Lógica Orientada a Objetos

www.3way.com.br

Seja um profissional Aprendendo com profissionais

SUMÁRIO

Algoritmos

Estruturas de Controle

Vetores

Matriz

Funções

Paradigma Orientação a Objetos

Algoritmos

Introdução

Neste curso estudaremos **Lógica de Programação** e, para isso é importante termos uma visão geral sobre as fases do processo de desenvolvimento de um programa (software), visto que nosso objetivo é construir uma base teórica e prática satisfatória para programação de computadores.

Pode-se encontrar na literatura, várias representações das etapas que compõem o ciclo de vida de um software. Para nosso estudo, será utilizado o Modelo Clássico, inicialmente apresentado por W. W. Royce.

Etapas do Modelo Clássico:

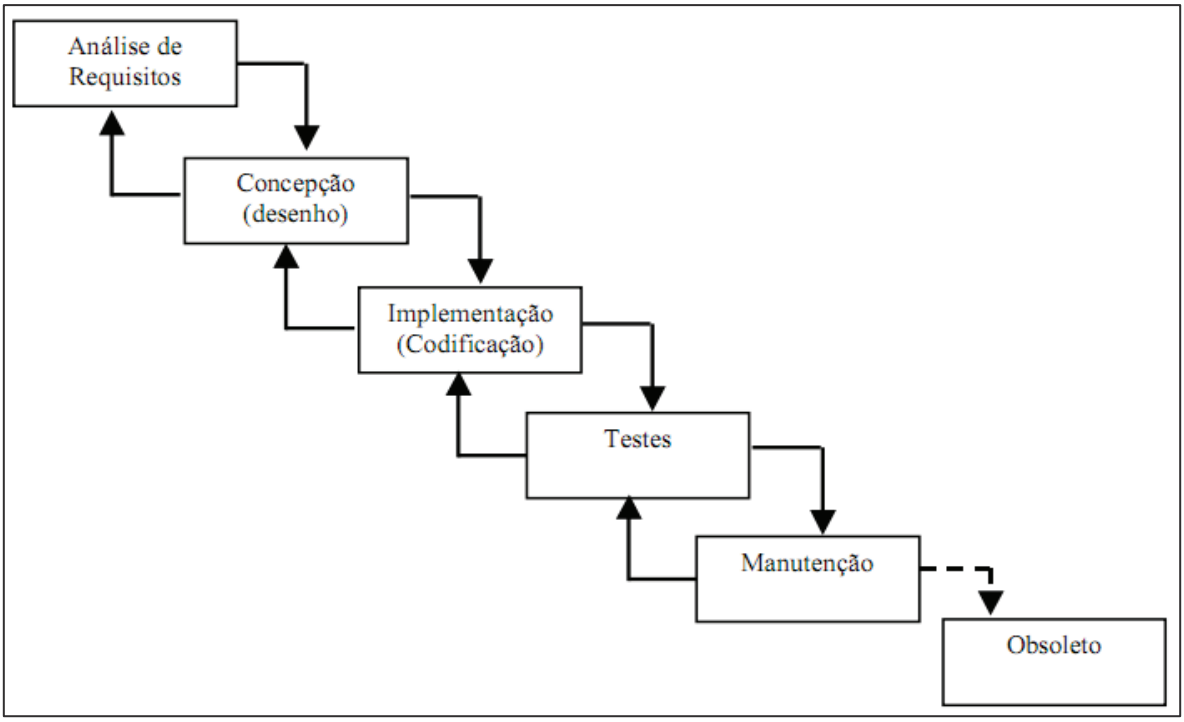

Figura 01: Modelo Clássico do processo de desenvolvimento de software.

1. **Análise de Requisitos**
   Nessa etapa devem ser definidas de forma clara as funcionalidades que o software deve apresentar. Esta fase será concluída ao se ter uma descrição completa e formal do comportamento do software a ser construído.

2. **Concepção**
   O objetivo desta etapa é traduzir os requisitos antes levantados na fase de análise, em uma representação de software com detalhes suficientes para que possa ser iniciada a fase de construção ou implementação.

3. **Implementação**
   Nesta etapa, ocorre a codificação do software, a tradução dos documentos elaborados nas fases anteriores para código (linguagem de programação). Neste momento as bases de dados são criadas e os módulos do software integrados.

4. **Teste**
   Esta etapa representa uma revisão final das fases de análise, concepção e implementação, portanto é um elemento crítico no controle da qualidade do software. Nesta fase, devem ser analisados os seguintes pontos:

   - Está sendo desenvolvido o produto de forma correta.
   - Está sendo desenvolvido o produto desejado.

   Com isso, são validados os procedimentos lógicos internos do programa, garantindo que todas as funcionalidades descritas na análise, produzam resultados esperados para determinadas entradas de dados.

5. **Manutenção**
   Nesta etapa ocorre a recodificação de determinadas funcionalidades do software, garantindo que o produto gerado esteja completamente de acordo com o desejado. O processo de alteração do software desenvolvido pode ser devido a:

   - Erros encontrados.
   - Requisitos alterados.
   - Adaptações devido ao ambiente externo.

6. **Obsoleto**
   Com o decorrer do tempo, o software se torna obsoleto, devido a mudanças no ambiente em que ele atua, ou no surgimento de novas tecnologias, mais apropriadas e eficientes na execução da função exercida por ele. Em geral o tempo de vida do software é grande, o que torna viável sua construção, visto o grande investimento feito em sua implementação.

É muito importante que todas estas etapas, citadas no modelo de desenvolvimento de software acima, sejam realizadas de forma a minimizar erros que possam prejudicar o produto utilizado na fase posterior, pois no desenvolvimento de um sistema, quanto mais tarde um erro é detectado, maior será o custo para repará-lo.

No modelo citado, os algoritmos seriam elaborados na 2º etapa do ciclo, na fase de Concepção. Assim sendo, a responsabilidade do programador é maior na criação dos algoritmos do que na sua

própria implementação, pois quando bem projetados, menor é o risco que se corre de ter que refazê-lo, reimplementá-lo e testá-lo novamente, assegurando assim o menor prazo e custo na entrega do produto final do software.

Mas, o que são algoritmos?

O que é algoritmo?

Um algoritmo pode ser definido como um conjunto de regras ou instruções bem definidas, para a solução de um determinado problema. Segundo o dicionário Michaelis, o conceito de algoritmo é a "utilização de regras para definir ou executar uma tarefa específica, ou para resolver um problema específico".

A partir destes conceitos, podemos observar que o termo "algoritmo" não se restringe a um termo computacional. É uma definição ampla, que pode ser utilizada em situações que não se refere à área da informática.

Tendo em vista, que um algoritmo é uma sequência de instruções para realizar determinada tarefa, temos a seguir um exemplo de um algoritmo que o professor passa a seus alunos em uma academia de ginástica:

1. Repetir 10 vezes os quatro passos abaixo:
   1.1. Levantar e abaixar braço direito.
   1.2. Levantar e abaixar braço esquerdo.
   1.3. Levantar e abaixar perna direita.
   1.4. Levantar e abaixar perna esquerda.

Para mostrar outro exemplo de algoritmo, não referente à computação, considere o seguinte problema:

*Temos duas jarras, **A** e **B** sem marcação da quantidade de litros contidos. Sabemos que a jarra **A** cheia possui 5 litros, e a **B** possui 3 litros. Queremos obter apenas 4 litros. Como proceder?*

1. Encher completamente a jarra **A**.
2. Despejar jarra **A** na jarra **B**. (assim teremos 3 litros na jarra **B** e 2 litros na jarra **A**)
3. Esvaziar jarra **B**.
4. Despejar jarra **A** na jarra **B**. (assim a jarra **A** ficará vazia e a jarra **B** terá 2 litros)
5. Encher completamente a jarra **A**.

6. Despejar jarra **A** na jarra **B**. (assim teremos 3 litros na jarra **B** e 4 litros na jarra **A**)
7. Entregar jarra **A**.

Na informática, o algoritmo é o "projeto do programa", as instruções que o computador deve executar para que o resultado esperado pelo usuário seja apresentado. Antes de fazer um programa, utilizando linguagens de programação (Java, C, C++, Delphi, Pascal, etc.), deve-se fazer o algoritmo do mesmo, onde todas as ações a serem executadas serão especificadas nos mínimos detalhes, para facilitar sua codificação.

A noção de algoritmo é de extrema importância para a área de desenvolvimento de software. Para resolver um determinado programa no computador, ou seja, para construir um software, é necessário que seja primeiramente encontrada uma maneira de descrever uma sequência de passos que permitam que o problema seja especificado de uma forma clara e precisa.

Uma das formas mais eficazes de aprender a construir algoritmos é através da resolução de exercícios, o que consequentemente aumenta o raciocínio lógico do desenvolvedor. O aprendizado da Lógica de Programação é fundamental para a formação de um bom desenvolvedor, servindo como base para o aprendizado de qualquer linguagem de programação, sendo ela orientada a objetos ou não.

Neste curso iremos estudar os passos básicos e as técnicas para construção de algoritmos. Serão apresentadas três formas de representação de um algoritmo, que são alguns dos mais conhecidos e utilizados.

O objetivo ao final deste curso é que você tenha adquirido a capacidade de transformar qualquer problema em um algoritmo de boa qualidade.

Formas de representação de algoritmos

Os algoritmos podem ser escritos de várias formas, cabendo ao desenvolvedor escolher qual forma utilizar, segundo sua disposição e facilidade em escrevê-la, como por exemplo:

- **Linguagem natural** (português, inglês, espanhol, etc.): forma utilizada pelo professor de ginástica no exemplo citado acima, em manuais de instruções, receitas de bolo, bulas de medicamentos, etc.

- **Representações gráficas** (diagramas, fluxogramas, etc.): emprega formas geométricas padronizadas para indicar as diversas ações e decisões que devem ser executadas para resolver o problema.

- **Linguagem de programação** (Java, C#, Delphi, etc.): esta forma é utilizada por alguns programadores já experientes, que "pulam" a etapa do algoritmo, e iniciam direto na codificação.

Algumas das formas mais utilizadas para representação de algoritmos são:

- **Diagrama de Nassi-Shneiderman** (Diagrama de Chapin)
- **Fluxograma** (Diagrama de Fluxo)
- **Português estruturado** (Pseudocódigo, Portugol ou Pseudolinguagem)

Entre os especialistas, não existe um consenso em qual a melhor forma de representar um algoritmo. Neste curso e na solução dos exercícios, será utilizada a forma de representação do Português Estruturado, pois é a forma mais didática de aprender e representar a lógica dos problemas, além de facilitar a tradução deste para uma linguagem de programação. Mas fica a critério de cada um escolher a forma mais conveniente de representar um algoritmo. A seguir será apresentada uma visão geral sobre cada uma das formas de representação.

Diagrama de Nassi-Shneiderman (Diagrama de Chapin)

Os Diagramas de Nassi-Shneiderman, também conhecidos como Diagramas de Chapin, surgiram nos anos 70, como uma maneira de ajudar na representação das soluções de problemas. Abaixo, temos um exemplo de um algoritmo para verificar a aprovação de um aluno utilizando a representação do Diagrama de Nassi-Shneiderman.

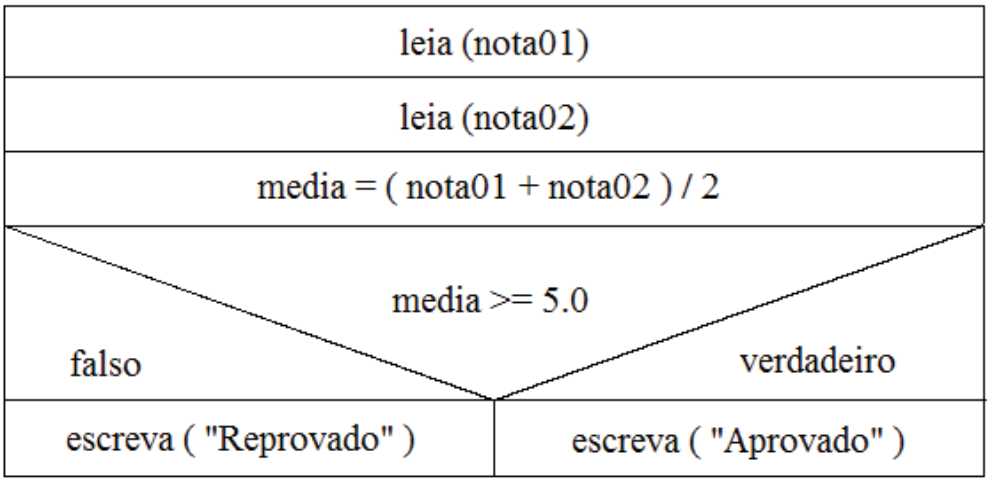

Figura 02 – Diagrama de Nassi-Shneiderman

A idéia básica deste diagrama é representar as ações de um algoritmo dentro de um único retângulo, subdividindo-o em retângulos menores, que representam as sequências de passos (instruções) do algoritmo.

Fluxograma

Os fluxogramas ou diagramas de fluxo são representações que utilizam formas geométricas bem definidas para indicar as ações e decisões tomadas pelo algoritmo. Dessa forma, podemos visualizar os diversos caminhos (fluxos) que o algoritmo pode seguir, prevendo as possíveis etapas de processamento dos dados para alcançar a solução do problema em questão. A seguir, mostraremos as formas geométricas utilizadas no diagrama, para representar as instruções de um algoritmo.

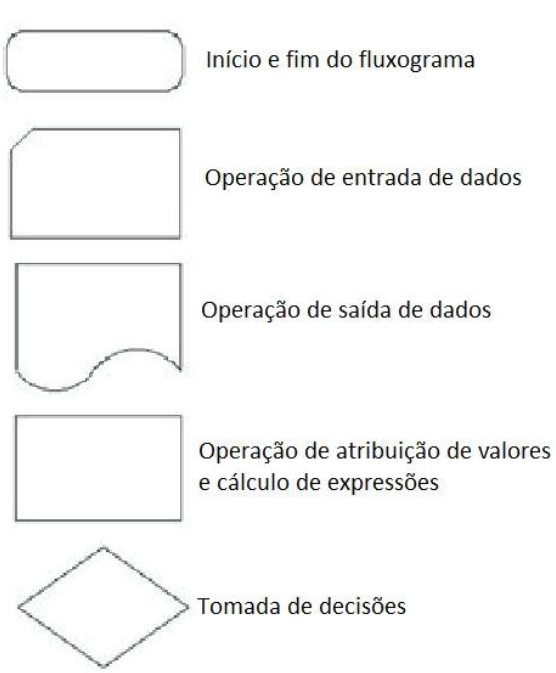

Figura 03 – Formas Geométricas de um Fluxograma.

