*) e é regra em java colocarmos primeiro o tipo e depois o identificador, por isso primeiro *class* depois o identificador *FirstPrg*, que é o nome da classe.

*// Nosso primeiro programa*

Note que a linha anterior é ignorada pelo compilador (*javac*), pois (*//*) representa que a seguir vem apenas comentário, outras forma de comentários são (*/\** e *\*/* para mais de uma linha), ou (*/\*\** e *\*/* para mais de linha e utilização de *javadoc*).

Na maioria dos nossos exemplos estaremos utilizando (*//*) para comentários.

Outra peça importante na codificação aqui é o bloco de comandos, os blocos são delimitados através chaves (*{}*), conseguimos agrupar vários comandos através das chaves, delimitando o escopo da classe, métodos e estruturas condicionais.

No *FirstPrg.java* temos dois blocos o principal que delimita a nossa classe *FirstPrg* e um outro bloco interno ao bloco da classe, que delimita o escopo do método *main*, portanto o método *main* pertence a classe *FirstProg*.

Regra de nomenclatura: Vale colocar aqui que uma prática comum de codificação em Java é colocar a primeira letra do nome de uma classe em maiúsculo e em nomes compostos ir fazendo o mesmo para cada primeira letra.

Um método (função) tem a capacidade de receber e se necessário devolver valores. No nosso exemplo temos:

```
public static void main(String[] args)
```

Esta declaração esta nos dizendo que o método é público, portanto se alguém fosse usar nossa classe poderia acessar *main*, já que a mesma tem acesso livre. O método do tipo *static* não precisa ser instanciado para ser utilizado, isso é útil quando temos apenas uma classe isolada, como é o caso, já que podemos executar diretamente a classe, não sendo necessário criar um objeto a partir da classe para usufruir sua funcionalidade.

O método *main* é padrão, ou seja, já tem embutido um significado, que significa para uma aplicação o método de *startup*, aquele em que a execução é iniciada, o interpretador *java* irá procurar por um *main* dentro da classe e quando encontrá-la irá começar a executar a partir da sua primeira linha do bloco.

“*main*” é por definição um método que não retorna valor, e *void* esta indicando isso, portanto resumindo *main* é pública, estática e não retorna valor, porém pode receber valor, isso é indicado por *String[] args* neste ponto é dito que parâmetros recebidos através da linha de comando serão armazenados em *args*, que é um array de *String*. Por exemplo, se chamarmos *FirstPrg* como abaixo:

```
java FirstPrg java 1 4
```

Teremos em *args* a seguinte estrutura armazenada:

Java	1	4
(0)	(1)	(2)

Portanto: *args[0]*=="java", *args[1]*=="1" e *args[2]*=="4"

Os arrays em java são definidos pelo uso de (`[]`) e o primeiro elemento é o zero, portanto o último é numerado como N-1, quando N é o número de elementos do array.

A linha `System.out.println ("2 + 2= " + (2+2));` demonstra como faremos para mostrar algo no console, o método `println` esta dentro de `out`, que esta dentro de `System`, lembrando a hierarquia de linguagens que trabalham com objetos.

Um outro detalhe aqui é que *Strings* são delimitadas por aspas (`"`), portanto, o que fica entre aspas representa um texto literal que não será modificado.

Já o que esta fora de aspas é processado e pode sofrer modificação, neste caso `+ (2+2)` esta fora das aspas, analisando `(2+2)`, que é processado primeiro, (já que há precedência pela existência dos parênteses) é calculado e o resultado é 4, o `System.out.println` se encarrega de compatibilizar os tipos e fazer a concatenação sugerida pelo primeiro `+` fora das aspas.

Esta concatenação entre um número e uma String resulta em String, em Java o tipo String é sempre mais forte em uma expressão, portanto:

### **String + Qualquer tipo = String**

Um último lembrete é que os comandos devem ser finalizados por dois pontos (`;`), para Java não importa se um comando ocupa mais de uma linha, pois quem delimita o fim de uma instrução sempre é o (`;`). Não há ponto ao final de um arquivo de codificação como em Delphi.

A saída deste programa será:

`2 + 2 = 4`

Nada de espantoso, porém os mecanismos básicos para definir uma classe e um método foram destrinchados, além disso, já reconhecemos array e strings, além de conhecer os mecanismos básicos para obter informações através da linha de comando e imprimir um valor para o console.

