II

Programação Avançada

6 Modularização de Programas

Onde é mostrado como um algoritmo pode ser dividido em vários pedaços que resolvem subproblemas específicos.

Pré-requisito(s):	Objetivos (ao final você deverá ser capaz de):
• Saber definir novos tipos de dados compostos • Saber declarar variáveis com esses tipos • Saber definir e utilizar vetores e classes como registros em Java	• Quebrar um problema em subproblemas menores • Entender o conceito de modularização de programas • Saber diferenciar funções e procedimentos • Entender os mecanismos de passagem de parâmetros • Saber programar com métodos em Java • Entender os mecanismos de passagem de parâmetros em Java

Todos os algoritmos que foram vistos até agora representavam a solução completa para um determinado problema proposto organizada em um único bloco de solução. Entretanto, para problemas mais complexos, a elaboração da solução torna-se mais facilitada quando conseguimos dividir o problema em pedaços lógicos mais simples, que possam ser resolvidos isoladamente. Uma vez resolvidos, são utilizados mecanismos para integrá-los em uma solução composta. Este capítulo mostra como realizar a divisão de problemas em subproblemas e como organizá-los em um algoritmos via modularização com funções e procedimentos. O capítulo mostra ainda como utilizar funções e procedimentos na linguagem Java através da implementação de métodos com passagem de parâmetros.

6.1 Subproblema

Um problema pode ter sua complexidade reduzida através de sua decomposição, isto é, da sua divisão em problemas menores. Uma técnica que pode ser empregada é realizar um refinamento sucessivo do problema, quebrando-o em partes com maior detalhamento. É importante verificar e garantir a coerência da divisão efetuada.

Como vimos, um algoritmo representa os passos para solução de um determinado problema. Em geral, problemas mais complexos exigem algoritmos mais extensos. Quando isso ocorre, o resultado é um amontoado de ações que não ficam muito claras, e é bem provável que existam trechos com instruções repetidas em vários pontos do algoritmo.

Uma solução para esta questão é de fato dividir este algoritmo mais complexo em pequenos algoritmos, cada um representando um subproblema identificado. Cada nível mais simples de uma divisão do algoritmo é chamado de **módulo**. Ao se modularizar um algoritmo, buscamos aumentar a funcionalidade das partes do conjunto, facilitando o seu entendimento e possibilitando a reutilização destas partes.

Um algoritmo completo é dividido em módulo principal e diversos módulos ou sub-algoritmos tantos quantos forem necessários. O módulo principal é aquele por onde a execução sempre se inicia, podendo invocar (ativar ou chamar) outros módulos que por sua vez poderão invocar outros ou até a si mesmos.

A chamada ao módulo, representa a execução das ações contidas nele, em seguida a execução retorna ao ponto da sua chamada (que poderá ser o módulo principal ou outros módulos). Não existe ordem para definição dos módulos.

Considere o problema de calcular o total de atrasos e de horas trabalhadas em um mês de um funcionário a partir de um cartão de ponto. O cartão de ponto marca os horários de entrada e de saída do funcionário no turno da manhã e no turno da tarde. O expediente começa às 8h e 14h, respectivamente.

O pseudocódigo da figura 6.1 ilustra o algoritmo não-modularizado para resolver o problema exposto.

```
1.  Algoritmo AtrasosHorasTrabalhadas;
2.  Tipos
3.     dia = registro
4.              em, sm, et, st: inteiro;
5.           fimregistro;
6.    totDia = registro
7.                atraso, horas: inteiro;
```

```
8.              fimregistro;
9.       V1 = vetor [1..31] de dia;
10.      V2 = vetor[1..31] de totDia;
11. Variaveis
12.    cartao: V1;
13.    totalDia: V2;
14.    dia, a, b, c, d, cont, i, me, ms, tm, tt, atrm, atrt,
15.                                        toth, totatr: inteiro;
16. Início
17.    cont <- 0;
18.    leia (dia);
19.    enquanto (dia > 0) e (dia < 32) faça
20.       leia (a, b, c, d);
21.       cartao[dia].em <- a;
22.       cartao[dia].sm <- b;
23.       cartao[dia].et <- c;
24.       cartao[dia].st <- d;
25.       cont <- cont + 1;
26.       leia (dia);
27.    fimenquanto;
28.    se cont > 0
29.    então
30.      inicio
31.         para i de 1 até cont faça
32.            me <- cartao[i].em;
33.            me <- (me / 100)*60 + me mod 100;
34.            ms <- cartao[i].sm;
35.            ms <- (ms / 100)*60 + ms mod 100;
36.            tm <- ms - me;
37.            atrm <- me - 480;
38.            me <- cartao[i].et;
39.            me <- (me / 100)*60 + me mod 100;
40.            me <- cartao[i].st;
41.            me <- (me / 100)*60 + ms mod 100;
42.            tt <- ms - me;
43.            totalDia[i].horas <- tm + tt;
44.            atrt <- me - 840;
45.            totalDia[i].atraso <- atrm + atrt;
46.            toth <- toth + (tm + tt);
47.            totatr <- totatr + (atrm + atrt);
48.         fimpara;
49.         para i de 1 até cont faça
50.            escreva (cartao[i].em, cartao[i].sm);
51.            escreva (cartao[i].et, cartao[i].st);
52.            escreva (totalDia[i].horas / 60);
53.            escreva (totalDia[i].horas mod 60);
54.            escreva (totalDia[i].atraso / 60);
55.            escreva (totalDia[i].atraso mod 60);
56.         fimpara;
57.         escreva ((toth/cont) / 60, (toth/cont) mod 60);
58.         escreva (toth / 60, toth mod 60);
59.         escreva ((totatr/cont) / 60, (totatr/cont) mod 60);
60.         escreva (totatr / 60, totatr mod 60);
61.      fim;
62.    fimse;
63. Fim.
```

Figura 6.1 – Algoritmo não modularizado para cálculo de horas trabalhadas e atrasos.