Utilizando as formas geométricas acima citadas, podemos construir um algoritmo para o mesmo problema de verificar a aprovação de um aluno, só que dessa vez utilizando a forma de representação do Fluxograma ou Diagrama de Fluxo. Veja abaixo como representá-lo:

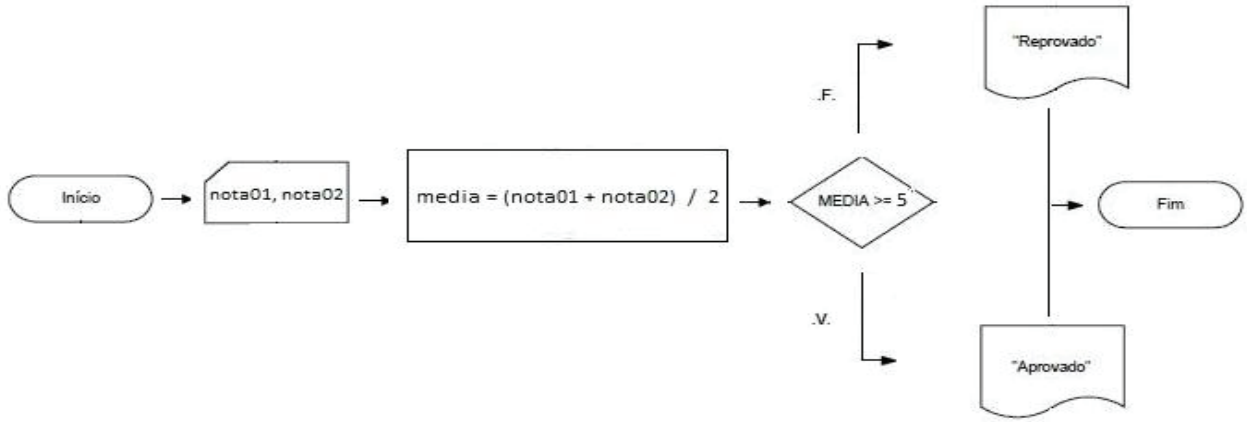

Figura 04 – Fluxograma do cálculo da média de um aluno.

Português Estruturado

Também conhecido como Portugol, Pseudocódigo ou Pseudolinguagem, o português estruturado é na verdade uma linguagem bem mais restrita que a língua portuguesa, com pouquíssimas palavras e estruturas, que representam de forma bem definida todos os termos utilizados nas instruções de uma linguagem de programação. O português estruturado é uma linguagem intermediária entre a linguagem natural (língua portuguesa) e uma linguagem de programação, utilizada como uma forma de representação de algoritmos.

Não existe um padrão a ser seguido na sintaxe utilizada por esta forma de representação, ao contrário de quando utilizamos uma linguagem de programação, como Java ou PHP; pois um algoritmo não passa de um rascunho, que não será compilado nem interpretado, com o objetivo de apenas expor a idéia para solução de um determinado problema.

Apesar da simplicidade da linguagem, com poucas palavras, o português estruturado possui todos os elementos básicos e estrutura semelhante a uma linguagem de programação de computadores. Portanto, resolver um problema utilizando esta forma de representação de algoritmos, pode ser tão complexo quanto utilizar diretamente uma linguagem de programação, porém a sintaxe do português estruturado não precisa ser seguida tão rigorosamente.

Nesta apostila, a forma de representação de algoritmos utilizada será o português estruturado, por ser uma forma mais didática de se entender e pela facilidade da tradução desta forma para qualquer linguagem de programação, sendo ela orientada a objetos ou não. Para facilitar nosso entendimento e aprendizado, além de tornar o algoritmo mais legível, este seguirá a seguinte estrutura:

```
algoritmo {nome do algoritmo}

var

  {declaração de variáveis}
```

```
inicio
   {instruções do algoritmo}
fimalgoritmo
```

Utilizando a estrutura acima padronizada, podemos construir um algoritmo para o mesmo problema de verificar a aprovação de um aluno, só que dessa vez utilizando a forma de representação do Português Estruturado. Veja abaixo como representá-lo:

```
algoritmo "CalculaMedia"
var
    nota01, nota02 : inteiro
    media : real
inicio
    escreva ("Informe a primeira nota do aluno:")
    leia (nota01)
    escreva ("Informe a segunda nota do aluno:")
    leia (nota02)
    media ← (nota01 + nota02) / 2
    se   media >= 5   entao
        escreva ("Aprovado")
    senao
        escreva ("Reprovado")
   fimse
fimalgoritmo
```

Conforme vimos até o momento, precisamos ter alguns conceitos bem definidos, fundamentais para a criação de algoritmos: como funcionam suas estruturas, o que são blocos, variáveis, e várias outras sintaxes utilizadas nas representações. A seguir, serão apresentadas as estruturas básicas de um algoritmo, que também serão utilizadas com a maioria das linguagens de programação. É importante lembrar que, se aprende a desenvolver algoritmos, construindo e testando-os, ou seja, resolvendo o

máximo de exercícios possíveis, dos mais variados níveis de complexidade. Não se aprende a construir algoritmos, copiando ou estudando algoritmos já prontos. Portanto, mãos a obra!

Instruções LEIA e ESCREVA

Existem basicamente duas instruções principais em um algoritmo, que são: **LEIA** e **ESCREVA.**

A instrução **ESCREVA** é usada quando se deseja apresentar algo na tela do computador, ou seja, é uma instrução de **saída de dados.** Nesta etapa do "projeto do programa" não nos interessa como este dado será mostrado, se é através de uma interface gráfica ou apenas pelo console do sistema operacional.

A instrução **LEIA** é usada quando se deseja obter algum dado do usuário. O algoritmo aguarda o usuário entrar com algum dado desejado, ou seja, é uma instrução de **entrada de dados.** Novamente, não precisamos nos preocupar com a forma de obtenção deste dado, apenas obtemos do usuário. Esta preocupação se torna indispensável, no momento da implementação do algoritmo, quando este é traduzido para a linguagem de programação.

Considere o algoritmo abaixo, que representa uma solução para o seguinte problema:

Um trabalhador recebeu seu salário e o depositou em sua conta bancária. Este trabalhador emitiu dois cheques e agora precisa saber seu saldo atual. Sabe-se que cada operação bancária de retirada paga CPMF de 0,38%. Também precisa saber se seu saldo está superior a R$ 200,00, pois precisa pagar as contas do mês.

```
algoritmo "SaldoTrabalhador"
var
   saldoFinal, saldoInicial, valorCheque01,valorCheque2, cpmf01, cpmf02 : real
inicio
   escreva ("Informe o saldo inicial da conta:")
   leia (saldoInicial)
   escreva ("Informe o valor do primeiro cheque:")
   leia (valorCheque01)
   escreva ("Informe o valor do segundo cheque:")
   leia (valorCheque02)
   cpmf01 ← valorCheque01 * 38 / 100
```

```
cpmf02 ← valorCheque02 * 38 / 100
saldoFinal ← saldoInicial – (valorCheque01 + cpmf01 + valorCheque02 + cpmf02)
se saldoFinal >= 200 entao
    escreva ("Saldo superior a R$ 200,00.")
senão
    escreva ("Saldo inferior a R$ 200,00.")
fimse
escreva ("Saldo atual:", saldoFinal)
fimalgoritmo
```

Tipos de variáveis

Na maioria das linguagens de programação, quando se declara uma variável é necessário informar qual o tipo de dado que ela irá armazenar, para que o computador reserve espaço suficiente na memória. Linguagens que seguem esta regra são chamadas de linguagens fortemente tipadas.

Como estamos na etapa do "projeto do programa", ou da construção do algoritmo, estes tipos de variáveis devem ser genéricos, pois cada linguagem de programação tem suas peculiaridades e particularidades. Enquanto no algoritmo não precisamos nos preocupar com a sintaxe, na linguagem de programação, devemos segui-la rigorosamente. Por exemplo: enquanto no algoritmo podemos declarar uma variável que irá armazenar um valor fracionário como:

nota : ***real***

Em uma linguagem de programação, devemos nos preocupar com sua sintaxe e palavras reservadas. Em Java poderíamos declarar esta mesma variável assim:

double *nota;*

ou ainda,

float *nota;*

Portanto, temos mais liberdade no algoritmo para informarmos o tipo de variável do que em uma linguagem de programação, que é aonde de fato vamos nos preocupar com a sintaxe a ser utilizada. Porém o tipo de dado informado no algoritmo deve ser significativo, ou seja, deve se deixar claro no momento da tradução para uma linguagem de programação qual tipo de dado mais apropriado deverá ser usado para aquela variável, dentro das especificações da linguagem utilizada.

Na construção de algoritmos, podemos utilizar os seguintes tipos de variáveis:

- **inteiro**: Números inteiros positivos ou negativos. Ex.: -150, 12, 2011...
- **real**: Números fracionários positivos ou negativos. Ex.: 343.45, -34.2, 15,7...
- **lógico**: Tipos lógicos ou booleanos que podem assumir: Verdadeiro ou Falso.
- **literal ou caractere**: São formados por um ou mais caracteres, representando um texto. Ex.: “abc”, “3Way Networks”, “12345”...

Variáveis

O bom entendimento do conceito de variável é fundamental para construção de algoritmos e, consequentemente de programas. Uma variável nada mais é do que um **espaço na memória** alocado para armazenar dados.

Como o próprio nome sugere, variável pode armazenar valores diferentes no decorrer da execução do algoritmo. Portanto, um único espaço na memória armazena valores diferentes em instantes diferentes, dependendo das instruções executadas. Mas como acessar este espaço alocado na memória?

Simples. As variáveis são referenciadas através de um nome (identificador), criado pelo desenvolvedor no momento da sua declaração. Este nome ou identificador deve ser sugestivo, ou seja, se quisermos declarar uma variável (espaço na memória) que irá armazenar o nome de um cliente, por exemplo, poderíamos chamá-la de “nomeCliente”, e não de “x”; pois ajuda no entendimento da lógica e da função do algoritmo. Veja um exemplo do mesmo algoritmo, utilizando identificadores de forma incorreta, e outro de forma correta.

Utilização incorreta

```
algoritmo “Alg”
var
   a : real
inicio
   leia ( a )
   se  a  >  500.50  entao
      escreva (“Operação realizada com sucesso !”)
```

```
   senao
      escreva ("ERROR: Contate o administrador do sistema!")
   fimse
fimalgoritmo
```

Utilização correta

```
algoritmo "VerificaDepositoSalario"
var
   saldo : real
inicio
   leia(saldo)
   se  saldo > 500.50  entao
         escreva ("Operação realizada com sucesso !")
   senao
         escreva ("ERROR: Contate o administrador do sistema !")
   fimse
fimalgoritmo
```

Observe que utilizando identificadores com significado para o domínio do problema, entender o algoritmo e determinar seu objetivo, se torna bem mais fácil do que se os identificadores não forem sugestivos.

Vejamos na imagem abaixo, o que acontece na declaração e inicialização de uma variável:

Instrução	Memória	Descrição
idade : inteiro	XH22A idade [ ]	Declarando a variável.
idade <-- 22	XH22A idade [22]	Inicializando a variável.

Tabela 01 – Declaração e inicialização de uma variável.

Fica claro na ilustração acima, que uma variável (idade) é então um espaço na memória (XH22A) que armazena um determinado tipo de dado (inteiro).

Constantes

São chamadas de constantes, as informações (dados) que não variam com o tempo, ou seja, ao contrário de uma variável, uma constante possui um valor fixo, que independente das instruções do algoritmo seu conteúdo permanece o mesmo.

```
algoritmo "ExemploConstante"
var
   idade, anoAtual, anoNascimento : inteiro
inicio
   anoNascimento ← 1989
   escreva ("Informe o ano atual:")
   leia (anoAtual)
   idade ← anoAtual – anoNascimento
   anoNascimento ← 2000
    escreva ("Hoje estou com " , idade , "anos de idade.")
fimalgoritmo
```

X

Na construção de um algoritmo, não temos uma sintaxe própria para a declaração de uma constante. Para simular esta condição, atribuímos um valor fixo à variável no início do algoritmo, que teoricamente, não será mais alterado.

Dito isto, podemos observar que no algoritmo acima, consideramos **anoNascimento** como uma constante. Portanto após atribuir um valor a ela, ou seja, após inicializá-la, o valor atribuído se torna fixo, inalterável.

Dessa forma, quando tentamos executar a instrução:

anoNascimento ← 2000

ocorre uma falha na lógica aplicada ao algoritmo, pois estamos tentando alterar o valor de uma constante, o que não é permitido. Como o próprio nome sugere, uma constante é capaz de assumir

apenas um valor em todo tempo de vida do algoritmo, sendo que se tentarmos alterá-la, estaremos causando um erro na lógica do mesmo.

Declaração de variáveis

No bloco das declarações (var), é onde vamos descrever todas as variáveis que serão utilizadas pelo algoritmo. É neste momento, que o computador aloca espaço na memória para armazenar os valores destas variáveis de acordo com o seu tipo.

Veja as sintaxes utilizadas na declaração de uma ou mais variáveis:

```
identificador1, identificador2, ..., identificadorN : tipo
```

Ex.:

```
a, b, c : inteiro
x, y, z : lógico
```

ou ainda,

```
identificador1 : tipo
identificador2 : tipo
...
identificadorN : tipo
```

Ex.:

```
a : inteiro
b : inteiro
x : inteiro
z : real
```

Conforme mostrado, podemos declarar mais de uma variável na mesma linha se estas forem do mesmo tipo, ou dividir uma instrução por linha, independente do tipo das variáveis.

Dependendo da complexidade do algoritmo, a preocupação neste tipo de organização se torna irrelevante. Porém se tratando de um algoritmo complexo, com uma lógica confusa, esta organização pode ajudar muito no seu entendimento, pois o torna mais legível.