Abaixo temos mais um exemplo, este programa irá pegar dois valores via teclado e imprimir a soma destes valores:

`Sum2Numbers.java`

```
package estruturas;
import javax.swing.*;
public class Sum2Numbers{
    public static void main(String[] args){
        int first, second;
        first=Integer.parseInt(JOptionPane.showInputDialog("Primeiro Número?"));
        second=Integer.parseInt(JOptionPane.showInputDialog("Primeiro Número?"));
        System.out.println (first + " somado com " + second +
```

```
    " é igual a " + (first+second));  
    System.exit(0);  
}  
}
```

Passamos a explicação das novas características do programa *Sum2Numbers*, o primeiro detalhe é que para executar este programa devemos fornecer dois número através de caixas de diálogos. Para executar usando NetBeans, crie um pacote chamado estruturas e dentro dele uma classe simples *Sum2Numbers* com o código anterior, use Shift+F6 no editor para rodar a classe.

Temos como novidade a declaração de variáveis temos duas variáveis *First* e *Second*, sendo declaradas como inteiras na linha:

```
int first, second;
```

A sintaxe para declaração de variáveis é da seguinte forma:

```
<TIPO> <nome_variável_1> [, ..., nome_variável_n];
```

Utilizaremos esta forma para declarar variáveis, arrays e objetos. Uma prática comum é declarar as variáveis no início de bloco de comandos, procuraremos seguir esta prática. Como Java é uma linguagem fortemente tipada é necessário declarar uma variável antes de utilizá-la.

Nas linhas abaixo estamos demonstrando como atribuir valores a variável, utilizamos para isso o igual simples (=), diferentemente de Delphi que usa (:=), também estamos fazendo uso de caixas de diálogo:

```
first=Integer.parseInt(JOptionPane.showInputDialog("Primeiro Número?"));  
second=Integer.parseInt(JOptionPane.showInputDialog("Primeiro Número?"));
```

Do lado esquerdo fica o identificador (variável, objeto, etc) que irá receber o valor da expressão que esta do lado direito, portanto do lado esquerdo temos um e somente um identificador, já do lado esquerdo podemos ter expressões mais complexas. A sintaxe é:

```
<IDENTIFICADOR> = <EXPRESSÃO>;
```

No caso do exemplo anterior a expressão *Integer.parseInt(JOptionPane.showInputDialog("Primeiro Número?"))* pega o valor que vem da caixa de diálogo transforma este dado String em inteiro e então este valor inteiro é atribuído a *first*, o mesmo foi feito para o segundo parâmetro.

Aqui cabe um detalhe, *Integer* é uma classe básica que contém o método estático *parseInt*, este método é capaz de transformar um valor String em Inteiro, portanto é um método de conversão de dados.

É interessante o aluno treinar questões tais como concatenação na saída de console (*System.out.println*) e também entender o porque da conversão para inteiro, abaixo exemplificamos:

System.out.println("2"+"2"); gera 22 no console  
System.out.println(2+2); gera 4 no console

Ou seja, somar strings é a operação de concatenação, já somar números é a operação usual da matemática.

Como o que vem dos diálogos é uma String, não teria como fazer a operação de soma usual, sem antes transformar os elementos Strings em números inteiros (int).

Em Java há oito tipos básicos de dados, acompanhe na tabela abaixo:

Tipo	Ocupa (Bytes)	Faixa de valores
byte	1	-128 a 127
short	2	-32.768 a 32.767
int	4	-2.147.483.648 a 2.147.483.647
long	8	-9x10e18 a 9x10e18
float	4	-3x10e38 a 3x10e38 (com 6 dígitos significativos)
double	8	-1,7x10e308 a 1,7x10e308 (com 15 dígitos significativos)
char	2	Utiliza padrão unicode, tem como subconjunto o ASCII
boolean	1	True ou False

O aluno deverá treinar a declaração e utilização dos tipos primitivos para ganhar mais intimidade com a linguagem.

Abaixo temos um programa que multiplica dois números reais:

Mult2Numbers.java

```
package estruturas;  
import javax.swing.JOptionPane;  
  
public class Mult2Numbers{  
    public static void main(String[] args){  
        float First, Second;  
        First =Float.parseFloat(JOptionPane.showInputDialog("Qual o primeiro  
real?"));  
        Second=Float.parseFloat(JOptionPane.showInputDialog("Qual o segundo  
real?"));  
        System.out.println (First + " Multiplicado por " + Second +  
        " é igual a " + (First*Second));  
        System.exit(0);  
    }  
}
```

Note que no exemplo anterior utilizamos a classe *Float* e o método *parseFloat* para converter a string em um número real simples, algo similar acontece com os outros tipos por exemplo existe *Double.parseDouble* para conversão de string para double, note que o tipo de dados é todo minúsculo (float) e a classe que contém o método tem a primeira maiúscula (Float).

Alguns detalhes interessantes de conversão de dados numéricos, falam sobre a hierarquia dos tipos, veja as regras abaixo para expressões matemáticas:

- Se um operando for *double* então qualquer outro é convertido para *double*.
- Se um operando for *float* então o outro, não sendo *double* é convertido para *float*.
- E assim segue com long, int, short e byte.