Assim como todos os demais pseudocódigos que vínhamos desenvolvendo até então, para o entendimento da lógica de resolução do problema há a real necessidade de uma leitura cuidadosa, linha por linha de todo o código; ou seja, por mais que tenhamos decomposto o problema mentalmente para poder conceber o algoritmo, esta decomposição não está claramente refletida na codificação, o que dificulta o entendimento.

6.1.1 Módulos

Podemos identificar no problema do cartão de ponto pelo menos três possibilidades claras de decomposição: (1) o subproblema da leitura dos dados de entrada, (2) o subproblema do cálculo dos atrasos e horas trabalhadas, e (3) o subproblema da impressão do resultado.

O subproblema do cálculo pode ainda ser subdividido em dois: um para cada turno de trabalho.

Podemos ir mais além e identificar que o cálculo do atraso em cada turno é composto por quatro probleminhas distintos: (1) compto do horário de entrada no trabalho, (2) compto do horário de saída do trabalho, (3) compto do atraso, e (4) compto do total de horas trabalhadas naquele turno.

A representação hierárquica dessa proposta de decomposição está ilustrada na figura 6.2.

Figura 6.2 – Proposta de decomposição do problema das horas trabalhadas e atrasos.

O pseudocódigo inicialmente elaborado para o problema, pode agora ser reorganizado seguindo a decomposição apresentada. Para isso, cada nó da árvore de decomposição ilustrada passará a representar um módulo no novo pseudocódigo. Esses módulos são declarados utilizando a notação a seguir:

módulo <*identificador*>

//declarações de variáveis internas

//comandos internos

fimmódulo;

onde, <identificador> é o nome pelo qual o módulo será referenciado no algoritmo.

Observe na figura 6.3 a versão modularizada do algoritmo para solução do problema exposto.

```
1.   Algoritmo AtrasosHorasTrabalhadasModularizado;
2.   Tipos
3.      dia = registro
4.              em, sm, et, st: inteiro;
5.            fimregistro;
6.    totDia = registro
7.                atraso, horas: inteiro;
8.             fimregistro;
9.       V1 = vetor [1..31] de dia;
10.      V2 = vetor[1..31] de totDia;
11.  Variaveis
12.     cartao: V1;
13.     totalDia: V2;
14.     cont, i, toth, totatr: inteiro;
15.
16.  Início
17.
18.     modulo Entrada
19.         dia, a, b, c, d: inteiro;
20.         cont <- 0;
21.         leia(dia);
22.         enquanto (dia > 0) e (dia < 32) faça
23.           leia (a, b, c, d);
24.           cartao[dia].em <- a;
25.           cartao[dia].sm <- b;
26.           cartao[dia].et <- c;
27.           cartao[dia].st <- d;
28.           cont <- cont + 1;
29.           leia (dia);
30.         fimenquanto;
31.     fimmodulo;
32.
33.     modulo Calculo
34.        tm, tt, atrm, atrt: inteiro;
35.
36.        modulo Manha
37.               me, ms: inteiro;
38.
39.               modulo MinutoEntrada
40.                  me <- cartao[i].em;
41.                  me <- (me div 100)*60 + me mod 100;
42.               fimmodulo;
43.
```

```
44.                modulo MinutoSaida
45.                   ms <- cartao[i].sm;
46.                   ms <- (ms / 100)*60 + ms mod 100;
47.                fimmodulo;
48.
49.                modulo Atraso
50.                   atrm <- me - 480;
51.                fimmodulo;
52.
53.                MinutoEntrada;
54.                MinutoSaida;
55.                tm <- ms - me;
56.                Atraso;
57.          fimmodulo;
58.
59.          modulo Tarde
60.                me, ms: inteiro;
61.
62.                modulo MinutoEntrada
63.                   me <- cartao[i].et;
64.                   me <- (me / 100)*60 + me mod 100;
65.                fimmodulo;
66.
67.                modulo MinutoSaida
68.                   ms <- cartao[i].st;
69.                   ms <- (ms / 100)*60 + ms mod 100;
70.                fimmodulo;
71.
72.                modulo Atraso
73.                   atrt <- me - 480;
74.                fimmodulo;
75.
76.                MinutoEntrada;
77.                MinutoSaida;
78.                tt <- ms - me;
79.                Atraso;
80.          fimmodulo;
81.
82.          para i de 1 até cont faça
83.               Manha;
84.               Tarde;
85.               totalDia[i].horas <- tm + tt;
86.               totalDia[i].atraso <- atrm + atrt;
87.               toth <- toth + (tm + tt);
88.               totatr <- totatr + (atrm + atrt);
89.          fimpara;
90.     fimmodulo;
91.
92.     modulo Impressao
93.          para i de 1 até cont faça
94.             escreva (cartao[i].em, cartao[i].sm);
95.             escreva (cartao[i].et, cartao[i].st);
96.             escreva (totalDia[i].horas / 60);
97.             escreva (totalDia[i].horas mod 60);
98.             escreva (totalDia[i].atraso / 60);
99.             escreva (totalDia[i].atraso mod 60);
100.         fimpara;
101.         escreva ((toth/cont) / 60, (toth/cont) mod 60);
```

```
102.          escreva (toth / 60, toth mod 60);
103.          escreva ((totatr/cont) / 60, (totatr/cont) mod 60);
104.          escreva (totatr / 60, totatr mod 60);
105.      fimmodulo;
106.
107.      Entrada;
108.      se cont > 0
109.      então
110.        inicio
111.           Calculo;
112.           Impressao;
113.        fim;
114.    fimse;
115.  Fim.
```

Figura 6.3 – Algoritmo modularizado para cálculo de horas trabalhadas e atrasos.