Atribuição

A atribuição é uma notação utilizada para atribuir um valor a uma variável, ou seja, para armazenar determinado conteúdo na memória, para em algum momento posterior utilizá-lo. Normalmente, é utilizada uma seta apontando para a esquerda, para representar uma expressão de atribuição, mas se tratando de algoritmo, sabemos que não temos uma sintaxe a ser seguida rigorosamente, portanto, esta notação é apenas uma convenção. A seguir são apresentados alguns exemplos de atribuição possíveis:

- **variável ← constante** *(Ex.: anoNascimento ← 2011)*
- **variável ← variável** *(Ex.: produto ← preço)*
- **variável ← expressão** *(Ex.: idade ← anoAtual - anoNascimento)*

Em todos os casos, lê-se *variável recebe valor,* por exemplo: *anoNascimento recebe 2011.*

Conforme mostrado nos exemplos, sempre teremos do lado esquerdo da notação uma variável, nunca uma expressão. O exemplo abaixo:

var1 – var2 ← variável

está incorreto, e causaria uma falha na execução do algoritmo.

Operadores Aritméticos

No desenvolvimento de um software, ou na solução de um problema representado por um algoritmo, muitas das vezes precisamos escrever expressões matemáticas para realização de cálculos. Segue abaixo os símbolos utilizados para representar as operações matemáticas na construção de um algoritmo:

Operação Matemática	Símbolo utilizado
Multiplicação	*
Divisão	/
Adição	+
Subtração	-
Resto da divisão	% ou *mod*
Potenciação	^

Tabela 02 – Operadores Aritméticos.

As operações matemáticas e os símbolos utilizados no algoritmo para a realização de cálculos obedecem às regras matemáticas comuns, ou seja:

- As expressões escritas dentro de parênteses tem maior prioridade que as escritas de fora.
- Quando existem vários níveis de parêntese, ou seja, um dentro do outro, o mais interno tem maior prioridade que o mais externo.
- Quando duas ou mais expressões tiverem a mesma prioridade, a solução é sempre iniciada da expressão mais a esquerda até a expressão mais a direita, obedecendo as duas regras citadas a cima.

Veja abaixo, exemplos da importância destas regras em uma expressão matemática:

Ex.:

*2 + (6 * (3 + 2)) = 32*

*2 + ((6 * 3) + 2) = 22*

10 % 3 = 1

10 mod 4 = 2

5 ^ 2 = 25

Operadores de Caracteres

Existem dois tipos de operadores utilizados com operandos do tipo caractere ou literal, ou seja, com textos. São eles:

Utilização	Símbolo utilizado
texto + texto	+
texto + variável	,

Tabela 03 – Operadores de Caracteres.

Quando o objetivo for concatenar, isto é, unir dois textos, utilizamos o operador ‘ + ’ para isso.

Ex.:

escreva *(“ 3Way” + “Networks”)*

Saída:

“*3Way Networks*”

...

Quando o objetivo for concatenar um texto a uma variável que não seja do tipo **caractere** ou **literal**, utilizamos o operador ' , ' para isso.

Ex.:

idade : ***inteiro***

nome : ***caractere***

idade ← *22*

nome ← *"Bruno M. Zafalão"*

escreva *(" Meu nome é "* + nome + *" e tenho "* , idade , *" anos de idade.")*

Saída:

"Meu nome é Bruno M. Zafalão e tenho 22 anos de idade."

...

Operadores Relacionais

Operações relacionais são operações de comparação entre valores, variáveis, expressões ou constantes. O retorno deste tipo de operação é sempre lógico, ou seja, verdadeiro ou falso, dependendo da comparação feita.

Veja abaixo as operações relacionais possíveis em um algoritmo:

Símbolo	Comparação
>	maior que
<	menor que
=	igual a
>=	maior ou igual a
<=	menor ou igual a
< >	diferente de

Tabela 04 – Operadores Relacionais.

Ex.:

algoritmo "*ExemplosRelacionais*"

```
var
   var1, var2, var3 : inteiro
   resultado : lógico
inicio
   var1 ← 10
   var2 ← 5
   var3 ← 15
   resultado ← ( var1 + var2 ) = var3
   escreva  (resultado)
   resultado ← var1 > var2
   escreva  (resultado)
   resultado ← var2 <> 5
   escreva  (resultado)
fimalgoritmo
```

Saída:

```
verdadeiro
verdadeiro
falso
```

...

Operadores Lógicos

Os operadores lógicos permitem que mais de uma comparação seja feita ao mesmo tempo, ou seja, permitem que mais de uma condição seja testada em uma única expressão. Veja abaixo os operadores lógicos que podemos utilizar na construção de um algoritmo:

Operação	Operador
Negação	NAO
Conjunção	E
Disjunção (não-exclusiva)	OU
Disjunção exclusiva	XOU (lê-se *ou exclusivo*)

Tabela 05 – Operadores Lógicos.

A tabela acima mostrada apresenta os operadores lógicos já ordenados de acordo com suas prioridades, ou seja, se na mesma expressão tivermos o operador NAO e um operador E, a operação NAO deve ser executada primeiro que a operação E.

É importante lembrar também que o resultado destas operações, assim como os operadores relacionais, sempre resultará em um valor lógico (verdadeiro ou falso).

Veja abaixo, as tabelas verdade de cada operador, e exemplos de sua utilização.

Operador *NAO*

Operação	Resultado
NAO *verdadeiro*	falso
NAO *falso*	verdadeiro

Tabela 06 – Operador *NAO*.

Observe na tabela verdade acima, que utilizando o operador **NAO**, ao negarmos uma expressão verdadeira, teremos uma expressão falsa (***NAO*** **V = F**) e analogamente, ao aplicarmos o operador **NAO** em uma expressão falsa, teremos como resultado, uma expressão verdadeira (***NAO*** **F = V**).

Ex.:

```
algoritmo "OperaçãoNAO"
var
   resultado01, resultado02 : logico
inicio
   resultado01 ← NAO ( 10 > 5 )
```

```
    resultado02 ← NAO ( 5 > 10 )

    escreva (resultado01)

    escreva (resultado02)

fimalgoritmo
```

Saída:

```
falso
verdadeiro
...
```

Operador *E*

Operação	Resultado
V *E* **V**	verdadeiro
V *E* **F**	falso
F *E* **V**	falso
F *E* **F**	falso

Tabela 07 – Operador *E*.

Observe na tabela verdade acima, que utilizando o operador **E**, a expressão será verdadeira, se e somente se, todas as condições consideradas, resultarem em ***verdadeiro***. Caso contrário, se ao menos uma das condições consideradas resultar em ***falso***, toda a expressão será falsa.

Ex.:

```
algoritmo "OperaçãoE"

var

    i, j : inteiro

    z : real

    resultado : logico

inicio

    i ← 3

    j ← - 5
```

```
    z ← 3.0
    resultado ← ( 10 > 5 ) E ( i > j ) E ( z <> 0 )
    escreva (resultado)
    resultado ← ( 10 > 5 ) E ( i > 2 ) E ( z <> 3 )
    escreva (resultado)
fimalgoritmo
```

Saída:

```
verdadeiro
falso
...
```

Operador *OU*

Operação	Resultado
V *OU* **V**	verdadeiro
V *OU* **F**	verdadeiro
F *OU* **V**	verdadeiro
F *OU* **F**	falso

Tabela 08 – Operador *OU*.

Observe na tabela verdade acima, que utilizando o operador ***OU***, a expressão será verdadeira, se e somente se, dentre todas as condições consideradas, ao menos uma resultar ***verdadeiro***. Caso contrário, se todas as condições consideradas resultar em ***falso***, toda a expressão será falsa.

Ex.:

```
algoritmo "OperaçãoOU"
var
    n, i : inteiro
    x : real
    resultado : logico
```

```
inicio
   n ← 55
   i ← 0
   x ← 4.0
   resultado ← ( i <> 0 ) OU ( x = 0 ) OU ( n < 100 )
   escreva (resultado)
   resultado ← ( i <> 0 ) OU ( x = 0 ) OU ( n > 100 )
   escreva (resultado)
fimalgoritmo
```

Saída:

```
verdadeiro
falso
...
```

Operador *XOU*

Operação	Resultado
V *XOU* **V**	falso
V XOU **F**	verdadeiro
F XOU **V**	verdadeiro
F XOU **F**	falso

Tabela 09 – Operador *XOU*.

Observe na tabela verdade acima, que utilizando o operador ***XOU***, a expressão será verdadeira, se e somente se, uma e apenas uma das condições consideradas resultar ***verdadeiro***. Caso contrário, se mais de uma condição resultar em ***verdadeiro***, toda a expressão será falsa.

Ex.:

```
algoritmo "OperaçãoXOU"
var
   n, i : inteiro
```

```
   x : real
   resultado : logico
inicio
   n ← 55
   i ← 0
   x ← 4.0
   resultado ← ( i <> 0 ) XOU ( x > 0 ) XOU ( n < 100 )
   escreva (resultado)
   resultado ← ( i <> 0 ) XOU ( x = 0 ) XOU ( n < 100 )
   escreva (resultado)
fimalgoritmo
```

Saída:

```
falso
verdadeiro
...
```

Precedência de Operadores

Segue abaixo uma tabela apresentando a ordem de precedência dos operadores utilizados na construção de um algoritmo, bem como sua associatividade. Operadores na mesma linha tem a mesma precedência e as linhas estão ordenadas em ordem decrescente de precedência, por exemplo: "()" tem maior prioridade que "[ ]" que por sua vez tem maior prioridade que "/".

Operador	Associatividade
( ) [ ] { }	esquerda-para-direita
^ * / %	esquerda-para-direita
+ -	esquerda-para-direita
< <= > >=	esquerda-para-direita
= < >	esquerda-para-direita
NÃO	esquerda-para-direita
XOU	esquerda-para-direita

E	esquerda-para-direita
OU	esquerda-para-direita

Tabela 10 – Precedência de operadores.

Considere a expressão:

A ***OU*** B ***XOU*** **FALSO** ***E*** H ***OU*** K

A mesma expressão acima poderia ser escrita da seguinte forma:

(A ***OU*** **(** **(** B ***XOU*** **(** **FALSO** **)** **)** ***E*** H **)** **)** ***OU*** K

. . .

Exercícios:

LABORATÓRIO 00.

LABORATÓRIO 01.

Estruturas de Controle

Estruturas de Decisão

As estruturas de decisão permitem testar condições e executar diferentes instruções dependendo do resultado do teste.

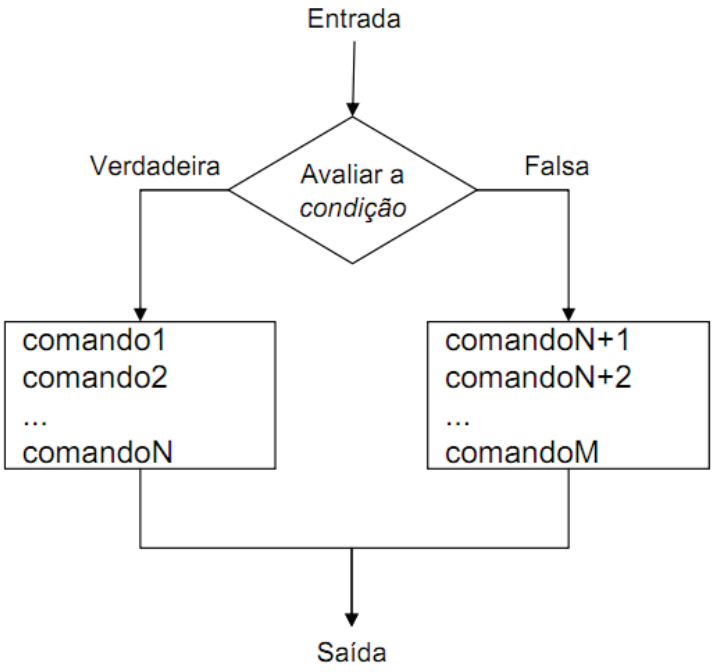

Figura05 – Fluxograma das estruturas de decisão.

Uma condição é uma expressão na qual o resultado é lógico, ou seja, resulta em **verdadeiro** ou **falso**. A partir deste resultado, podemos decidir qual conjunto de instruções (bloco) será executado.

Podemos validar uma condição com as seguintes estruturas, conhecidas como estruturas de decisão:

- **SE**
- **SE-SENAO**
- **SE-SENAO-SE**
- **ESCOLHA**

É importante lembrar que a condição avaliada pelas estruturas de decisão deve sempre resultar em um valor lógico. Caso a expressão não retorne **verdadeiro** ou **falso**, ocorrerá um erro na execução do algoritmo.

```
se   x ← 10   entao   X
    escreva ("x recebe 10")
fimse
```

Esta estrutura não está de acordo com a sintaxe exigida, pois "x ← 10" não é uma condição, ou seja, não retorna verdadeiro ou falso. No exemplo acima é utilizada uma expressão de atribuição e, portanto este trecho do algoritmo resultará em falha.

```
se   x = 10   entao
    escreva ("x é igual a 10")
fimse
```

Este trecho do algoritmo está correto, pois "x = 10" é uma comparação de igualdade, e, portanto resultará em verdadeiro ou falso, logo é uma condição válida para estrutura de decisão.

Vamos analisar com mais detalhes cada uma das estruturas utilizadas para tomadas de decisão no algoritmo.

SE

A estrutura **SE,** valida uma condição para verificar se a execução das instruções em seu bloco será realizada ou não. No caso em que a condição resultar em **verdadeiro,** as instruções serão executadas, caso a condição resulte em **falso**, a execução do algoritmo começa exatamente após o bloco das instruções da estrutura **SE.**

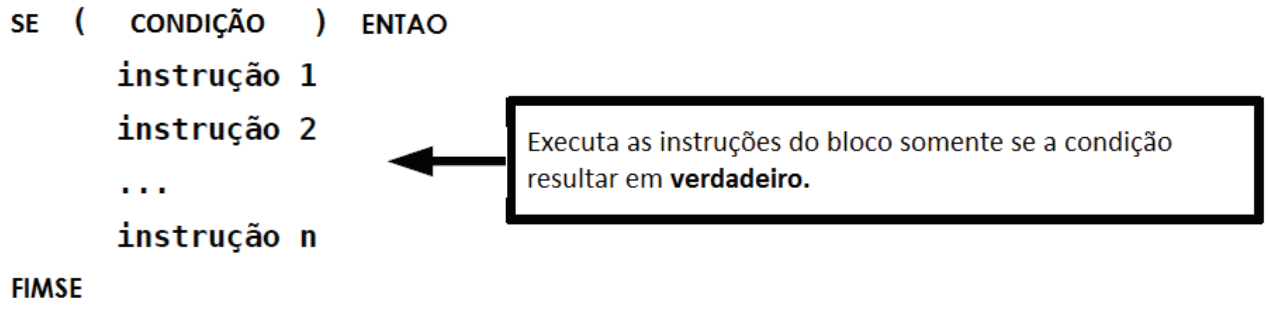

Figura 06 – Estrutura de decisão *SE*.