Porém nem sempre queremos partir para conversões para os tipos mais abrangentes, às vezes, queremos restringir o dado, ou seja, converter para um tipo mais limitado, por exemplo, de real para inteiro. Nestes casos fazemos uso de conversões explícitas do tipo (cast), vale lembrar que neste tipo de conversão pode haver perda de informação.

Exemplos:

```
double num = 7.7;  
int numInt = (int) num;
```

A variável *numInt* terá o valor 7, desprezando portanto a parte fracionária.

```
double num = 7.7;  
int numInt = (int) Math.round(num);
```

Neste momento *numInt* recebe o valor 8, porque *round* da classe *Math*, faz arredondamento.

Uma outra forma de declaração é a constante, como usual constantes não podem ter o valor alterado durante o processamento do programa, exemplo:

```
final double salario = 3767.54;
```

Para finalizar esta seção de conversão de dados, gostaríamos de apresentar o método *toString*, este método existe nas classes numéricas e é capaz de transformar o número no String correspondente, portanto o *toString* faz o inverso dos métodos *parse*, exemplo:

```
float num=10.9;  
String str=Float.toString(num);
```

Para completarmos a seção de tipos numéricos, passamos uma lista de operadores que podem ser úteis na manipulação destes tipos:

Operadores aritméticos: + - \* / % são usados para somar, subtrair, multiplicar, dividir e resto da divisão.

Para fazer exponenciação pode ser utilizado o método *pow*, que trabalha com o tipo *double*, exemplo:

```
double num=Math.pow(2,3);
```

Num será igual à 8, ou seja 2 elevado a 3.

Como em C, java aceita os operadores de incremento(++ ) e decremento (- -), exemplo:

```
int contador=0;  
contador++; // contador fica igual a 1
```

Os operadores lógicos também estão presentes em java, inclusive com uma gama extensa de possibilidades, inclusive em termos de métodos, estes operadores devolvem valores do tipo booleano, os principais operadores lógicos comparativos são:

- Igualdade (==), exemplos: (4 == 7) é falso, (5 == 5) é verdadeiro.
- Diferente (!=), exemplos: (4 != 7) é verdadeiro, (5 != 5) é falso.
- Além desses temos os operadores < (menor que), > (maior que), <= (menor ou igual a) e >= (maior ou igual a)

Para completar os operadores lógicos temos os conectivos (&&) e lógico, (||) ou lógico e (!) a negação.

Exemplos:

```
(1=1) && (1!=1) é falso  
(1=1) || (1!=1) é verdadeiro  
!(1=1) é falso
```

Para modificar a precedência é possível fazer uso dos parênteses, a precedência segue a seguinte lista do que é feito primeira para o último a ser avaliado:

```
()  
!, ++, --  
*, /, %  
+ -  
<, <=, >, >=  
==, !=  
&&  
||  
=
```



## char

O char é um tipo simples que representa um caractere da tabela Unicode, ele ocupa dois bytes de espaço e no caso do Java esta de acordo com os padrões internacionais que inclusive contemplam línguas orientais como japonês e chinês.

O tipo char em termos de processamento pode ser considerado um *int*, já que a conversão implícita e de *cast* pode ser feita.

char -> int (Retorna o código número da posição na tabela Unicode)  
int -> char (Retorna o caractere que estiver na posição da tabela Unicode)

Exemplos:

```
int codUni=65;  
char caract=(char) codUni; // caract=='A'
```

Acompanhe o programa a seguir que mostra as primeiras três letras do alfabeto através de conversões de inteiro em char:

### ConvChar.java

```
package estruturas;
```

```
public class ConvChar{  
    public static void main(String[] args){  
        int a, b, c;  
        char c1, c2, c3;  
        a=97; b=98; c=99;  
        c1=(char) a;  
        c2=(char) b;  
        c3=(char) c;  
        System.out.println(c1);  
        System.out.println(c2);  
        System.out.println(c3);  
        System.exit(0);  
    }  
}
```

No próximo exemplo temos um programa que recebe um número inteiro e depois mostra qual é o caractere que corresponde a este número na tabela Unicode:

### QualCaractere.java

```
package estruturas;
import javax.swing.JOptionPane;

public class QualCaractere{
    public static void main(String[] args){
        int cod;
        char caractere;

        cod=Integer.parseInt(JOptionPane.showInputDialog("Qual o código?"));
        caractere=(char)cod;

        System.out.println("O código é " + cod);
        System.out.println("O caractere correspondente é " + caractere);
        System.exit(0);
    }
}
```

## **Strings**

São seqüências de caracteres e tem funcionalidade similar aos das linguagens tradicionais, no Java o delimitador de Strings é as aspas ("").