Observe que os módulos definidos nessa versão ado algoritmo correspondem à proposta de decomposição ilustrada na figura 6.2.

Para um determinado módulo ser executado, seu identificador precisa ser **invocado (ou chamado)** por um dos módulos mais externos, ou o próprio módulo principal (raiz da árvore de decomposição). Essa invocação (ou chamada) ocorre simplesmente escrevendo-se o nome do identificador do módulo que se deseja executar em algum momento do código. No algoritmo da figura 6.3 essas invocações de módulos ocorre nas linhas 53, 54, 56, 76, 77, 79, 83, 84, 107, 111 e 112.

Ao se invocar um módulo, o fluxo de execução do programa é desviado para o mesmo e, imediatamente após o término de sua execução, o fluxo retorna para o módulo que realizou a chamada exatamente no ponto onde havia parado.

O início de execução do algoritmo anterior se dá propriamente na linha 107 com a chamada do módulo `Entrada`. O fluxo é então desviado para esse módulo e ao final de execução do mesmo, o fluxo retorna para a linha 108. Mais uma vez, na linha 111, o fluxo é desviado, desta vez para o módulo `Calculo`. Dentro do módulo `Calculo`, ele é desviado outras vezes até retornar ao programa principal novamente na linha 112, quando imediatamente é desviado para o modulo `Impressao`. A figura 6.4 ilustra o fluxo de ativação de alguns módulos do algoritmo.

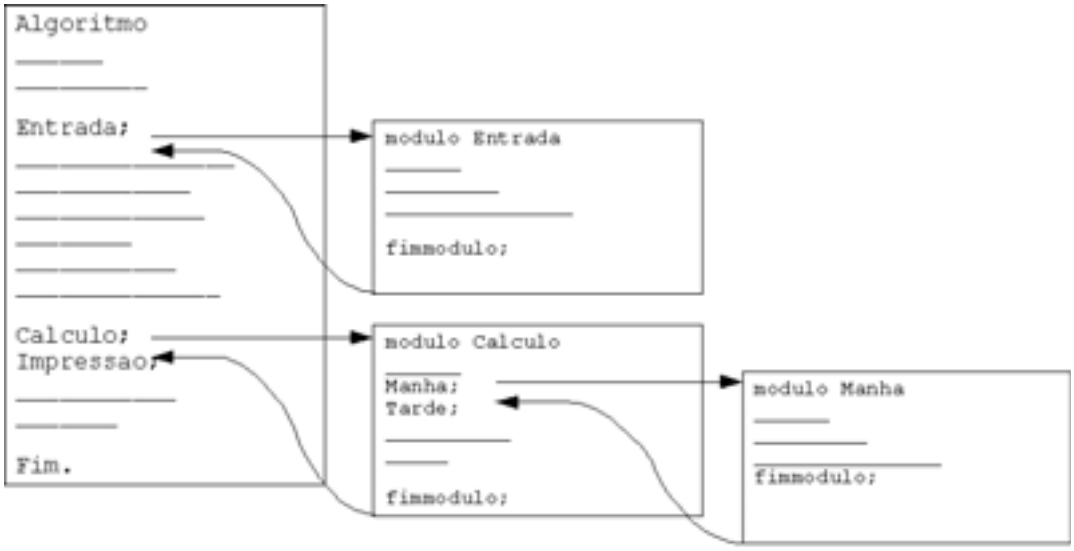

Figura 6.4 – Seqüência de ativação de módulos em um programa.

6.1.2 Escopo de variáveis

Uma vez que um módulo representa a solução de um problema em particular, é natural que ele possua características semelhantes a de um programa completo. Uma dessas características é a definição de variáveis.

Quando uma variável é declarada no módulo mais externo de um programa (o módulo principal), essa variável pode ser utilizada por todo e qualquer módulo definido naquele programa; dizemos, nesse caso, que se trata de uma **variável global**. Entretanto, existem variáveis cuja utilizada se resume aos limites lógicos de um módulo específico, sendo portanto injustificável sua declaração de forma global. Nesse caso, dizemos que se trata de uma **variável local**.

O escopo (ou abrangência) de uma variável denota sua visibilidade perante os diversos módulos que compõe o algoritmo de solução do problema. Essa visibilidade é relativa à hierarquia de decomposição. Ou seja, uma variável X pode ser ao mesmo tempo global e local: é global em relação aos módulos hierarquicamente inferiores, e local em relação aos módulos hierarquicamente superiores, não sendo portanto visível dentro destes.

Observe na figura 6.5 o escopo de cada uma das variáveis definidas para o algoritmo da figura 6.3.