Veja abaixo, um exemplo da utilização da estrutura **SE**:

```
algoritmo "ValidaEntrada"
var
   idade : inteiro
inicio
   escreva ("Informe sua idade:")
   leia (idade)
   se  idade >= 18  entao
      escreval ("Entrada liberada")
   fimse
  escreva ("Operação realizada!")
fimalgoritmo
```

Caso a condição "idade >= 18" resulte em verdadeiro, ou seja, caso o usuário entre com uma idade maior ou igual a 18, a execução do algoritmo entrará no bloco **SE** e executará a instrução **escreval ("*Entrada liberada*")** , caso contrário, a condição será avaliada como falso, e a execução do algoritmo acontecerá logo após o bloco **SE**, não executando a instrução do bloco. Portanto, a saída do algoritmo será apenas "**Operação realizada!**". Veja como ficariam as duas hipóteses da execução deste algoritmo:

Caso em que "idade >= 18" seja verdade:

Saída: *Entrada liberada*

Operação realizada!

Caso em que "idade >= 18" seja falso:

Saída: *Operação realizada!*

SE-SENAO

Podemos observar acima, que a estrutura **SE** valida apenas se a condição verificada resulta em verdade. Caso isto aconteça o bloco de instruções da expressão é executado. Mas e se quisermos tratar caso a condição retorne falso? A resposta desta pergunta está na estrutura **SE-SENAO**.

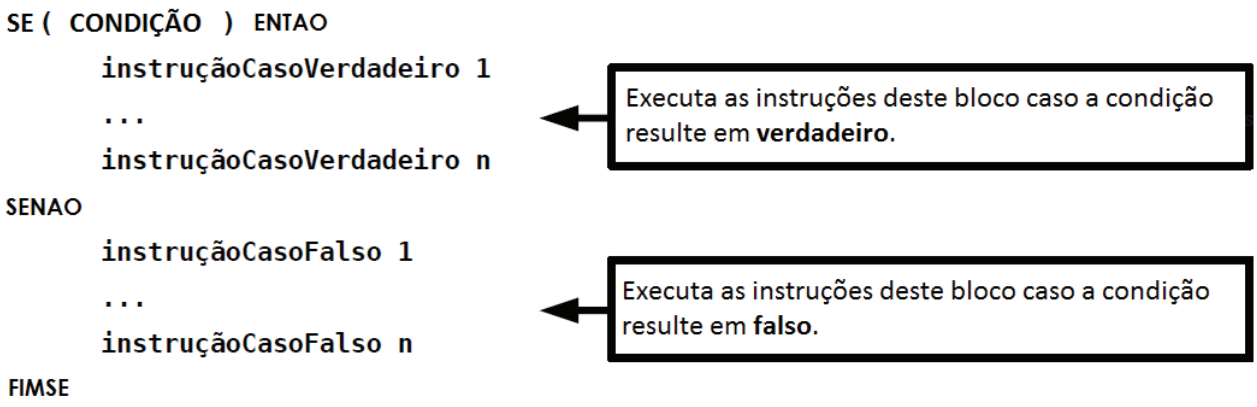

Figura 07 – Estrutura de decisão *SE-SENAO*.

Veja abaixo, um exemplo da utilização da estrutura **SE-SENAO**:

```
algoritmo "ValidaEntrada"
var
   idade : inteiro
inicio
   escreva ("Informe sua idade:")
   leia (idade)
   se  idade >= 18  entao
       escreval ("Entrada liberada")
   senao
      escreval ("Entrada negada")
   fimse
   escreva ("Operação realizada!")
fimalgoritmo
```

Caso a condição "idade >= 18" resulte em verdadeiro, ou seja, caso o usuário informe uma idade maior ou igual a 18, a execução do algoritmo entrará no bloco **SE** e executará a instrução ***escreval ("Entrada liberada")*** , caso contrário, a condição será avaliada como falso, e a execução do algoritmo entrará no bloco **SENAO** e executará a instrução ***escreval ("Entrada negada").***

Após a execução do bloco **SE-SENAO** o algoritmo continua a ser executado normalmente até o seu fim, portanto, após o bloco **SE-SENAO**, a instrução ***escreva ("Operação realizada!")*** é executada.

Veja como ficariam as duas hipóteses da execução deste algoritmo:

Caso em que "idade >= 18" seja verdade:

Saída: *Entrada liberada*

Operação realizada!

Caso em que "idade >= 18" seja falso:

Saída: *Entrada negada*

Operação realizada!

SE-SENAO-SE

A estrutura **SE-SENAO-SE** permite tomar decisões lógicas mais complexas, pois conseguimos através desta estrutura tratar mais detalhadamente as possíveis condições do problema em questão.

Podemos ter várias declarações **SE-SENAO** aninhadas, sendo que a última declaração **SENAO** pode ser omitida; ela é opcional.

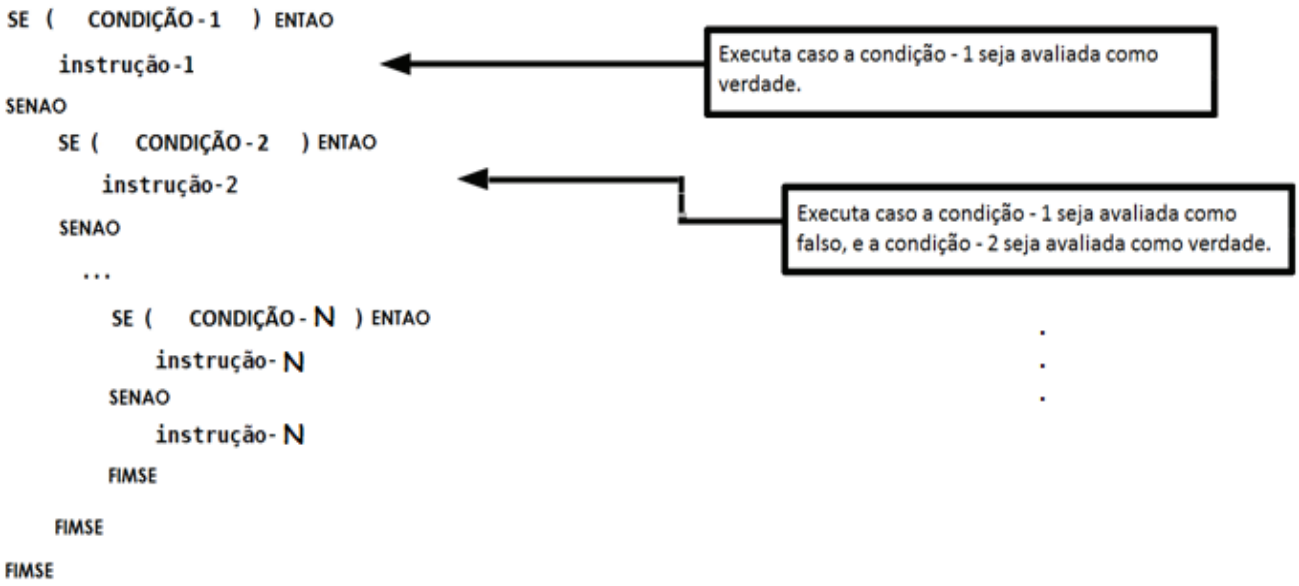

Figura 08 – Estrutura de decisão *SE-SENAO-SE.*

Veja abaixo, um exemplo da utilização da expressão **SE-SENAO-SE**:

```
algoritmo "VerificaAprendizado"
var
   nota : real
inicio
   nota ← 4.9
   se  nota >= 8.0  entao
   .     escreva ("Excelente!")
   senao
   .  se   ( nota < 8.0 )  E  ( nota > 6.0 )  entao
   .  .     escreva ("Bom trabalho!")
   .  senao
   .  .     se  ( nota >= 5.0 )  E  ( nota <= 6.0 )  entao
   .  .  .     escreva ("Regular, estude mais um pouco.")
   .  .     senao
   .  .  .     escreva ("Sua situação é crítica!")
   .  .     fimse
   .  fimse
  fimse
fimalgoritmo
```

Saída:

```
Sua situação é crítica!
...
```

Caso a condição "nota >= 8.0" resulte em verdadeiro, a execução do algoritmo entrará no bloco **SE** da primeira condição e executará a instrução ***escreva*** **("*Excelente*")**, caso contrário, a condição será avaliada como falso, e a verificação será feita na segunda condição. Caso a condição "nota < 8.0 E nota > 6.0" resulte em verdadeiro, a execução do algoritmo entrará no bloco **SE** da segunda condição e executará a instrução ***escreva ("Bom trabalho!")*** , caso contrário, a condição será avaliada como falso, e a verificação será feita na próxima condição, e assim sucessivamente, até que alguma

condição da estrutura seja avaliada como verdadeiro, e suas instruções sejam executadas. Caso nenhuma condição seja avaliada como verdade, o bloco do último **SENAO** será executado, que é o que aconteceu no exemplo citado acima.

Após a execução do bloco **SE-SENAO-SE** o algoritmo continua a ser executado normalmente até o seu fim.

ESCOLHA

A estrutura **ESCOLHA** tem sua lógica bem parecida com a estrutura **SE-SENAO-SE**.

Utilizamos esta estrutura de decisão para verificar o valor exato de uma determinada variável.

Veja abaixo, como funciona a sua sintaxe:

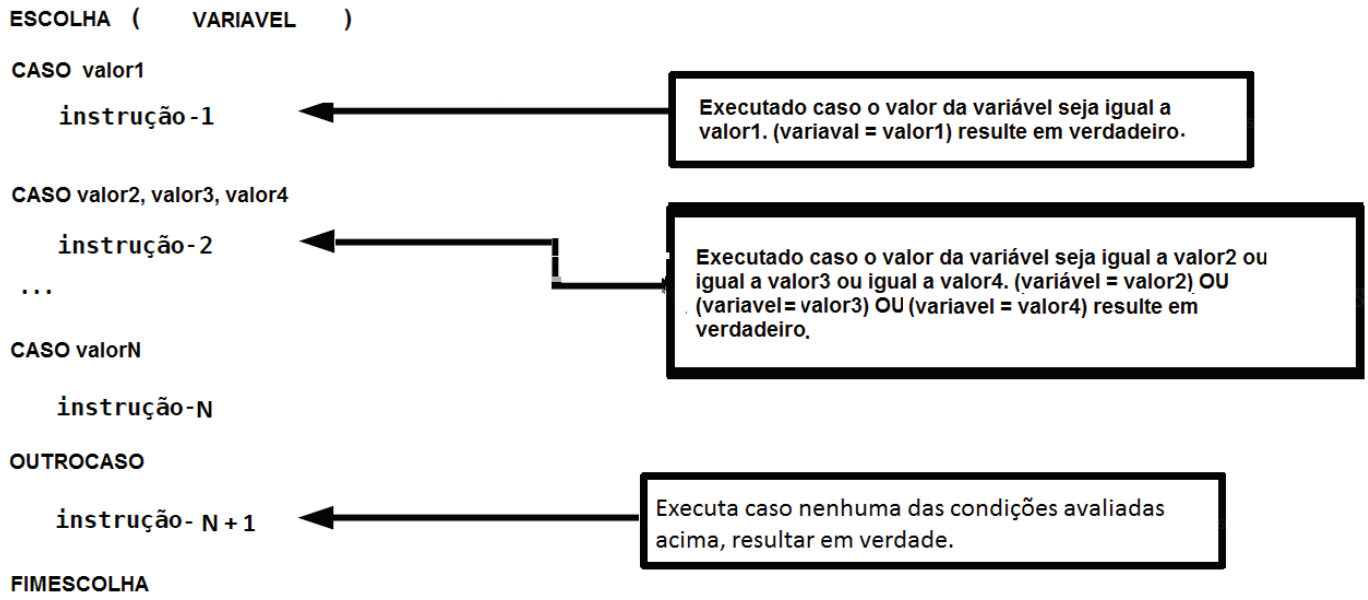

Figura 09 – Estrutura de decisão *ESCOLHA.*

Veja abaixo, um exemplo da utilização da estrutura **ESCOLHA**:

```
algoritmo "VerificaOrigemTime"
var
   time : literal
inicio
   time ← "Corinthians"
```

```
escolha time
    caso "Flamengo", "Fluminense", "Vasco", "Botafogo"
        escreva ("É um time carioca.")
    caso "São Paulo", "Corinthians", "Santos", "Palmeiras"
        escreva ("É um time paulista.")
    outrocaso
        escreva ("É de outro estado.")
fimescolha
fimalgoritmo
```

Saída:

```
É um time paulista.
...
```

Estruturas de Repetição

Nos exemplos anteriores que vimos até agora, sempre foi possível resolver problemas com uma sequência de instruções que necessitam executar apenas uma vez. A combinação das estruturas de decisão e as estruturas de repetição nos permite construir algoritmos para resolução de problemas extremamente complexos.

Uma estrutura de repetição permite que um conjunto de instruções seja executado um número determinado ou indeterminado de vezes, sem que seja necessário escrevê-las mais de uma vez. As instruções serão executadas enquanto uma determinada condição for satisfeita.

Sendo assim, fica simples descobrirmos quando é necessário utilizar estruturas de repetição. Quando observarmos que uma instrução ou um bloco de instruções deve ser executado mais de uma vez, neste ponto é viável utilizar estas estruturas. As estruturas de repetição também são chamadas de Laço ou Loops. São elas:

- **ENQUANTO-FACA**
- **REPITA-ATE**
- **PARA-ATE-FACA**

Vamos analisar com mais detalhes cada uma das estruturas citadas acima.