Para concatenar uma String a outra basta usar o operador (+), exemplo:

```
String frase = "Esta é uma ";
frase = frase + "frase";
```

String em java não é um tipo primitivo, uma String qualquer é uma instância da classe String e como tal tem alguns métodos interessantes, tais:

```
substring:
String s="Linguagem";
String d=s.substring(0,6); // d será igual à "Lingua"
String e=s.substring(3,6); // e será igual à "gua"
```

```
Tamanho de uma string:
String s="Servlet";
int n=s.length(); // n será igual à 7
```

Retornando um determinado caractere de uma String:

```
String s="Java Server Pages";  
char c1=s.charAt(1); // c1 será igual à 'a'  
char c2=s.charAt(5); // c2 será igual à 'S'
```

\* No java os índices sempre iniciam em zero.

Transformar em maiúscula e minúsculas:

```
String s="TomCat";  
String u=s.toUpperCase(); // u será igual à "TOMCAT"  
String l=s.toLowerCase(); // l será igual à "tomcat"
```

Comparação entre Strings:

Não é aconselhável usar (==) para comparar Strings, existem dois métodos adequados para isso equals e equalsIgnoreCase, como em:

```
String s="Java";  
boolean t1=s.equals("java"); // t1 será igual à falso  
boolean t2=s.equals("Java"); // t2 será igual à verdadeiro  
boolean t3=s.equalsIgnoreCase("java"); // t3 será igual à verdadeiro
```

Trocando caractere de uma String:

```
String s="Jakarta";  
String x=s.replace('a','x'); // x será igual à "Jxkxrtx"
```

Trocando uma substring de uma String:

```
String s="Java Server Pages";  
String x=s.replaceAll("a","ah"); // x será igual à "Jahvah Server Pahges"
```

Eliminando espaços em branco:

```
String s=" Tem espaço de mais ";  
String x=s.trim(); // x será igual à "Tem espaço de mais"
```

Existem outros métodos interessantes na classe String, é válido acessar a documentação da Sun para esta classe.

Para finalizar esta parte relativa a dados, gostaríamos de listar mais três métodos, encarregados de formatar números, estes métodos fazem parte da classe `NumberFormat`:

```
getNumberInstance()
getCurrencyInstance()
getPercentInstance(), exemplo:
```

Veja que no exemplo a seguir temos que instanciar um objeto do tipo *NumberFormat*, para então poder utilizar os métodos acima citados.

```
Moeda.java
import java.text.*;
public class Moeda{
    public static void main(String[] args){
        double sal=2876.87;
        NumberFormat nf=NumberFormat.getCurrencyInstance();
        System.out.println(nf.format(sal));
    }
}
```

### **Decisão (estrutura condicional)**

Qualquer programa inteligente utiliza ao menos alguma estrutura mais complexa que a sequencial, no caso do Java a estrutura fundamental para desvio de execução é a estrutura *if*.

A seguir temos a sintaxe básica de *if*:

```
if (expressão booleana){
    <bloco-se-verdadeiro>
}
else{
    <bloco-se-falso>
}
```

Note que a expressão booleana pode ser qualquer expressão que resulte em verdadeiro ou falso, não importando a quantidade de parâmetros na fórmula de comparação. Veja o exemplo a seguir:

```
Comparar.java
package estruturas;
import javax.swing.JOptionPane;

public class Comparar{
    public static void main (String[] args){
        int n1=Integer.parseInt(JOptionPane.showInputDialog("Qual o primeiro inteiro?"));
    }
}
```

```
int n2=Integer.parseInt(JOptionPane.showInputDialog("Qual o segundo inteiro?"));

if (n1>n2)
    System.out.println (n1 + " é maior que " + n2);
else if (n2>n1)
    System.out.println (n2 + " é maior que " + n1);
else
    System.out.println (n1 + " é igual a " + n2);

System.exit(0);
}
}
```

Como é usual nas linguagens de programação atual, o *if* é a estrutura básica de decisão, ele executa ou não um bloco de comandos (ou comando) conforme o valor booleano da condição.

Note que acima não foi necessário utilizar chaves {}, visto que para cada resultado de condição é associado apenas um comando, ou seja, a chamada do método System.out.println.

O exemplo a seguir diz o tamanho de uma string de entrada e julga se ela ultrapassa o limite de quinze caracteres, isso pode ser usado na validação de dados, por exemplo, em um banco de dados o nome não pode conter mais que “n” caracteres.

#### ValNome.java

```
package estruturas;
import javax.swing.*.*;

public class ValNome{
    public static void main (String[] args){
        String nome;

        nome=JOptionPane.showInputDialog("Qual o nome?");

        System.out.println ("O nome " + nome + " tem " + nome.length()
            + " caracteres ");

        if (nome.length()>15)
            System.out.println ("Foi ultrapasso o valor máximo" +
                " de 15 caracteres");

        System.exit(0);
    }
}
```

## Laços

Outra estrutura importante em java é a estrutura de repetição, normalmente com variável de controle numérica, ou seja, o *for*. A sintaxe básica de *for* é:

```
for (<inicialização>; <condição>; <modificador>){  
    <bloco-de-comandos>  
}
```

O laço *for* pode ser usado para implementar as mais diversas contagens e varreduras que tenham uma determinada sequência numérica, o diferencial aqui é justamente que o passo, incremento ou decremento é feito pelo programador, diferentemente de Delphi por exemplo, que só permite incrementar ou decrementar em uma unidade.