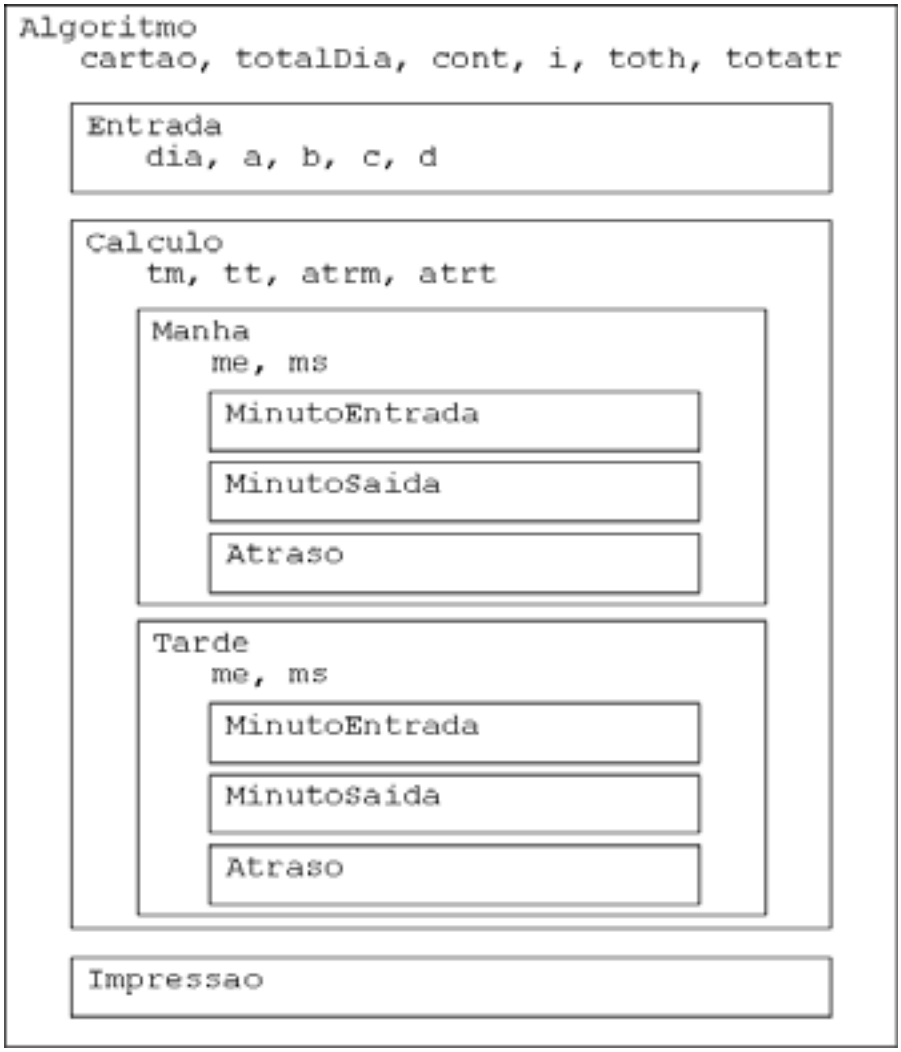

Figura 6.5 – Escopo de variáveis do algoritmo de horas trabalhadas e atrasos.

6.2 Procedimentos e Funções

Os módulos que vimos até agora executam comandos de leitura, impressão, e/ou realizam ações que tem por objetivo atualização do valor de variáveis globais ao programa principal.

Módulos desse tipo são comumente denominados de **procedimentos**. No problema exposto, temos então um procedimento de entrada de dados, procedimentos de cálculo, um procedimento de impressão de resultados, etc.

Um outro tipo de módulo que pode ser criado é chamado de **função.** Assim como na matemática, uma função representa algo que realiza o cálculo do valor de uma expressão e retorna o valor do resultado desse cálculo. Esse resultado deve ser atribuído a alguma variável externa à função.

Funções também são módulos e portanto sua declaração se dá de forma semelhante. A diferença está na presença do tipo do valor de retorno da função:

módulo <*identificador*>: <*tipo*>;
    //declarações de variáveis internas
    //comandos e expressões internos
    retorne valor;
fimmódulo;

onde, <identificador> é o nome pelo qual o módulo será referenciado no algoritmo, <tipo> é um dos tipos primitivos existentes (inteiro, real, logico, literal) ou um tipo composto definido pelo programador (vetores, registros, etc), e o termo *retorne* é obrigatoriamente utilizado para definir que valor ou expressão será retornada como resultado da execução da função.

Considere como exemplo do uso de funções, um algoritmo que lê continuamente vários vetores de inteiros e calcula para cada um a soma de seus elementos (figura 6.6). A leitura será finalizada quando a soma do último vetor for igual a 0 (zero).

```
1.  Algoritmo SomaVetores;
2.  Variáveis
3.      n: inteiro;
4.      v: vetor[10] de inteiro;
5.      s : real;
6.  Inicio
7.
8.      módulo Leitura_do_Vetor;
9.         i : inteiro;
10.        para i <- 1 até n faça
11.           leia( v[i]);
12.     fimmodulo;
13.
14.     modulo Soma : real;
15.        i: inteiro;
16.        S: real;
17.        S <- 0;
18.        para i <- 1 até n faça
19.          S <- S + v[i];
20.        retorne S;
21.     fimmodulo;
22.
23.     repita
24.       leia(n);
25.       Leitura_do_Vetor;
26.       s <- Soma;
27.       escreva("Soma = ", s);
28.     até s = 0;
29. Fim.
```

Figura 6.6 – Exemplo do uso de função.

Observe a definição do procedimento `Leitura_do_Vetor` (linha 8), que é chamado na linha 25, e que pede ao usuário que forneça `n` valores inteiros para preenchimento de um vetor. Note na linha 26 que a variável `s` está recebendo o valor resultante da invocação da função `Soma`. A função `Soma` é responsável por acumular em uma variável chamada `S` o somatório dos elementos do vetor lido (linhas 14 à 21). Note, na linha 20, que o valor armazenado na variável `S` é o valor de retorno da função `Soma`.

6.3 Passagem de parâmetros

Até então vimos como decompor um problema em pedaços menores e como eles podem ser representados através de módulos, sejam estes procedimentos ou funções. Essa decomposição em módulos ajuda a estruturar o algoritmo de solução do problema e melhora de forma significativa o seu entendimento.

Podemos, entretanto, aumentar ainda mais o poder de generalização desses módulos.