ENQUANTO-FACA

Na estrutura **ENQUANTO-FACA** as instruções a serem repetidas podem ser executadas nenhuma, uma ou várias vezes, pois a condição a ser verificada, fica no início do bloco.

As instruções serão executadas repetidas vezes enquanto esta condição for satisfeita, ou seja, enquanto essa condição resultar em verdade. Quando no laço, a condição resultar em falso, a repetição é finalizada e o algoritmo continua sua execução após o bloco, até o seu fim.

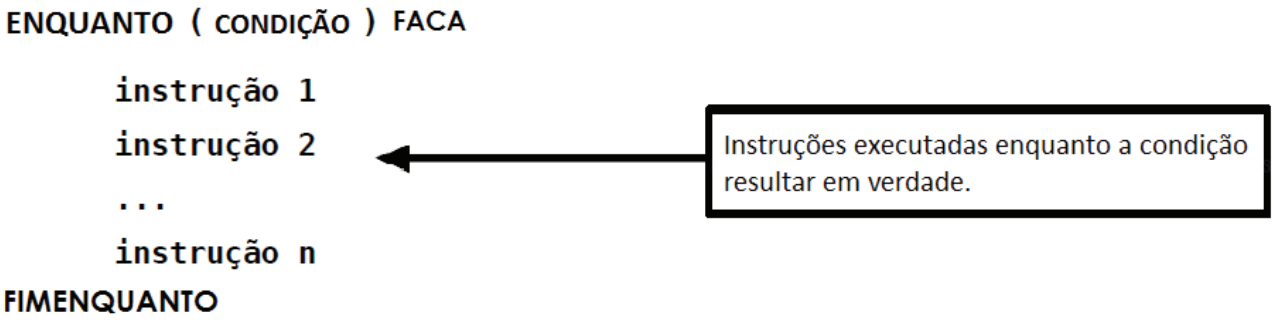

Figura 10 – Estrutura de repetição *ENQUANTO-FAÇA*.

Veja abaixo, exemplos da utilização da estrutura **ENQUANTO-FACA**:

```
algoritmo "LaçoExecutado"
var
   índice : inteiro
inicio
   indice ← 0
   enquanto  índice <= 10  faca
      escreva (índice)
      indice ← indice + 1
   fimenquanto
  escreva ("Fim do algoritmo.")
fimalgoritmo
```

Saída:

```
0 1 2 3 4 5 6 7 8 9 10 Fim do algoritmo.
...
```

A variável **índice** recebe o valor **0** no início do algoritmo. A condição "*índice <= 10*" então é satisfeita, e a execução do bloco **ENQUANTO-FACA** é iniciada.

A cada iteração, ou seja, a cada vez que o bloco é executado, o algoritmo exibe o valor da variável e incrementa seu valor em uma unidade. Isso acontece até que o valor da variável **índice** seja superior a 10 (índice = 11), caso em que a condição não será mais satisfeita , e o algoritmo finalizará a repetição, executando as instruções localizadas logo após o bloco **ENQUANTO-FACA.**

```
algoritmo "LaçoNãoExecutado"
var
   índice : inteiro
inicio
   indice ← 0
   enquanto  índice <> 0  faca
      escreva (índice)
      indice ← índice + 1
   fimenquanto
   escreva ("Fim do algoritmo.")
fimalgoritmo
```

Saída:

```
Fim do algoritmo.
...
```

No exemplo acima, as instruções do bloco **ENQUANTO-FACA** não são executadas, pois de início, a condição que valida a entrada no bloco tem resultado lógico **falso,** condição essa que não permite a sua execução.

Portanto o algoritmo só executa o que se encontra fora da estrutura de repetição.

```
algoritmo "LoopInfinito"
var
   índice : inteiro
inicio
   índice ← 0
   enquanto índice <= 10 faca
      escreva (índice)
   fimenquanto
   escreva ("Fim do algoritmo.")
fimalgoritmo
```

Saída:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 ....
...
```

O algoritmo acima exemplifica um caso muito corriqueiro no desenvolvimento. Quando nos esquecemos de incrementar a variável utilizada para o controle do laço. Esta situação é conhecida como **Loop infinito,** e a repetição do bloco nunca irá parar, pois sempre a condição terá como resultado o valor lógico **verdadeiro,** uma vez que sempre *0 <= 10.*

REPITA-ATE

A instrução **REPITA-ATE** é muito similar a instrução **ENQUANTO-FAÇA.** A única diferença é que **sempre** as instruções do bloco **REPITA-ATE** serão executadas **no mínimo uma vez**.

Inicialmente, as instruções do bloco são executadas e só depois a condição é verificada, pois a condição fica no fim do bloco. Caso a condição resulte em falso, a repetição se inicia e as instruções são executadas repetidas vezes enquanto a condição de parada não for satisfeita. Caso a condição de parada resulte em verdadeiro, as instruções do bloco serão executadas somente uma vez e o algoritmo continuará a executar as instruções localizadas logo abaixo da estrutura de repetição.

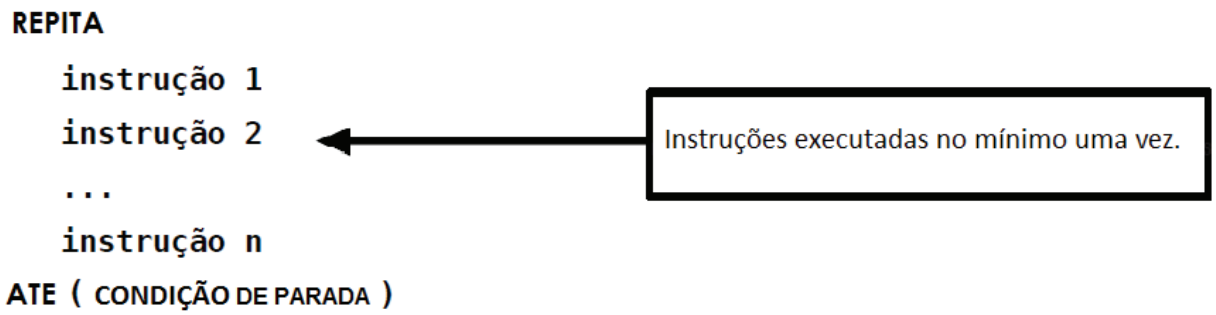

Figura 11 – Estrutura de repetição *REPITA-ATE*.

Veja abaixo, os mesmos exemplos da expressão **ENQUANTO-FACA,** mas utilizando a estrutura **REPITA-ATE**. Observe a diferença entre as duas estruturas de repetição**:**

```
algoritmo "LaçoExecutado"
var
   índice : inteiro
inicio
   indice ← 0
   repita
      escreva (índice)
      indice ← índice + 1
   ate  índice >= 3
  escreva ("Fim do algoritmo.")
fimalgoritmo
```

Saída:

```
0 1 2 Fim do algoritmo.
```

...

```
algoritmo "LaçoExecutadoUmaVez"
var
    indice : inteiro
```

```
inicio
   indice ← 0
   repita
      escreva (índice)
      indice ← indice + 1
   ate  índice = 1
  escreva ("Fim do algoritmo.")
fimalgoritmo
```

Saída:

```
0 Fim do algoritmo.
...
```

No exemplo acima, as instruções do bloco **REPITA-ATE** são executadas apenas uma vez, pois a variável *índice* foi incrementada e seu valor atual é **1**, fazendo com que a condição de parada seja satisfeita.

```
algoritmo "LoopInfinito"
var
   indice : inteiro
inicio
   indice ← 0
   repita
      escreva (índice)
   ate  índice >= 10
  escreva ("Fim do algoritmo.")
fimalgoritmo
```

Saída:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 ....
```

Veja que até mesmo na estrutura **REPITA-ATE** temos que prestar bastante atenção para não nos esquecermos de incrementar a variável de controle, e entrar em um loop infinito.

PARA-ATE-FACA

Veja abaixo a sintaxe utilizada nesta estrutura de repetição:

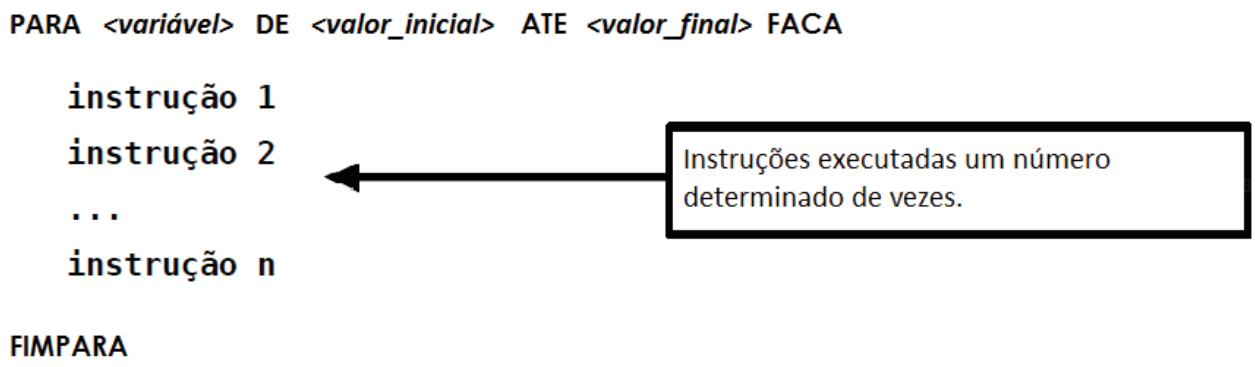

Figura 12 – Estrutura de repetição *PARA-ATÉ-FACA*.

Podemos observar que na estrutura **PARA-ATE-FACA**, temos uma variável de controle que assume valor inicial e final já pré-determinado, ou seja, podemos definir quantas iterações serão feitas naquele bloco de instruções.

Veja um exemplo da utilização desta estrutura de repetição:

```
algoritmo "Tabuada"
var
   i, tab : inteiro
inicio
   tab ← 5
   escreval ("Tabuada do: ", tab )
   para i de 1 ate 10 faca
      escreval ( i , " * " , tab , " = " , (i * tab) )
   fimpara
fimalgoritmo
```

Saída:

```
1 * 5 = 5
2* 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
6 * 5 = 30
7 * 5 = 35
8 * 5 = 40
9 * 5 = 45
10 * 5 = 50
```

...

Observe que utilizamos o operador "," para concatenar duas ou mais variáveis do tipo **caractere** ou **literal**, ou seja, juntar dois textos.

Ex.:

"a" , "b" , "c" = abc

Comando interrompa

As três estruturas de repetição citadas acima permitem o uso da instrução *interrompa,* que causa a saída imediata do laço.

Veja abaixo a utilização desta instrução na estrutura **REPITA-ATE.**

```
algoritmo "UtilizandoInterrompa"
 var
  índice : inteiro
 inicio
   índice ← 1
   repita
        escreval ("Repetição " , índice)
        índice ← índice  + 1
        se  índice = 4  entao
            interrompa
```

```
    fimse
  ate  índice >= 10
  escreva ( "Fim do algoritmo" )
fimalgoritmo
```

Saída:

```
Repetição 1
Repetição 2
Repetição 3
Fim do algoritmo
...
```

. . .

Exercícios:

LABORATÓRIO 02.

Vetores

O que são vetores?

Vetores, também conhecidos como Arrays, são variáveis que armazenam um conjunto de valores do mesmo tipo de forma sequencial na memória. São coleções homogêneas de dados. Por exemplo, se tivermos que criar 20 variáveis do tipo inteiro, poderíamos utilizar um vetor de inteiros com 20 posições.

Um vetor pode ser representado da seguinte forma:

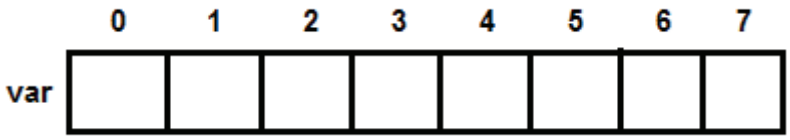

Figura 12 – Representação de um vetor.

Cada posição do vetor é indicada por um índice, onde, a primeira posição do vetor possui índice 0, a segunda posição índice 1, a terceira posição índice 2 e assim por diante. O espaço alocado na memória para o vetor depende do tipo dos valores que serão armazenados nele.

A utilização de vetores envolve três etapas:

- Declarar o vetor.
- Definir seu tamanho, para alocação na memória.
- Armazenar valores no vetor.

Primeiramente vamos aprender a como declarar um vetor.

Declarando um Vetor

Podemos tratar um vetor, ou um array, como uma coleção homogênea de dados, considerada a mais simples das estruturas de armazenamento. O tamanho de um vetor é fixo, ou seja, não se altera no decorrer da execução do algoritmo.

A declaração de um vetor em Português Estruturado é feita seguindo a sintaxe abaixo:

nome_vetor : **vetor[** posições **] de** tipo_dados_armazenados

Assim temos como exemplo de declaração de vetores:

```
arr : vetor[0..100] de inteiro

x : vetor[0..50] de real

a : vetor[0..2] de logico

vet : vetor[0..10] de literal
```

Após a declaração do vetor, podemos armazenar valores em suas posições conforme o seu tipo.

Acessando um elemento do Vetor

Como vimos, um vetor é uma coleção de dados homogêneos, ou seja, do mesmo tipo, alocados na memória. Para acessarmos um local específico dessa memória devemos indicar entre os colchetes a posição desejada no vetor, que chamamos de índice.

É importante saber, que não interessa a quantidade de posições que o vetor tem, ou seja, o tamanho do vetor, ele sempre começará no índice 0 e terminará na posição (tamanho – 1). Por exemplo, um vetor com 20 posições, começará no índice 0 e terminará no índice 19.

O armazenamento de um valor em uma determinada posição do vetor pode ser feito da seguinte forma:

nomeVetor [índice] <-- valor

Assim podemos armazenar o valor 10 na primeira posição do vetor, com a seguinte instrução:

```
arr[0] ← 10
```

Veja abaixo um algoritmo que preenche um vetor de 5 posições:

```
algoritmo "PopularVetor"

var

   vetorInteiros : vetor[0..4] de inteiro

inicio

   vetorInteiros[0] ← 1
```

```
    vetorInteiros[1] ← 2
    vetorInteiros[2] ← 3
    vetorInteiros[3] ← 4
    vetorInteiros[4] ← 5
fimalgoritmo
```

Observe que quando o vetor tem poucas posições, podemos desenvolver uma sequência simples de instruções para preenchê-lo. Mas, suponha que o vetor tenha 1000 posições. Seria inviável escrever 1000 instruções do tipo "*vetor [índice] ← valor*".