Abaixo temos um programa que imprime JDBC 10 vezes:

JDBC10.java

```
public class JDBC10{  
    public static void main (String[] args){  
        for (short i=0; i<10; i++)  
            System.out.println("JDBC");  
        System.exit(0);  
    }  
}
```

A inicialização pode conter a declaração da variável numérica que servirá de controle para o laço, além disso, se desejar repetir mais de um comando deverá necessário fazer uso das chaves.

Utilizamos o operador (++) para fazer o incremento, mas nota que poderia ser qualquer outro comando de atribuição, como (i=i+1).

O próximo exemplo faz uma contagem de 0 até 10 de 0,1 em 0,1:

ContaReais.java

```
public class ContaReais{  
    public static void main (String[] args){  
        for (float i=0.0F; i<=10.0F; i=i+0.1F)  
            System.out.println("Contando " + i);  
        System.exit(0);  
    }  
}
```

Note que para indicar que um valor é do tipo *float* devemos fazer uso do sufixo (F), no caso do tipo *double*, use o sufixo (D).

Interessante no exemplo anterior é notar a precisão com que é calculada a soma, veja que com *float* você vai aos poucos perdendo a precisão, já com *double* a perda é menor, compare utilizando a variável *i* como *float* e depois como *double*.

O próximo exemplo lista todos os parâmetros que você recebe via *args* do método *main*:

#### ListaArgs.java

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

O detalhe aqui é que usamos o contador para variar o índice do array *args*, além disso, note que a propriedade *length* retorna o número de elementos do vetor.

### **Laços com while**

O laço *while* nunca executa se a condição for falsa logo no início, porém enquanto ela for verdadeira são repetidos os procedimentos que fazem parte do seu escopo. Veja a sintaxe:

```
while (<condição>){
    <bloco-de-comandos>
}
```

O exemplo a seguir sorteia um número de 0 a 100 e então vai pedindo palpites, caso o usuário consiga acertar paramos o programa, senão continuamos pedindo palpites. Cada iteração do laço é contada como tentativa, além disso, devemos fornecer ao usuário se o palpite é maior ou menor que o número sorteado. Veja o código:

#### Palpites.java

```
package estruturas;
import javax.swing.*;

public class Palpites{
    public static void main(String[] args){
        byte sorteado=(byte) (Math.random()*100);
        byte palpite;
        byte tentativas=0;
        boolean acertou=false;
        String LeTeclado;

        while (! acertou){
            try{
                palpite=Byte.parseByte(JOptionPane.showInputDialog("Qual o
palpite?"));
            }
        }
    }
}
```

```
tentativas++;
if (palpite==sorteado){
    acertou=true;
    System.out.println ("*** VOCÊ ACERTOU !!! É mesmo " + palpite +
"***");
}
else if (palpite<sorteado)
    System.out.println ("Palpite é menor que o valor sorteado.");
else if (palpite>sorteado)
    System.out.println ("Palpite é maior que o valor sorteado.");
}
catch (NumberFormatException err){
    System.out.println("Erro na conversão ...");
    System.out.println(err);
}

}
System.out.println("-----");
System.out.println("Você acertou em " + tentativas + " tentativas.");
}
}
```

Necessitamos usar *try*, porque estamos fazendo operação de conversão, é uma maneira mais robusta de tratar possíveis erros no momento da conversão, por exemplo, não é possível converter um caractere “?” por um número, porém como a entrada de dados é liberada o usuário final poderá digitar algo inadequado, resultando em erro e quebra da execução do programa por falha, com o *try* podemos evitar esta queda brusca e então tratar o erro da melhor forma.

Utilizamos o método *random*, da classe *Math*, esta classe *Math* possui alguns métodos interessantes, tais:

*ceil()* – Retorna a parte inteira de um *double*.  
*floor()* – Retorna o menor inteiro de um *double*.  
*min()* – Retorna o menor entre dois números.  
*max()* – Retorna o maior entre dois números.  
*sqrt()* – Retorna a raiz quadrada do número.  
*random()* – Retorna um número *double* entre 0 e 1, qualquer.

A sintaxe básica para estes métodos é:

`x = Math.método(<lista-de-parâmetros>)`, sendo x alguém que possa receber o valor calculado.