Suponha, por exemplo, um algoritmo que calcula o valor do volume de três esferas diferentes, com raios de tamanho 3, 4 e 5. Podemos modularizar esse algoritmo definimos três funções de cálculo, uma para esfera considerada (figura 6.7).

```
1.  Algoritmo VolumeEsferas;
2.  constantes
3.     pi: real = 3,14;
4.
5.  Início
6.
7.    modulo calcVolumeEsferaRaio3: real;
8.       R: real = 3; //constante
9.       retorne 4/3*pi*R*R;
10.   fimmodulo;
11.
12.   modulo calcVolumeEsferaRaio4: real;
13.      R: real = 4;
14.      retorne 4/3*pi*R*R;
15.   fimmodulo;
16.
17.   modulo calcVolumeEsferaRaio5: real;
18.      R: real = 5;
19.      retorne 4/3*pi*R*R;
20.   fimmodulo;
21.
22.   escreva(calcVolumeEsferaRaio3);
23.   escreva(calcVolumeEsferaRaio4);
24.   escreva(calcVolumeEsferaRaio5);
25. Fim.
```

Figura 6.7 – Uso de funções sem passagem de parâmetros.

Observe que no módulo principal fica bem claro o propósito do algoritmo: calcular os valores dos volumes de três esferas de raios 3, 4 e 5, respectivamente, e então imprimir esses valores.

Note porém que essa modularização é muito restrita. Ela não serve, por exemplo, para calcular o volume de esferas com raios diferentes ou ainda para calcular outras propriedades de uma esfera, como a área da superfície esférica ou a área da seção plana máxima. O formato da expressão do cálculo dessas três grandezas é similar:

x * pi * R ^ y

Para uma esfera de raio R, o volume é calculado quando x = 4/3 e y = 3. Para a área da superfície esférica, x = 4 e y = 2. Finalmente, para o cálculo da área da seção plana máxima de uma esfera (área do círculo), x = 1 e y = 2.

Ou seja, vemos claramente que x, R e y são apenas **parâmetros** de uma função matemática genérica para cálculo de grandezas de uma esfera.

A utilização de parâmetros em módulos funciona de maneira equivalente à da matemática; ou seja, a definição do módulo é única e genérica, o que irá variar são os valores de x, R e y fornecidos no momento da invocação (uso, aplicação) da função.

Um módulo parametrizado possui a seguinte sintaxe:

módulo <identificador> (par1:<tipo>, ..., parN:<tipo>, ...): <tipo>;

//declarações de variáveis internas

//comandos e expressões internos

fimmódulo;

onde, par1...parN são identificadores para cada um dos parâmetros definidos. No caso de um procedimento o tipo de retorno do módulo não existirá.

Considere agora uma versão bem mais genérica do algoritmo de cálculo do volume de esferas mostrado anteriormente. Com o uso apropriado de módulos parametrizados, o pseudocódigo da figura 6.8 exibe o valor do cálculo das três grandezas referentes a uma esfera, citadas anteriormente: volume, área da superfície, e área da seção plana máxima.

```
1.  Algoritmo GrandezasEsferas;
2.  constantes
3.     pi: real = 3,14;
4.  variáveis
5.     Raio, volume, superficie, seção: real;
6.
7.  Início
8.      modulo calcGrandezaEsfera(x, R: real, y: inteiro): real;
9.        retorne x*pi*R^y;
```

```
10.     fimmodulo;
11.
12.     escreva("Forneça o raio da esfera: ");
13.     leia(Raio);
14.     volume <- calcGrandezaEsfera(4/3, Raio, 3);
15.     superficie <- calcGrandezaEsfera(4, Raio, 2);
16.     secao <- calcGrandezaEsfera(1, Raio, 2);
17.     escreva("VOLUME: ", volume);
18.     escreva("SUPERFICIE: ", superficie);
19.     escreva("SEÇÃO: ", secao);
20. Fim.
```

Figura 6.8 – Uso de funções com passagem de parâmetros.

Observe a definição da função `calcGrandezaEsfera` (linha 8). A função possui três parâmetros, `X`, `R`, `y` que irão receber valores de acordo com a invocação dessa função. Por exemplo, na linha 14, a função é invocada para calcular o volume de uma esfera, cujo raio é fornecido pelo usuário (linha 13). Como dito anteriormente, o valor do parâmetro `X` para o cálculo do volume de uma esfera deve ser 4/3 e do parâmetro `y` deve ser 3. Veja raciocínio semelhante para as chamadas das linhas 15 e 16.

6.4 Modularização em Java

A linguagem Java trata com profundidade a questão da modularização de programas. Na verdade, existem três níveis diferentes de modularização em Java: o conceito de pacotes, de classes e de métodos. Neste curso, entretanto, iremos nos restringir à modularização via o uso de **métodos**.

Os métodos em Java correspondem às funções e procedimentos definidos anteriormente. Assim como vimos com o uso de funções e procedimentos, as três maiores motivações para o uso de métodos em Java é permitir uma melhor estruturação de um programa a partir de fragmentos mais simples (subproblemas), evitar a repetição de código em diferentes trechos do programa principal e generalizar uma solução a partir do uso de parâmetros.

Um **procedimento** em Java é um método que não possui um tipo de valor de retorno e é definido pela sintaxe abaixo:

```
public static void <nomeDoMetodo>(<parametros>) {
   //declaração de variáveis
   //instruções
},
```

onde a palavra reservada **void** significa que o método não possui um tipo para o valor de retorno.

Um exemplo de procedimento em Java é o método abaixo que escreve o nome

e a média de um aluno:

```
public static void escreveDadosAluno(String nome, float media)
{
   System.out.println("Nome = " + nome);
   System.out.println("Nota final = " + media);
}
```

O módulo principal de um programa Java é também um método que não retorna valor, e é obrigatoriamente chamado de `main`. Ou seja, todo programa Java precisa obrigatoriamente conter um método main para poder ser executado. Isto pode ser observado em todos os exemplos de programas Java que vimos até aqui.