Para aperfeiçoar esta solução podemos utilizar uma estrutura de repetição, como por exemplo, a estrutura PARA-ATE-FAÇA.

```
algoritmo "PopularVetor"
var
    vetorInteiros : vetor[0..999] de inteiro
    índice : inteiro
    valor : inteiro
inicio
    valor ← 1
    para índice de 0 ate 999 faca
        vetorInteiros[índice] ← valor
        valor ← valor + 1
    fimpara
fimalgoritmo
```

Para concluirmos nosso estudo sobre vetor, veja um exemplo de algoritmo que utiliza esta estrutura para solucionar o problema abaixo:

Elaborar um algoritmo que lê um conjunto de 30 valores e os coloca em 2 vetores conforme estes valores forem pares ou impares. O tamanho dos vetores são de 5 posições. Se algum vetor estiver cheio, escrevê-lo. Terminada a leitura escrever o conteúdo dos dois vetores. Cada vetor pode ser preenchido tantas vezes quantas forem necessárias.

```
algoritmo "ExemploVetor"
var
   vetorPares : vetor[0..4] de inteiro
   vetorImpares : vetor[0..4] de inteiro
   i, valor, indicePar, indiceImpar, auxPar, auxImpar : inteiro
 inicio
   indicePar ← 0
   indiceImpar ← 0
   para  i  de  1  ate  30  faca
         leia ( valor )
         se  valor % 2 = 0  entao
               vetorPares[indicePar] ← valor
               se  indicePar > 3  entao
                     para  auxPar  de  0  ate  4  faca
                           escreva ("  " , vetorPares[auxPar] , "  " )
                           se  auxPar  =  4  entao
                              escreval ( )
                           fimse
                     fimpara
                     indicePar ←0
               senao
                     indicePar ← indicePar + 1
               fimse
         senao
               vetorImpares[indiceImpares] ← valor
               se  indiceImpar > 3  entao
```

```
                para auxImpar de 0 ate 4 faca
                        escreva ( " " , vetorImpares[auxImpar] , " " )
                        se auxImpar = 4 entao
                           escreval ( )
                        fimse
                fimpara
                indiceImpar ← 0
            senao
                indiceImpar ← indiceImpar + 1
            fimse
        fimse
    fimpara
fimalgoritmo
```

Matriz

O que são matrizes?

Matrizes são vetores multidimensionais, ou seja, não são lineares e sim geométricos. Enquanto um vetor tem apenas uma linha com vários valores, em uma matriz podemos ter várias linhas com vários valores, que convenientemente dividimos em linhas e colunas.

Didaticamente tratamos um vetor multidimensional com 2 dimensões, como uma matriz convencional, que provém da álgebra linear. Por exemplo, uma matriz com 3 linhas e 2 colunas pode ser representada da seguinte forma:

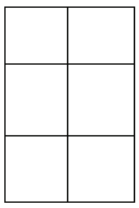

Figura 13 – Matriz 3x2.

Mas sua disposição na memória não é essa. Na verdade, uma matriz, ou um vetor multidimensional nada mais é que um vetor capaz de armazenar outros vetores, ou seja, em cada posição de um vetor, temos outro vetor armazenado.

O mesmo vetor multidimensional apresentado anteriormente pode ser representado da seguinte forma:

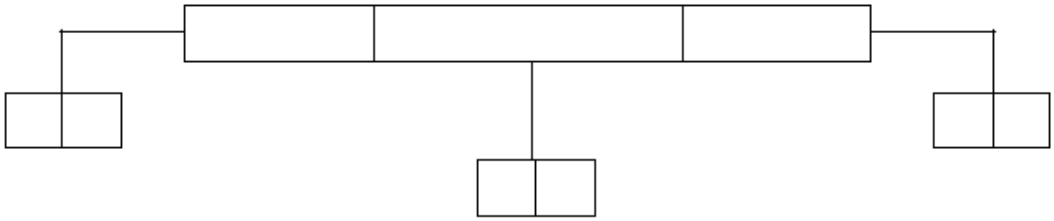

Figura 14 – Matriz 3x2.

Nesta representação temos 2 dimensões, onde a primeira dimensão do vetor, com 3 posições, armazena endereços de memória que estão referenciando outros vetores com 2 posições cada.

É importante entender que os valores que serão armazenados nesta estrutura serão alocados apenas na última dimensão, ou seja, na matriz representada acima, poderíamos armazenar 6 valores (3 x 2), todos na segunda dimensão, que neste caso é a última. Na primeira dimensão não podemos armazenar valores, pois ela está armazenando outros vetores, que por sua vez estão armazenando os valores de fato.

Veja como ficaria então um vetor com 3 dimensões:

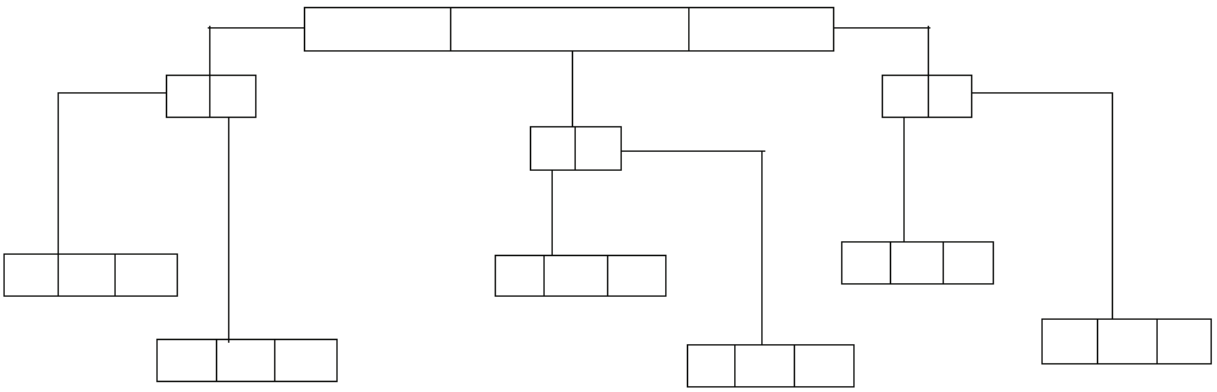

Figura 15 – Matriz 3x2x3.

Com isso, fica simples então entender como funciona um vetor multidimensional com N dimensões.

Agora, entendido o funcionamento de uma matriz, devemos aprender como acessar suas posições, para preenchê-la e manipulá-la.

Declarando uma Matriz

Matrizes são arranjos ordenados que ao contrário dos vetores, podem ter N dimensões, sendo que estas dimensões lhe dão o nome N-dimensional. Uma matriz de duas dimensões será chamada bi-dimensional, uma de três dimensões tri-dimensional e assim consecutivamente.

Declarar uma matriz ou um vetor multidimensional com N dimensões, funciona praticamente da mesma forma que um vetor unidimensional.

Por exemplo, a declaração de um vetor de 2 dimensões, mais conhecida como uma matriz, pode ser feita da seguinte forma:

nome_da_matriz : **vetor** [*tamanhoDimensao1* , *tamanhoDimensao2*] **de** *tipo_da_matriz*

Assim temos como exemplos de declarações:

```
mat : vetor[0..3 , 0..2] de inteiro

matrix : vetor[0..3 , 0..5] de real

x : vetor[0..4, 0..3] de logico
```

Após declararmos um vetor multidimensional, devemos aprender como acessar suas posições, para poder manipulá-lo.

Acessando um elemento da Matriz

Como vimos, um vetor multidimensional nada mais é do que vetores armazenando vetores, e por conveniência e didática, denominamos como matriz um vetor bi-dimensional.

Para efeitos de estudo, nos limitaremos somente as matrizes (vetores com 2 dimensões).

Assim como nos vetores unidimensionais, utilizamos índices para acessar determinada posição da matriz, separando com vírgula o índice para acessar sua respectiva dimensão.

É importante lembrar novamente, que os valores de fato estão armazenados todos na ultima dimensão, pois as anteriores armazenam apenas outros vetores.

Portanto, considere a matriz especificada abaixo:

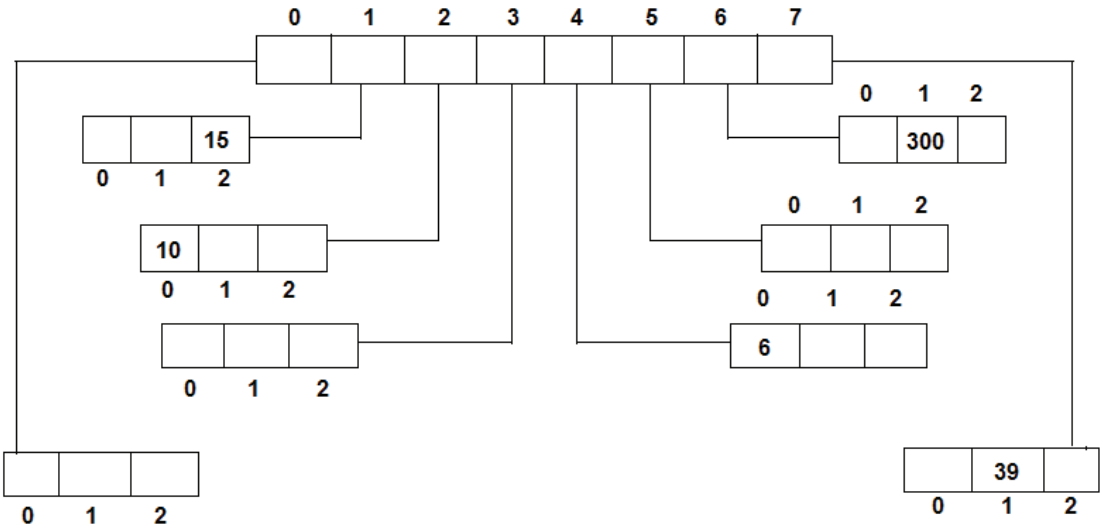

Figura 16 – Matriz 7x3.

Para construirmos uma matriz como essa, poderíamos executar o seguinte algoritmo:

```
algoritmo "PopularMatriz"
var
   matriz : vetor[0..6 , 0..2] de inteiro
inicio
   matriz[1, 2] ← 15
   matriz[2,0] ← 10
   matriz[4,0] ← 6
   matriz[6,1] ← 300
   matriz[7,1] ← 39
fimalgoritmo
```

Novamente, suponha que temos uma matriz com N posições na primeira dimensão e M posições na segunda. Com N e M assumindo valores pequenos, o algoritmo acima se torna viável, caso contrário, como ficaria o algoritmo?

A resposta novamente seria a utilização de uma estrutura de repetição.

```
algoritmo "ExemploMatriz"
```

```
var
   matriz  : vetor[0..N , 0..M] de inteiro
   linha, coluna : inteiro
   valor : inteiro
inicio
   valor ← 1
   para linha de  0  ate  N  faca
        para coluna de 0 ate M faca
           matriz[linha, coluna] ← valor
           valor ← valor + 1
        fimpara
   fimpara
fimalgoritmo
```

Dessa forma, concluímos que para cada dimensão temos uma estrutura aninhada do tipo PARA-ATÉ-FAÇA, para percorrer suas respectivas posições.

. . .

Exercícios:

LABORATÓRIO 03.

Funções

Métodos

Conforme um problema se torna mais complexo, surge uma série de situações a serem resolvidas para que este problema possa ser solucionado. Podemos dizer que passamos a ter dentro deste problema uma série de pequenos probleminhas. Muitas vezes, esta grande quantidade de pequenos probleminhas afeta a legibilidade (clareza) de um algoritmo, fazendo com que uma consulta ou manutenção futura da lógica aplicada a ele seja uma tarefa difícil de se realizar. Com a modularização é possível evitar esta situação.

A **modularização** é a técnica de quebrar um problema em pequenas partes, sendo que cada uma destas partes terá seu papel bem definido na execução do algoritmo.

A estas pequenas partes, damos o nome de **métodos**, que são subprogramas a serem executados no decorrer do fluxo de execução do algoritmo principal, para que o problema seja solucionado.

Um método tem a função de auxiliar o programa principal através da realização de uma determinada subtarefa. Os métodos são chamados de dentro do programa principal como se fossem instruções. Após seu término, a execução continua a partir do ponto onde ele foi chamado. É importante compreender que a chamada de um método simplesmente gera um **desvio provisório do fluxo de execução do algoritmo.**

Na construção de um algoritmo um método deve sempre ser declarado antes do algoritmo principal, e pode ser de dois tipos: **função** ou **procedimento.**

Uma função, além de executar uma determinada tarefa, retorna um valor para quem o chamou, que é o resultado da sua execução. Já um procedimento não retorna nenhum valor.

Cada método, além de ter acesso às variáveis do algoritmo que o chamou (são as variáveis globais), pode ter também suas próprias variáveis (são as chamadas variáveis locais), que existem apenas durante sua execução.

Ao se chamar um método, também é possível passar-lhe determinados valores que recebem o nome de parâmetros (são valores que na linha de chamada, ficam entre os parênteses e que estão separados por vírgula, quando do mesmo tipo, e por ponto-e-vírgula, quando de tipos diferentes). A quantidade de parâmetros, sua ordem e respectivos tipos não podem mudar, devem estar de acordo com o que foi especificado na sua correspondente declaração.

Veja a seguir qual é a sintaxe a ser utilizada na declaração de um **procedimento:**

```
procedimento { nome do método }( lista de parâmetros )
var
   {declaração de variáveis locais}
inicio
   {instruções do método}
fimprocedimento
```

Ex.:

```
procedimento  soma( x, y : inteiro  ;  w : real )
var
   soma : real
inicio
   soma ← x + y + w
   escreva (soma)
fimprocedimento


algoritmo "SomaVariaveis"
inicio
    soma(10 , 10 , 2.5)
fimalgoritmo
```

Saída:

```
22.5
...
```

Vejamos agora como é a sintaxe a ser utilizada na declaração de uma **função:**

```
funcao { nome do método }( lista de parâmetros ) : tipo do retorno
var
   {declaração de variáveis locais}
inicio
   {instruções do método}
fimfuncao
```

Ex.:

```
funcao  soma( x, y : inteiro  ;  w : real ) : real
var
inicio
   retorne   x + y + w
fimfuncao


algoritmo "SomaVariaveis"
var
    resultado : real
inicio
    resultado ← soma(10 , 10 , 2.5)
   escreva (resultado)
fimalgoritmo
```

Saída:

```
22.5
...
```

Assinatura

Todo método definido possui uma assinatura que garante que não haja dois métodos iguais no programa. Quando passamos a instrução para executar um determinado método, o algoritmo sabe qual método executar pela sua assinatura.