Para finalizar as estruturas de repetição básicas do java, iremos tratar de *switch*, esta estrutura é útil quando estamos fazendo decisão encima de um mesmo valor inteiro com várias alternativas, normalmente é mais legível que utilização de *if* aninhado.

Veja o exemplo abaixo que ilustra como esta estrutura pode ser utilizada:



Escolha.java

```
package estruturas;

import javax.swing.*;

public class Escolha{
    public static void main(String[] args){
        byte mes=0;
        String leTeclado;

        while (true){
            try{

                leTeclado=JOptionPane.showInputDialog("Qual o mês?");
                mes =Byte.parseByte(leTeclado);
                if (mes==0){
                    System.out.println ("*** Boa Noite Tchau tchau ***");
                    break;
                }

                switch (mes){
                    case 1 :
                        System.out.println ("Janeiro");
                        break;
                    case 2 :
                        System.out.println ("Fevereiro");
                        break;
                    case 3 :
                        System.out.println ("Março");
                        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 ("Mês inválido");
    }

}

}
catch (NumberFormatException err){
    System.out.println("Erro na conversão ...");
    System.out.println(err);
}
}
}
}
```

No exemplo acima é pedido um número ao usuário, caso ele entre com zero (0) o programa termina, caso contrário ele entra no *switch* e conforme o valor digitado vai imprimir o mês por extenso adequadamente, caso entre com um valor numérico inválido para mês é então impressa a mensagem de aviso “ Mês inválido”.

Note que o *break*, o primeiro do código serve para sair do laço *while*, já os *breaks* que estão dentro do *switch*, servem justamente para sair do *switch*.

Faça um teste eliminando os *breaks* do *switch*, veja como se comporta o programa.

## Arrays

Imagine ter que tratar uma coleção de nomes, digamos 100 nomes através de variáveis simples, tais como nome1, nome2, ....., nome100, com certeza esta não é a melhor forma.

Como na maioria das linguagens, Java, contém o tipo *array*, que pode ser visto como uma coleção de variáveis do mesmo tipo, sendo que cada elemento é acessado através de um índice.

Em Java é possível utilizar *arrays* multidimensionais, o *array* com uma dimensão é normalmente chamado de vetor. Um vetor que já estamos acostumados a encontrar é o *args*, parâmetro recebido através do método *main*, este é um vetor de Strings.

A sintaxe básica para declarar um array é:

```
<tipo-do-dado[]> <nome-do-array> = new <tipo-do-dado[quantidade]>
```

Exemplo:

```
int numeros = new int[10];
String nomes = new [100];
```

Veja o exemplo anterior que a partir do número do mês imprimimos o mês por extenso, ele pode ser reescrito utilizando um vetor:

### EscolhaArray.java

```
package estruturas;
import javax.swing.*;

public class EscolhaArray{
    public static void main(String[] args){
        byte mes=0;
        String leTeclado;
        String[] meses = {"Janeiro", "Fevereiro", "Março", "Abril", "Maio",
                           "Junho", "Julho", "Agosto", "Setembro",
                           "Outubro", "Novembro", "Dezembro"};

        while (true){
            try{
                leTeclado=JOptionPane.showInputDialog("Qual o mês?");
                mes      =Byte.parseByte(leTeclado);

                if (mes==0){
                    System.out.println ("*** Boa Noite Tchau tchau ***");
                    break;
                }

                if (mes>12 || mes<1){
                    System.out.println ("Mês inválido");
                }
                if (mes>0 && mes<13){
                    System.out.println (meses[mes-1]);
                }
            }
            catch (NumberFormatException err){
                System.out.println("Erro na conversão ...");
                System.out.println(err);
            }
        }
    }
}
```

Veja que é possível declarar um *array* de forma literal, ou seja, informando entre chaves ({}), os valores elemento a elemento separados por vírgulas do *array*.

O acesso ao elemento ocorre através do índice que é colocado entre colchetes ([]), sendo que o primeiro elemento é numerado como zero.

Portanto se é declarado um vetor como em `int[] num = new int[10]`, você terá os seguintes elementos: `num[0]`, `num[1]`, ..., `num[9]`.

Um aspecto interessante em java é que um *array* pode ter como elementos, ou seja, o tipo dele pode ser inclusive uma classe, portanto você pode criar um *array* de objetos, tal como em Delphi.