O exemplo abaixo ilustra a definição do módulo principal de Java, onde é feita uma invocação ao procedimento `escreveDadosAluno(...)` definido acima.

```
public static void main(String[] args) {
   escreveDadosAluno("Maria", 8.5);
}
```

O parâmetro `nome` do procedimento iria corresponder ao valor `"Maria"` e o parâmetro `media` iria corresponder ao valor `8.5`. O resultado da execução desse programa seria então:

```
Nome = Maria
Nota final = 8.5
```

Uma **função** em Java também é um método. Porém, esse método possui um tipo de retorno. A sintaxe de declaração de funções em Java está ilustrada abaixo:

```
public static <tipo> <nomeDoMetodo> (<parâmetros>) {
      //declaração de variáveis
      //instruções
      return <resultado da expressão>;
}
```

onde *<tipo>* representa o tipo do valor que será retornado pela função, e a palavra reservada **return** causa o encerramento imediato da execução da função, retornando algum valor.

Considere a implementação do programa para cálculo das grandezas de uma esfera, ilustrado em pseudocódigo anteriormente, como exemplo da definição e uso de funções em Java (figura 6.9).

```
1.  public class GrandezasEsferas {
2.     static final int pi = (int)3.14;
3.     static float Raio, volume, superficie, secao;
4.
```

```
5.      static float calcGrandezaEsfera(float x, float R, int y) {
6.            return x*pi*(float)Math.pow(R,y);
7.      }
8.
9.      public static void main(String[] args) {
10.           System.out.print("Forneça o raio da esfera: ");
11.           Raio = System_in.readFloat();
12.           volume = calcGrandezaEsfera(4/3, Raio, 3);
13.           superficie = calcGrandezaEsfera(4, Raio, 2);
14.           secao = calcGrandezaEsfera(1, Raio, 2);
15.           System.out.println("\nVOLUME: "+volume);
16.           System.out.println("SUPERFICIE: "+superficie);
17.           System.out.println("SEÇÃO: "+secao);
18.     }
19. }
```

Figura 6.9 – Uso de funções com passagem de parâmetros em Java.

Note que em Java não existe operador para exponenciação, portanto, utilizamos uma chamada de função da classe `Math` (pré-definida da linguagem Java), chamada `pow`. A aplicação dessa função a dois valores `X` e `Y` realiza o cálculo X^Y.

Uma questão importante à cerca da passagem de parâmetros para para métodos em Java diz respeito à forma como as variáveis definidas nos parâmetros dos métodos armazenam os valores que recebem.

Quando as variáveis dos **parâmetros são de tipos primitivos**, a passagem é chamada de **passagem por valor**. Nesse formato de passagem, uma **cópia do valor** armazenado no endereço de memória identificado pela variável é feita e passada para o método invocado. Neste caso, eventuais alterações feitas nessa cópia não afetam o valor da variável original no método invocador.

Um exemplo de passagem por valor pode ser visto no programa da figura 6.10. Neste exemplo, foi definido um procedimento chamado translação que recebe dois valores do tipo `float` como argumentos. Ambos os argumentos são passados por valor, já que são de tipo de dados primitivos. No método principal, duas variáveis do tipo `float` são definidas (`x1` e `y1`) e inicializadas com os valores 15.5 e 10.5.

O método translação é então invocado e as duas variáveis são passadas como argumentos. Ao final do método translação as variáveis passadas como argumentos recebem o valor 0 (zero). Essa alteração é vista apenas dentro do método chamado e não é refletida no método chamador. Isso pode ser comprovado, ao se observar o resultado das duas linhas finais do método principal, que irão escrever na tela:

```
O valor de dx é 15.5
O valor de dy é 10.5
```

```
1.  class Retangulo{
2.      static float orig_x = 10, orig_y = 20;
3.      static float altura, largura;
4.
5.      public static void translacao (float x, float y) {
6.           //Realiza translação
7.           orig_x = orig_x + x;
8.           orig_y = orig_y + y;
9.
10.          //Altera valores dos argumentos
11.          x = (float)0.0;
12.          y = (float)0.0;
13.     }
14.
15.     public static void main( String args[] ){
16.          float dx = 15.5f;
17.          float dy = 10.5f;
18.          System.out.println("A origem é (" + orig_x + ",
19.                              "+orig_y+")");
20.          translacao(dx, dy);
21.          System.out.println("A nova origem é (" + orig_x + ",
22.                              "+orig_y+")");
23.          System.out.println("O valor de dx é " + dx);
24.          System.out.println("O valor de dy é " + dy);
25.     }
26. }
```

Figura 6.10 – Passagem de parâmetros por valor em Java.

A figura 6.11 ilustra a configuração de memória do programa anterior antes, durante e após a chamada do método `translacao` da linha 20.

Quando, entretanto, as variáveis dos **parâmetros são de tipos compostos**, como por exemplo uma variável do tipo vetor, a passagem é chamada de **passagem por referência**. Nesse formato de passagem, não é feita uma cópia dos valores das variáveis originais para as variáveis dos parâmetros do método invocado. O que acontece nesse caso é que o endereço de memória referenciado pelos identificadores das variáveis originais passa a ser referenciado também pelos identificadores das variáveis dos parâmetros. Dessa forma, caso o valor destas variáveis seja alterado dentro do método invocado, estaremos automaticamente alterando o valor das outras, já que referenciam o mesmo endereço de memória.