Os elementos do método que fazem parte da sua assinatura são:

- nome do método.
- quantidade de parâmetros existentes.
- tipo de cada parâmetro.
- ordem desses parâmetros.

Obs.: o tipo de retorno de um método não faz parte de sua assinatura.

Recursividade

Recursividade é uma técnica aplicada a métodos para que este possa chamar a si próprio. Um método que invoca ele mesmo é chamado de método recursivo.

Todo cuidado é pouco ao se construir métodos recursivos. Um método recursivo tem que seguir duas regras básicas:

- Ter uma condição de parada.
- Tornar o problema mais simples.

A primeira coisa a se providenciar é um critério de parada. Este critério vai decidir quando o método vai parar de invocar ele mesmo, o que impede que o método se chame infinitas vezes.

Um método que calcule o fatorial de um número inteiro N é um bom exemplo de uma função recursiva. Vejamos um método não recursivo e um recursivo para solução deste problema, a fim de concluirmos esta definição.

Não recursivo

```
procedimento  calculaFatorial( N : inteiro )

var

   fatorial, cont : inteiro
```

```
inicio
   fatorial ← 1
   para cont de 1 ate N faca
      fatorial ← fatorial * cont
   fimpara
   escreva ( fatorial )
fimprocedimento
```

Recursivo

```
funcao calculaFatorial( N : inteiro ) : inteiro
inicio
   se N = 1 entao
      RETORNE 1
   senao
      retorne N * calculaFatorial ( N – 1 )
   fimse
fimfuncao
```

Observe que utilizando a técnica da recursividade o algoritmo fica mais conciso e mais bem elaborado. Veja a ilustração abaixo da execução recursiva do método; considere N = 4.

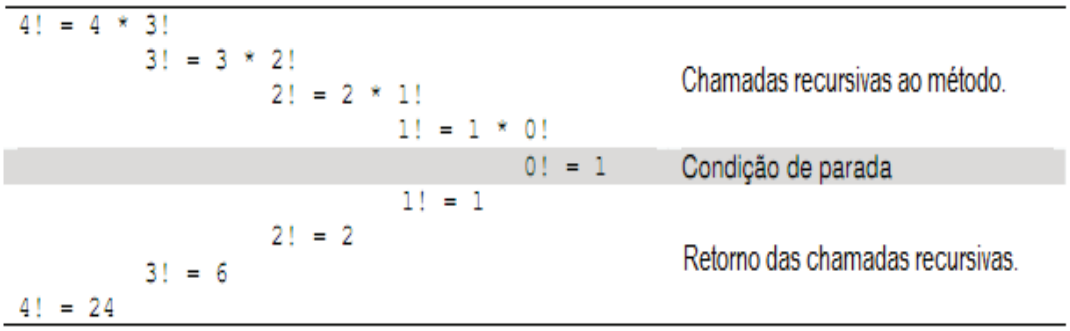

Figura 17 – Representação gráfica do fatorial de 4.

Paradigma Orientação a Objetos

Programação Orientada a Objetos

O termo Programação Orientada a Objetos foi criado por Alan Kay, autor da linguagem de programação *Smalltalk.*

Mesmo antes da criação do Smalltalk, algumas das idéias defendidas pela orientação a objetos já eram aplicadas, sendo que a primeira linguagem de programação a utilizar estas idéias foi a linguagem *Simula 67,* criada por Ole Johan Dahl e Kristen Nygaard em 1967. Observe que este modelo de programação é bastante antigo, sendo que só agora vem sendo aceito realmente pela comunidade de desenvolvimento de software.

Algumas das vantagens que despertou este grande interesse pelas linguagens de programação, são:

- Melhor representação do mundo real.
- Maior ênfase nos dados.
- Facilidade na modularização do problema.
- Facilidade na reutilização de código.
- Facilidade na manutenção e extensão do código.

A programação orientada a objetos foi criada para tentar aproximar o mundo real do mundo virtual, através da abstração dos dados. Nesse contexto, a abstração refere-se à capacidade de modelar o mundo real, pelo qual restringimos o nosso universo de análise, desprezando as informações que são irrelevantes para a solução do problema. Podemos demonstrar o uso de abstração facilmente, quando fechamos os olhos e pensamos em uma mesa.

Esta mesa imaginária provavelmente não vai ser igual à outra imaginada por outras pessoas, mas o que importa é que todas as pessoas que imaginaram uma mesa colocaram nessa as informações que para elas são necessárias para a sua função (de ser uma mesa). Não importa se a mesa é de três pés ou quatro, ou se o tampão é de vidro, madeira ou mármore; o que importa é que a imagem que idealizamos em nossa cabeça é de uma mesa que tem as informações necessárias para cumprir sua função. Contudo, a idéia fundamental é tentar simular o mundo real dentro do computador.

Na programação orientada a objetos, o desenvolvedor é responsável por moldar o mundo dos objetos, dizendo também como estes irão se relacionar. Para isso, implementa-se um conjunto de

classes que definem os objetos presentes no sistema. Cada classe determina o comportamento (definido nos métodos) e as propriedades (atributos) de seus objetos, assim como o relacionamento com outros objetos.

Suponha que estamos construindo um software para gerenciar uma sala de aula. A partir deste cenário, podemos criar classes como: Pessoa, Aluno, Professor, Computador, e assim por diante. A solução do problema é dada pelo desenvolvedor, então este é responsável pela criação dos objetos e relacionamentos, sendo dele a decisão de quantos e quais objetos devem ser criados para a simulação do cenário real no mundo virtual. Cada classe citada tem propriedades bem definidas, como por exemplo, uma Pessoa tem um nome, uma idade, número de CPF; um Aluno tem matricula, nota. Esta modelagem é responsabilidade do desenvolvedor e é resultado da abstração que ele teve do problema. Para se criar um objeto, devemos atribuir valores as estas propriedades, por exemplo: um objeto do tipo Pessoa, tem nome igual a *Bruno Zafalão*, idade igual a *22* e número de CPF é igual a *999.999.999-99*.

Antes de começarmos a utilizar uma linguagem orientada a objetos, devemos especificar mais alguns conceitos fundamentais deste paradigma.

Classes

Uma classe define um objeto. A classe é o "esqueleto" de um objeto. Através dela definimos quais propriedades e comportamentos seus objetos terão, mas sem atribuir valores a estas estruturas, pois ao atribuir valores, estaremos criando objetos desta classe, assunto tratado nas próximas seções.

As propriedades de uma classe serão representadas através de variáveis, e os comportamentos do objeto serão representados através de métodos.

Um método nada mais é que um procedimento ou função, que executa um número determinado de instruções para alcançar seu objetivo.

A definição das classes que irão compor o sistema e seus inter-relacionamentos é o principal resultado da etapa de Concepção.

Em geral, o resultado desta definição é expresso através de uma linguagem de modelagem, tal como a *Unified Modeling Language,* a UML.

Utilizando a UML, uma classe é tipicamente expressa na forma gráfica, composta por três regiões: o nome da classe, as propriedades da classe e os métodos (comportamentos) da classe.

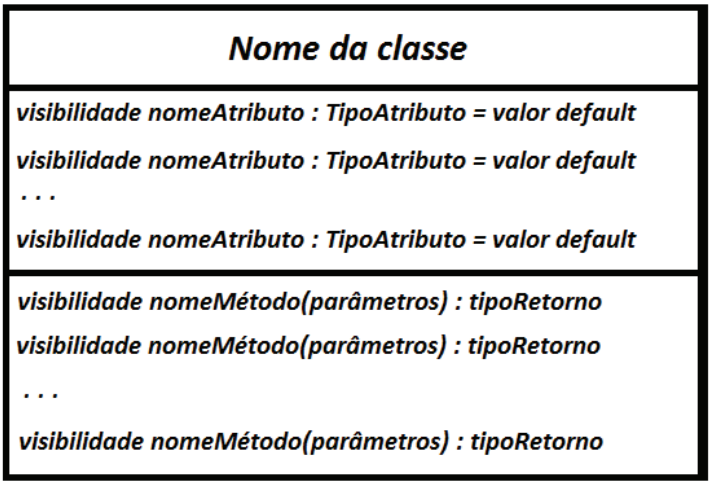

Figura 18 – Definição UML de uma classe.

O **nome da classe** é o identificador desta classe. Através de seu nome, conseguimos referenciá-la posteriormente na execução do código, por exemplo, na criação de um objeto desta classe.

Os **atributos** representam as propriedades da classe. Cada atributo é identificado por um **nome** e um **tipo** associado, ou seja, são **variáveis da classe**. A maior parte das linguagens de programação orientadas a objetos oferecem tipos primitivos de dados, que são os tipos inteiros, real, caracteres, dentre outros, que podem ser usados na declaração de um atributo (variável). Cada atributo possui ainda um **valor default,** que é um valor inicial para a variável, ou seja, o atributo ao ser declarado já recebe um valor para armazenar. Este valor é opcional.

Os **métodos** definem o comportamento da classe, suas funcionalidades, ou seja, o que os objetos desta classe conseguem fazer. O que especifica um método é a **assinatura do método**, composto pelo nome do método e a lista de parâmetros que ele espera.

Atributos e métodos de uma classe são conhecidos como **membros da classe.**

O **modificador de acesso** pode ser aplicado tanto aos atributos, quanto aos métodos. Em geral, podem ser especificados em três tipos:

- **público**: Denotado na UML pelo símbolo "+", indica que este membro pode ser acessado por qualquer outro objeto.

- **privado:** Denotado na UML pelo símbolo "-", indica que este membro pode ser acessado apenas pela classe onde eles foram declarados.

- **protegido:** Denotado na UML pelo símbolo "#", indica que o membro pode ser acessado por outros objetos criados de uma classe armazenada no mesmo pacote (diretório), ou por outros objetos que não se encaixam nesta regra, através de um mecanismo conhecido como herança.

Abaixo, temos um exemplo da definição da classe Pessoa utilizando a Linguagem de Modelagem Unificada (UML), e sua codificação utilizando a linguagem de programação orientada a objetos Java, para demonstrar os conceitos definidos até o momento:

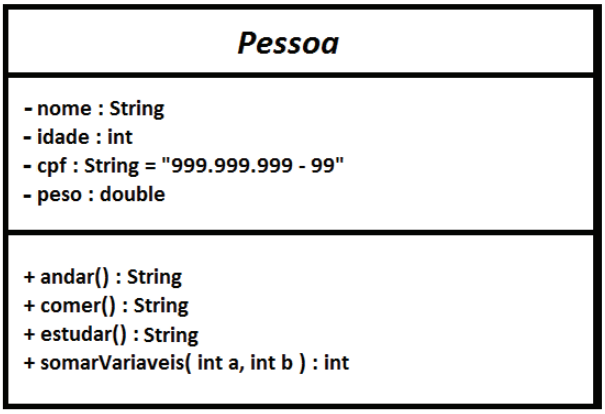

Figura 19 – Definição UML da classe Pessoa.

Codificação em Java:

```java
public class Pessoa {

    private String nome;

    private int idade;

    private String cpf = "999.999.999 - 99"

    private double peso;

    public String andar() {
        return "já estou andando...";
    }

    public String comer() {
        return "já estou comendo...";
    }

    public void estudar() {
        return "já estou estudando...";
    }

    public int somarVariaveis( int a, int b) {
        return a + b ;
    }

    // métodos getters e setters.

```

Figura 20 – Codificação da classe Pessoa em Java.

É importante lembrar, que nosso objetivo neste curso é dominar a lógica do problema e representa-lo em um algoritmo. A codificação será feita após essa etapa estar finalizada, o que será matéria para um próximo curso. Portanto, você deve estar se perguntando: mas como representar a UML da classe Pessoa em um algoritmo?

Se tratando de algoritmo, já sabemos que a sintaxe a ser seguida não é fundamental, porém utilizando um padrão para sua construção, a legibilidade e entendimento da sua função se torna uma tarefa bem mais simples. O importante é deixar claro a lógica que deve ser seguida para a solução do problema.

Para a representação da classe Pessoa em um algoritmo, podemos dividi-la de acordo com suas funções, seus comportamentos, onde teríamos um algoritmo para cada método ou função, sendo que no momento da codificação juntaríamos todos estes na definição de uma só classe.

Observe na definição da classe em Java, que omitimos alguns métodos denominados métodos getters e setters. Por que fizemos isso, e de onde saíram estes métodos?

A resposta para esta pergunta é: Encapsulamento.

Encapsulamento

O conceito de encapsulamento está intimamente ligado ao conceito de abstração. É a técnica que permite separar a parte funcional da classe de suas propriedades e definições.

O encapsulamento é uma das técnicas mais utilizadas na programação orientadas a objetos. Encapsular, na programação, significa esconder os dados das propriedades de uma classe de seus utilizadores, pois é responsabilidade da classe manter seus dados íntegros, sem alterações não previstas no fluxo da aplicação.

Para implementar o encapsulamento em uma classe, ou seja, para encapsular os dados de uma classe, devemos manter os atributos da classe como privados, e criar métodos getters, para obter e setters, para alterar o valor destes atributos.

O método getter para obter o valor do atributo, tem a seguinte sintaxe:

```
public tipoAtributo getNomeAtributo() {
    return atributo ;
}
```

E o método setter para alterar o valor do atributo, segue a sintaxe:

```
public void setNomeAtributo ( tipoAtributo parametro ) {
    atributo = parametro;
}
```

Na codificação da classe pessoa, os métodos getter e setter do atributo *nome*, por exemplo, poderiam ser implementados da seguinte forma:

```
1   public String getNome() {
2       return nome;
3   }
4
5   public void setNome(String n) {
6       nome = n;
7   }
```

Figura 21 – Codificação dos métodos getter e setter em Java.