O exemplo a seguir preenche uma matriz com 3 linhas e 3 colunas e então imprime a soma das linhas e a soma das colunas:

Matriz.java

```
package estruturas;
import javax.swing.*;

public class Matriz{
    public static void main(String[] args){
        byte mes=0;
        String leTeclado;
        int[][] matriz = new int[3][3];
        int soma;
        String linha;

        // Recebendo dados ...
        for (byte l=0;l<3;l++){
            for (byte c=0;c<3;c++){
                try{
                    leTeclado=JOptionPane.showInputDialog ("Entre com valor para
linha " + l +
                                " e coluna " + c + " : ");
                    matriz[l][c]=Integer.parseInt(leTeclado);
                }
                catch (NumberFormatException err){
                    System.out.println("Erro na conversão ...");
                    System.out.println(err);
                }
            }
        }

        // Mostrando em forma de matriz
        System.out.println (" *** Matriz de Entrada *** ");
        for (byte l=0;l<3;l++){
            linha="";
            for (byte c=0;c<3;c++){
                linha+=" " + String.valueOf(matriz[l][c]);
            }
            System.out.println(linha);
        }

        // Calculando soma nas linhas
        System.out.println (" *** ");
        for (byte l=0;l<3;l++){
            soma=0;
            for (byte c=0;c<3;c++){
                soma+=matriz[l][c];
            }
            System.out.println("Soma da linha " + l + " é " + soma);
        }
    }
}
```

```
    }

    // Calculando soma nas colunas
    System.out.println ( " *** " );
    for (byte c=0; c<3; c++) {
        soma=0;
        for (byte l=0; l<3; l++) {
            soma+=matriz[l][c];
        }
        System.out.println("Soma da coluna " + c + " é " + soma);
    }

    System.out.println ( "----- FIM " );
}
}
```

Mais uma vez é interessante ressaltar que o índice inicial é zero, neste caso de matriz, temos então para `int[][] matriz = new int[3][3]`, os seguintes elementos:

```
matriz[0][0], matriz[0][1], matriz[0][2]
matriz[1][0], matriz[1][1], matriz[1][2]
matriz[2][0], matriz[2][1], matriz[2][2]
```

Note que você vai criando pares de colchetes ([]) para cada dimensão do *array*.

## Vector

Existe uma classe que substitui com algumas vantagens o arrays. Esta Classe pode ser considerada um array dinâmico, ou seja, o número de elementos é variável ao longo da execução.

No exemplo a seguir iremos mostrar que não há necessidade de saber de antemão a quantidade de elementos para utilizar o Vector, você pode ir adicionando elementos ao Vector, é também possível inserir elementos entre elementos já existentes e fazer a eliminação de elementos já armazenados.

Apesar do Vector ser uma classe mais flexível o seu uso indiscriminado não é recomendável, principalmente por questão de performance, os arrays são mais rápidos.

Em sistemas comerciais esta diferença de velocidade é desconsiderada.

### Nomes.java

```
package estruturas;

import javax.swing.JOptionPane;
import java.util.Vector;

public class Nomes {
    public static void main(String[] args){
        Vector nomes=new Vector();
        String nome="";
        boolean sair=false;

        while (!sair){
            nome=JOptionPane.showInputDialog("Qual o nome? em branco
sai");
            if (nome.equals("")){
                sair=true;
            } else {
                nomes.add(nome);
            }
        }

        nomes.insertElementAt("Novo Primeiro Elemento",0);

        System.out.println("-> Lista de elementos - " + nomes.size());
        for (int i=0; i<nomes.size(); i++){
            System.out.println(nomes.get(i));
        }

        System.out.println("-> Último Elemento");
        System.out.println(nomes.lastElement());

        System.exit(0);
    }
}
```

### **Métodos**

O último tópico deste roteiro trata sobre a criação de métodos para uma classe, note que você pode considerar sem perda de conhecimento que método é similar a uma função como em outras linguagens, a diferença básica esta na associação forte entre classe e métodos, portanto um método sempre esta ligado a sua classe, faz parte da classe.

O Java é mais purista em relação à orientação a objetos que Delphi, portanto não existem “funções” perdidas, cada método esta hierarquicamente ligado a uma classe, portanto você sempre irá acessar um método através:

### **Classe.Método() ou Objeto.Método**

Para os métodos estáticos não é necessário instanciar um objeto para usá-lo, já nos outros casos, a funcionalidade do método só é possível após instanciação, que é a criação de um objeto a partir da sua classe.

Já utilizamos métodos de classes pré-definidas como em:

```
Integer.parseInt()  
String.charAt()  
String.length(), entre outros.
```

Um exercício para o aprendiz é rever os exemplos e procurar onde estamos fazendo uso de métodos.

Podemos criar nossos próprios métodos e já temos feito isso nos nossos exemplos, já que código funcional em Java sempre esta dentro de métodos.

O método *main* que recebe o argumento *String[] args* tem sido utilizado em todos os exemplos, porém podemos dividir, modularizar mais ainda, distribuindo a lógica em outros métodos. Até aqui os exemplos têm sido pequenos, imagine algo maior, com várias tarefas agregadas, cada um com seu escopo natural, neste contexto o uso de métodos se faz quase imprescindível.

O uso de métodos separa e organiza a lógica, além disso, tratar problemas grandes, dividindo os mesmos em problemas menores é uma técnica bastante efetiva.