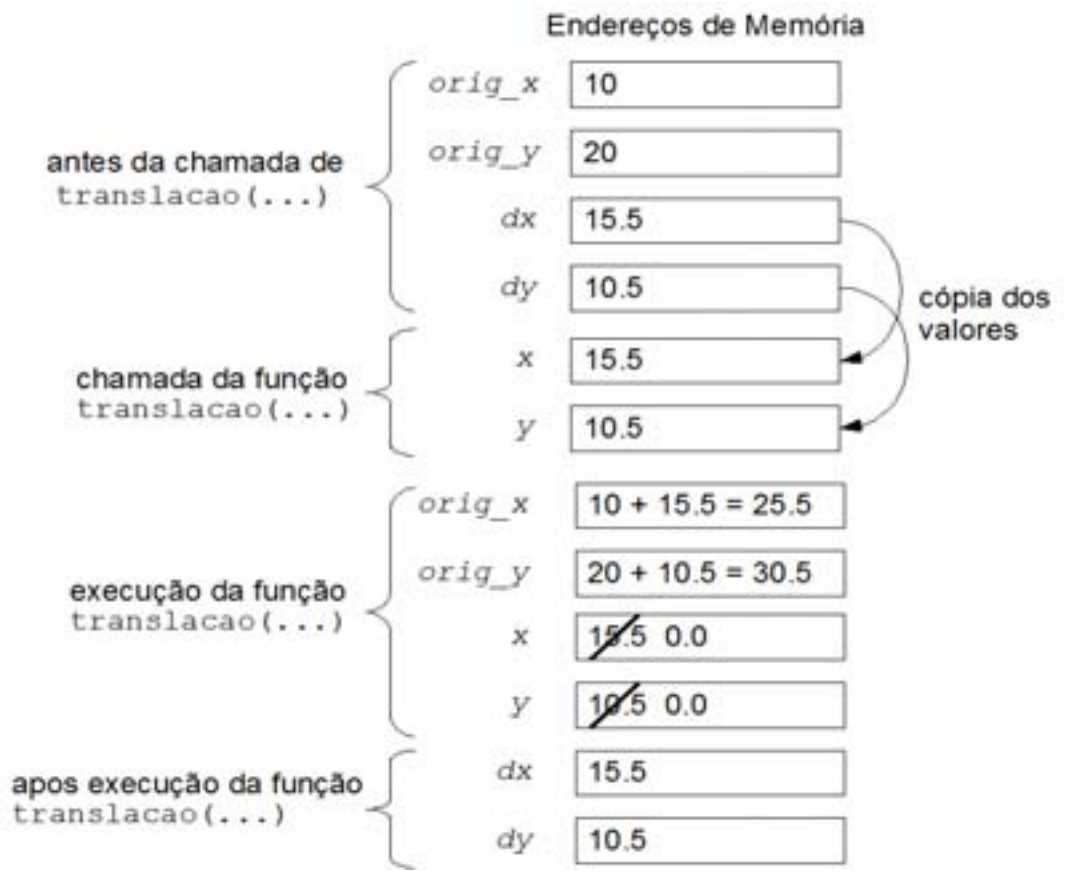

Figura 6.11 – Configuração de memória na passagem de parâmetros por valor em Java.

O programa da figura 6.12 ilustra a passagem de parâmetros por referência utilizando um vetor que é passado como parâmetro de uma função. A função multiplica todos os elementos do vetor passado por 3 (três).

```
1.  class Vetor{
2.     //Variável global da classe
3.     static String mensagem;
4.     //Método de inicialização do vetor
5.     public static void inicializa(){
6.     //Declaração e inicialização do vetor
7.           int v[] = {1, 2, 3, 4, 5};
8.           //Mensagem a ser escrita na tela
9.           mensagem = "Os valores originais do vetor são: ";
10.          for (int i = 0; i < v.length; i++ ){
11.                mensagem += " " + v[i];
12.          }
13.          /* Chamada ao método modificaVetor, passando v1 como
14.             argumento. Passagem por referência */
15.          modificaVetor( v );
16.          mensagem += "\nOs valores do vetor após modificação
17.                      são: ";
18.          for (int i = 0; i < v.length; i++ ){
19.                mensagem += " " + v[i];
```

```
20.           }
21.           modificaElemento( v[2] );
22.           mensagem += "\nO valor de v[2] é " + v[2];
23.     }
24.     //Modifica todos os elementos do vetor. Multiplica todos
25.       por 3
26.     public static void modificaVetor( int v1[] ){
27.           for (int j = 0; j < v1.length; j++ ){
28.                 v1[j] *= 3;
29.           }
30.     }
31.     //Modifica um elemento, multiplicando-o por 3
32.     public static void modificaElemento( int elem ){
33.           elem *= 3;
34.     }
35.     //Retorna a mensagem
36.     public static String getMensagem(){
37.           return mensagem;
38.     }
39.     public static void main( String args[] ){
40.           //Vetor v = new Vetor();
41.           Vetor.inicializa();
42.           System.out.println( Vetor.getMensagem() );
43.     }
44. }
```

Figura 6.12 – Passagem de parâmetros por referência em Java.

A classe `Vetor` é formada por uma variável global mensagem do tipo `String`, um procedimento chamado inicializa, que não recebe nenhum argumento, um procedimento chamado `modificaVetor`, que recebe como argumento um vetor (passado por referência), um procedimento chamado `modificaElemento` que recebe como argumento um elemento do tipo inteiro (passado por valor) e, finalmente, uma função chamada `getMensagem` que retorna uma `String`.

No método inicializa, um vetor `v` é declarado com 5 elementos (1, 2, 3, 4, 5). Em seguida, a variável `mensagem` recebe o valor: "*Os valores originais do vetor são: 1 2 3 4 5*"

O próximo passo é a invocação do método `modificaVetor` passando `v` como argumento. Esse método faz com que todos os elementos do vetor sejam multiplicados por 3 e essa modificação vai ser vista pelo método chamador, já que o vetor foi passada uma referência. A seguir, a variável `mensagem` passa a ser: "*Os valores do vetor após modificação são: 3 6 9 12 15*"

Por fim, o segundo elemento do vetor `v[2]` é passado para o procedimento `modificaElemento`. Dentro deste procedimento, o argumento passado é multiplicado por 3. Como `v[2]` se trata de um tipo primitivo, a passagem é por cópia e portanto seu valor não é alterado de 9 para 27, como se poderia achar.