O método setter tem tipo de retorno **void,** o que significa que o método não retorna nenhum valor, apenas executa um determinado número de instruções. É fácil observar que o método setNome() não possui a instrução *return*, portanto não tem retorno, é void.

Um exemplo prático da grande vantagem do encapsulamento pode ser mostrado quando queremos atribuir valores a variável *idade*.

Sem encapsular os dados de uma classe, suas propriedades serão públicas, logo poderíamos executar a seguinte instrução:

idade = -5;

O que no mundo real, seria uma situação impossível, logo a lógica do meu sistema possui uma falha.

Se por outro lado aplicarmos a técnica do encapsulamento, este atributo não poderia ser acessado diretamente, como na instrução acima, pois este seria privado, ou seja, nenhum objeto que utiliza esta classe tem acesso à ele, somente através dos métodos getters e setters.

Como o nosso objetivo é atribuir um valor a este atributo, devemos invocar o método setter, que em sua execução poderíamos validar o valor que será atribuído a *idade*. Este método poderia implementar o seguinte algoritmo :

```
procedimento setIdade ( parametro : inteiro)
var
inicio
   se parametro < 0 entao
      idade ← 0
   senão
```

```
    idade ← parâmetro
  fimse
fimprocedimento
```

Dessa forma, conseguimos impedir que um valor negativo fosse atribuído a variável *idade.*

Resumindo, o encapsulamento então, tem como objetivo garantir que a responsabilidade de manter íntegro o valor das propriedades de uma classe, seja apenas da classe, impedindo que outros objetos possam acessar diretamente estas propriedades. Este objetivo é alcançado mantendo os atributos da classe privados, com métodos getters e setters públicos para obter e alterar seus valores.

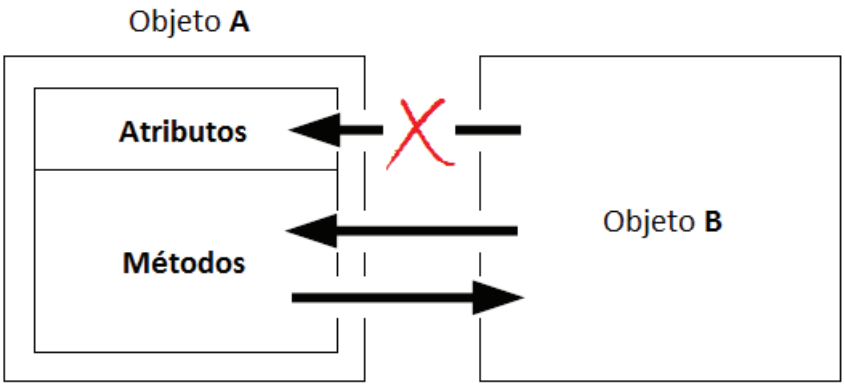

Figura 22 – Encapsulamento.

Definido o que são classes e como utilizar o encapsulamento, precisamos definir o elemento mais importante da programação orientada a objetos, os objetos.

Objetos

O que caracteriza a programação orientada a objetos são os objetos. Eles são a chave para a implementação de sistemas complexos e eficientes.

De um modo geral, podemos encarar os objetos como sendo os objetos físicos do mundo real, como: carro, cachorro, computador, avião, telefone, aluno, pessoa, etc..., que através da linguagem, é representado no mundo virtual, onde possuem propriedades (atributos) e comportamentos (métodos) bem definidos.

Um objeto é uma instancia de uma classe. É a classe que possui a definição de como será aquele objeto, quais suas propriedades e operações. Ao atribuir valores a estas propriedades estamos instanciando a classe, ou seja, criando um objeto.

Da mesma forma que as variáveis, um objeto quando criado irá ocupar um espaço na memória, para armazenar seu estado (valores dos atributos da classe) e o conjunto de operações que podem ser aplicadas a ele (os métodos definidos na classe).

No paradigma de orientação a objeto, tudo pode ser potencialmente representado por um objeto, que por sua vez é capaz de interagir com outros objetos através de "trocas de mensagens", compondo assim um programa orientado a objetos. Na prática, esta "troca de mensagens" traduz-se na invocação de métodos, ou seja, um objeto se comunica com outro, invocando seus métodos.

Utilizando uma linguagem de modelagem, como a UML, podemos representar um objeto da seguinte forma:

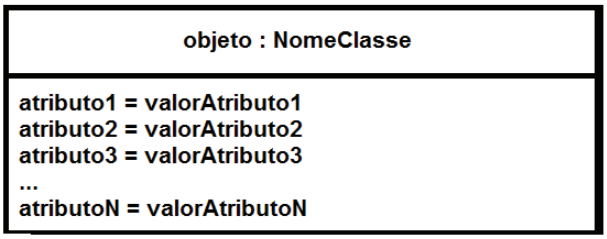

Figura 23 – Definição UML de um objeto.

Um exemplo concreto desta forma de representação poderia ser representar um objeto da classe *Pessoa,* com identificador igual a pessoa1**.**

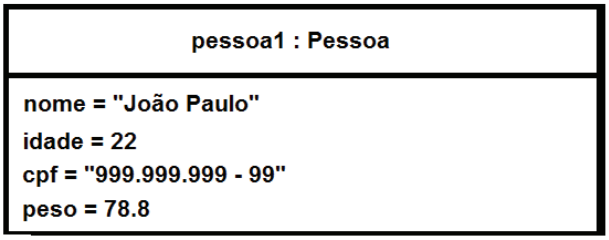

Figura 24 – Definição UML de um objeto do tipo Pessoa.

Ou simplesmente mantendo o nome e o tipo do objeto:

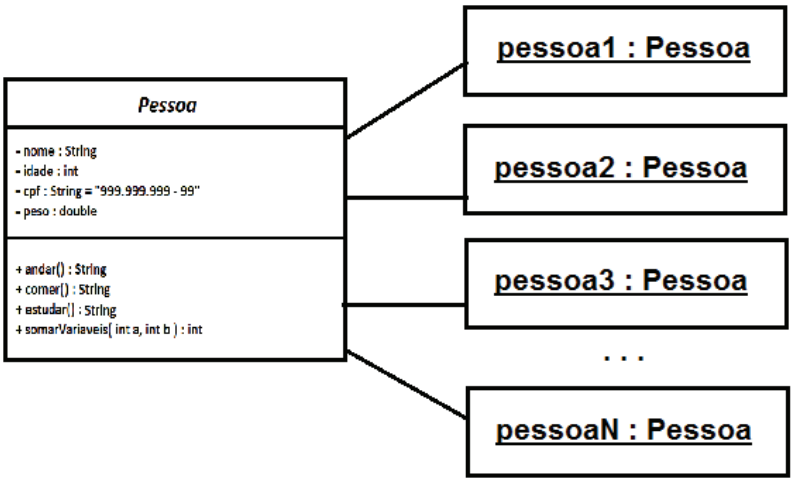

Figura 25 – Diagrama de objetos da classe Pessoa.

Todo objeto possui um ciclo de vida que engloba o momento em que é declarado até sua eliminação. No instante em que um objeto é declarado, é alocado um espaço na memória para ele e automaticamente executado seu construtor. Todo objeto possui um **construtor**, que é responsável pela criação do objeto. A partir deste momento o objeto está pronto para ser usado.

Esta fase vai até a eliminação do objeto. A sua eliminação pode ser de duas formas: a primeira, que todas as linguagens utilizam, elimina o objeto no final do programa se ele for global, no final de um método se for local, e no final de um bloco se for declarado dentro deste. A segunda forma de eliminação é chamada de Garbage Collection. Esta maneira de eliminação não é implementada em todas as linguagens, não é uma característica somente de orientação a objetos.

Garbage Collection consiste na eliminação do objeto pelo compilador, depois de sua última utilização. A partir do momento em que ele não é mais referenciado, passa a não existir mais na memória. Por exemplo, Garbage Collection é implementado em Java e SmallTalk enquanto que em C++ e Pascal não.

Herança

O conceito de encapsulamento não é exclusivo da orientação a objetos. O que torna a orientação a objetos única é o conceito de herança.

A herança é um mecanismo que permite que características comuns a diversas classes sejam agrupadas em uma classe base, ou superclasse. A partir desta superclasse, outras classes podem ser especificadas. Cada classe especificada, ou subclasse, herda as propriedades e comportamentos de sua superclasse e acrescenta o que for definido de particularidade para ela.

Há várias formas de relacionamento em herança:

- **Extensão**: onde a subclasse estende a superclasse, acrescentando novos membros (atributos e métodos).

- **Especificação**: onde a superclasse especifica o que uma subclasse deve oferecer, mas não implementa nenhuma funcionalidade, deixando como responsabilidade para as subclasses.

Podemos também utilizar uma combinação entre estas duas formas, onde a subclasse herda as definições de métodos sem implementação, implementa-os e acrescenta novos comportamentos particulares a ela.

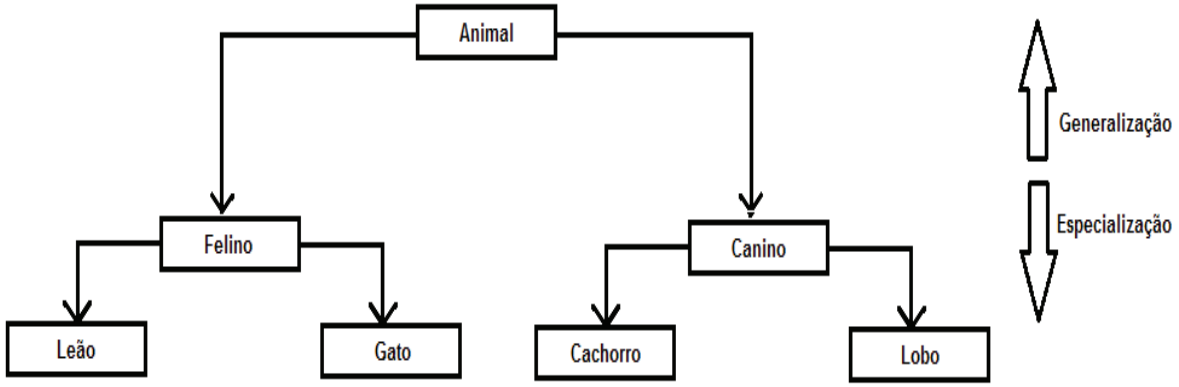

Figura 26 – Diagrama de Herança.

Podemos ver na figura acima, que quanto mais descemos na árvore da herança, mais específicos nós somos, e quanto mais subimos, mais gerais nós somos. Pela definição, Leão e Gato (subclasses) são Felinos (superclasse), portanto eles herdam comportamentos em comum de um Felino, mas acrescentam propriedades particulares de cada um, como porte físico por exemplo. Da mesma forma, Felino e Canino (subclasses) são Animais (superclasse), portanto tanto Canino quanto Felino tem comportamentos de animais, pois os herdaram, mas Felino tem suas particularidades e Canino também.

É assim que funciona o mecanismo da herança, as subclasses herdam propriedades e comportamentos da superclasse, e acrescenta o que é particular de cada uma. Segue um exemplo para deixar claro o funcionamento da herança:

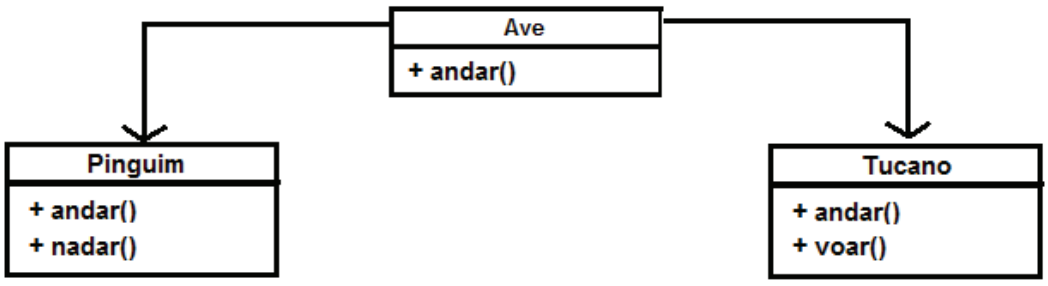

Figura 27 – Herança Ave.

Como podemos ver, Pinguim e Tucano (subclasses) são Aves (superclasse), portanto, os dois são capazes de andar, pois herdaram este comportamento de Ave. Porém, Pinguim tem como particularidade a capacidade de nadar, e Tucano tem como particularidade a capacidade de voar. Dessa forma concluímos nosso estudo sobre Herança.

Polimorfismo

O polimorfismo está intimamente ligado ao conceito de herança. Polimorfismo é o principio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que tem a mesma identificação (assinatura), mas implementações distintas, especializadas para cada classe derivada, usando para isso uma variável do tipo da superclasse.

Por exemplo, suponha que o algoritmo utilizado no método andar() da classe Pinguim retorne o texto "Pinguim andando..." e o algoritmo utilizado no método andar() de Tucano retorne "Tucano andando...". Pela definição de herança os dois objetos, Pinguim e Tucano, podem andar, mas Pinguim anda do seu jeito e Tucano do seu, por isso que os métodos possuem implementações diferentes.

Utilizando a linguagem Java como exemplo:

```
1  Ave a = new Pinguim();
2  a.andar();
3
4  a = new Tucano();
5  a.andar();
```

Figura 28 – Polimorfismo entre Pinguim e Tucano.

Na linha 1 criamos um objeto do tipo Pinguim e armazenamos em uma variável do tipo Ave. Isso é possível porque Pinguim é uma Ave, segundo nossa representação de Herança na Figura 27.

Na linha 2 invocamos o método andar() do objeto Pinguim (subclasse) através de uma variável Ave (superclasse), fazendo uso do Polimorfismo.

Da mesma forma, utilizamos o polimorfismo na linha 5, através do objeto Tucano.

Dessa forma, a saída destas instruções seria:

```
Pinguim andando...

Tucano andando...
```

Esse mecanismo é fundamental na programação orientada a objetos, permitindo definir funcionalidades que operem genericamente com objetos, abstraindo-se de seus detalhes particulares quando estes não forem necessários.

Para o polimorfismo ser utilizado, é necessário que o método definido na subclasse tenha exatamente a mesma assinatura do método definido na superclasse. Esta técnica de manter a assinatura, mas alterar a implementação do método, é conhecida como **sobreposição de método.**

Estes foram os principais conceitos do paradigma da orientação a objetos.