Um método deve ser codificado dentro da classe a qual pertence, portanto estará entre as chaves da classe. Um método em Java, tal como uma função retorna um valor, mas alternativamente podemos dizer que ele retorna vazio (*void*), sendo portanto somente um procedimento. Vejamos a sintaxe básica para criação de um método:

```
<qualificadores> <tipo-de-retorno> <nome-do-método> ([lista-de-parâmetros]){  
  <bloco-de-comandos>  
}
```

O primeiro detalhe é relativo a <qualificadores> estes podem assumir várias formas, iremos destacar mais a frente este tópico, mas neste momento você deve conhecer:

*public static* – Permite criar um método que pode ser executado por agentes externos, inclusive independente de instanciação.

*private static* – Como o nome sugere este método só é visível dentro da própria classe onde foi definido e poderá ser executado diretamente sem necessidade de instanciação.

Em ambos os casos omitindo-se a palavra *static* estaremos obrigando a instanciação de um objeto para então utilizar o método, um método estático é mais oneroso para o sistema, porém sempre temos algum método estático que inicia o processo de execução.

Com estes dois qualificadores é possível atender a praticamente todos os casos.

O <tipo-de-retorno> é um valor ou objeto que é retornado pelo método após o processamento interno do método, como em uma função matemática, você pode passar parâmetros e ter um valor como resposta. Você declara o <tipo-de-retorno> como um tipo de dados ou uma classe.

O <nome-do-método> segue o padrão de nomenclatura adotado até agora, veja que você poderá executar o método chamado-o pelo nome. Relembrando este nome deverá ter letras minúsculas, a não ser as iniciais das segundas componentes em diante.

A [lista-de-parâmetros] é opcional, mas muito interessante para fazer a interface entre o exterior e a rotina que esta internalizada ao método, ao seja, pela parametrização que você consegue criar métodos genéricos, que atendam a várias situações.

As chaves servirão para agrupar o que faz parte do método, em termos de codificação.

Vejamos um exemplo, onde iremos definir um método chamado *terco*, que calcula um terço do valor que foi passado como parâmetro, veja que continuamos a definir *main* e é por onde tudo se inicia.

UmTerco.java

```
public class UmTerco{
    public static void main(String[] args){
        double num;

        for (num=1; num<10; num++){
            System.out.println ("Um terço de " + num + " é " + terco(num));
        }
        System.out.println ("*** FIM ***");
    }

    // método auxiliar terco (calcula um terço de um parametro double)
    private static double terco (double n){
        return n/3;
    }
}
```

Note que a declaração de parâmetros segue a regra de declaração comum, que é colocar o tipo primeiro, depois o nome do parâmetro.

Outro detalhe é que como estamos dentro da classe que contém o método não precisamos preceder o nome do método com o nome da classe.

No próximo exemplo iremos criar uma função de recebe como parâmetro uma frase e retorna esta frase espacejada, ou seja, iremos adicionar um espaço em branco para cada caractere da frase.



FraseSpace.java

```
public class FraseSpace{
    public static void main(String[] args){
        System.out.println(espacando ("java"));
        System.out.println(espacando ("O método é uma boa prática de
programação"));
    }

    private static String espacando (String f){
        String aux="";
        for (byte i=0;i<f.length();i++){
            aux=aux+f.charAt(i)+ " ";
        }
        return aux;
    }
}
```

Note que o parâmetro pode ser passado também como um literal ("java"), não precisa ser uma variável, porque estamos usando passagem de parâmetro por valor, ou seja, é feita uma cópia do valor e colocada na variável que representa o parâmetro no caso *f*.

Para finalizar temos um exemplo com dois métodos, somatório e produto, que recebem um vetor de inteiros e devolvem a soma e o produto dos elementos respectivamente.

CalcVetor.java

```
public class CalcVetor{
    public static void main(String[] args){
        int[] vetor = new int[5];
        vetor[0]=10;
        vetor[1]=5;
        vetor[2]=7;
        vetor[3]=8;
        vetor[4]=1;
        System.out.println("O somatório é " + somatorio(vetor));
        System.out.println("O produto é " + produto(vetor));
    }

    private static int somatorio (int[] v){
        int aux=0; // Elemento neutro do somatório
        for (byte i=0;i<v.length;i++)
            aux+=v[i];
        return aux;
    }

    private static int produto (int[] v){
        int aux=1; // Elemento neutro do produto
        for (byte i=0;i<v.length;i++)
            aux*=v[i];
        return aux;
    }
}
```

Como exercício o aluno poderá criar funções para fatorial, equação de segundo grau, manipulações de strings, tais como uma função que faça a troca de uma substring por outra, por exemplo:

`replacesub("Java é uma ótima linguagem ... ótima mesmo", "ótima", "boa") = "Java é uma boa linguagem ... boa mesmo"`