Finalmente, a variável `mensagem` passa a ser:

"*Os valores originais do vetor são: 1 2 3 4 5*

Os valores do vetor após modificação são: 3 6 9 12 15

O valor de v[2] é 9"

A figura 6.13 ilustra a configuração de memória quando o método `modificaVetor` é invocado. Observe que o identificador `v1`, parâmetro do método, identifica o mesmo endereço de memória identificado por `v`, vetor original passado como argumento na linha 15.

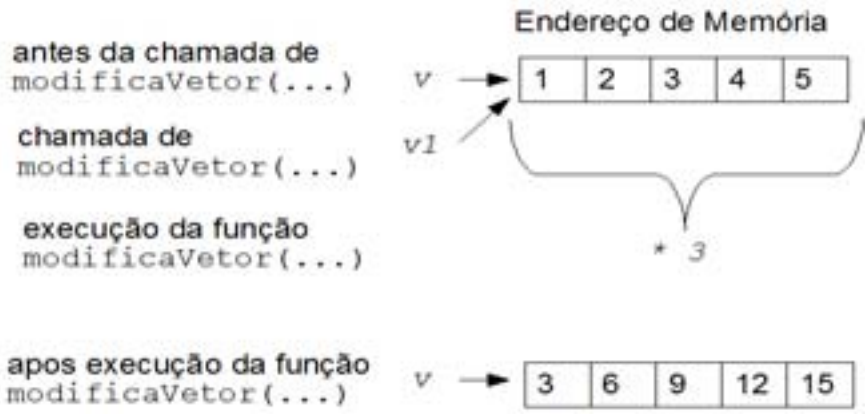

Figura 6.13 – Configuração de memória na passagem de parâmetros por referência em Java.

Resumo

- Problemas grandes podem ser decompostos em problemas menores.
- O algoritmo para solução de um problema desse tipo pode, mesma forma, ser decomposto pedaços menores.
- Em programação, chamamos esses pedaços de algoritmos criados para solucionar subproblemas, de módulos.
- Um módulo pode ser um procedimento ou uma função.
- Um procedimento executa uma seqüência de comandos, modificando ou não valores de variáveis.
- Uma função realiza a avaliação de expressões e retorna um valor como resultado de seu processamento.
- Tanto procedimentos quanto funções podem fazer uso de parâmetros, que aumentam o poder de genericidade de uma solução e a possibilidade de reuso de código.
- Parâmetros são identificadores definidos que assumem os valores passados na

chamada daquele procedimento ou função.

- Essa passagem pode ser de dois tipos: por valor ou por referência.
- Na passagem por valor, uma cópia dos valores armazenados no endereço de memória representado pela variável utilizada como argumento da passagem é feita para as variáveis definidas no parâmetro. Ou seja, os valores das variáveis originais não são alterados por manipulação das variáveis dos parâmetros.
- Na passagem por referência, as variáveis dos parâmetros passam a representar o mesmo endereço de memória que a variável passada representa e, portanto, qualquer alteração na variável do parâmetro é automaticamente refletida na variável passada.
- Em Java, funções e procedimentos são chamados de métodos.
- Valores primitivos em Java são passados por valor como argumentos de um método.
- Valores compostos como vetor, por exemplo, são passados como referência, em Java.

Na próxima aula...

… veremos como fazer para buscar um determinado item em uma lista de itens e como ordenar os valores de uma lista segundo algum critério.

Referências e Sugestões de Leitura

Modularização de programas também são abordados em

LEITE, M., Técnicas de Programação: uma Abordagem Moderna, BRASPORT, 2006.

no capítulo 7, e em

FORBELLONE, A., EBERSPÄCHER, H. Lógica de Programação. Prentice Hall, 3ed, 2005.

no capítulo 6

Detalhes aprofundados sobre subprogramas e passagens de parâmetros são amplamente discutidos no capítulo 9 de

SEBESTA, R. Conceitos de Linguagens de Programação, Bookman, 5ed, 2005.

Um exame profundo sobre métodos em Java é apresentado no capítulo 6 de **DEITEL, H., DEITEL, P. Java como Programar, Prentice-Hall, 2005.**

Exercícios Propostos

Para cada uma das questões abaixo, crie um programa em Java que utilize a função e/ou procedimento que se pede.

6.1 – Escrever uma função que receba como parâmetro o lado e altura de um triângulo e calcule a área.

6.2 – Criar uma função que calcule o volume de um cubo.

6.3 – Criar uma função que calcule o número de raízes de uma função de 2º grau a partir dos valores de A, B e C passados como parâmetro.

6.4 – Escrever uma função que retorne o valor do N-ésimo termo da seqüência de Fibonacci (1, 1, 2, 3, 5, 8, ...). Valide o valor de N, que deve estar entre 1 e 20, retornando 0 quando isso acontecer.

6.5 – Escrever uma função para retornar o maior valor entre 3 números inteiros passados como parâmetro.

6.6 – Escrever um procedimento para inverter uma string qualquer passada.

6.7 – Escrever um procedimento para retirar as vogais de uma string qualquer passada.

6.8 – Criar uma função que a partir de uma data (parâmetro do tipo string) no formato dd/mm/aaaa (Ex: 02/04/2004), retorne uma string no formato aaaammdd (Ex: 20040402).

6.9 – Escreva uma função que receba duas datas (parâmetros do tipo string) no formato dd/mm/aaaa e retorne:

a) -1 quando a 1º data for menor que a 2º data

b) 1 quando a 1º data for maior que a 2º data

c) 0 quando a 1º data for igual à 2º data

Dica: Utilize a função da questão anterior para facilitar a comparação das datas.

6.10 – Criar uma função que calcule a velocidade final (em km/h) de um automóvel, a partir dos valores da velocidade inicial (em m/s), da aceleração (m/s^2) e do tempo de percurso (em s) que serão digitados (V=V0 + A*T e Vkm/h = 3,6 * Vm/s), passados como parâmetro para a função.