Unidade II

Na unidade anterior, apresentamos os principais fundamentos e princípios para avaliar as LPs, tendo como base as implementações de cada uma delas. Agora, estudaremos os conceitos, as propriedades, as técnicas e a própria implementação das LPs dos paradigmas estruturado, orientado a eventos e orientado a objetos.

5 PARADIGMAS DAS LPs

De acordo com Varejão (2004, p. 19), "dá-se o nome de paradigma a um conjunto de características que servem para categorizar um grupo de linguagens".

Conhecer os diferentes paradigmas das LPS permite ao desenvolvedor dominar várias técnicas para resolução de problemas por meio de uma visão mais ampla das aplicações computacionais disponíveis. É essencial que todos os profissionais que desejam trabalhar com desenvolvimento de software dominem as diversas LPs e conheçam os diversos paradigmas de programação (TUCKER; NOONAN, 2009).

Algumas LPs são multiparadigmas, suportando e permitindo o uso combinado ou não de diversos estilos de programação, o que facilita a criação de programas mais eficientes e flexíveis.

5.1 Programação estruturada

O paradigma de programação estruturada foi um modelo fundamental de programação que caracterizava as principais LPs desenvolvidas nas décadas de 1970 e 1980, e que até os dias atuais, ainda é muito utilizado para o desenvolvimento de software. Ele oferece uma metodologia na qual a visão que o programador possui sobre a estrutura e execução do programa se abstrai como uma sequência de funções executadas de modo empilhado.

Outros paradigmas, como os orientados a objetos, em que nos aprofundaremos posteriormente, utilizam características do paradigma estruturado, por exemplo, os módulos ou subprogramas da programação estruturada se converteram nos métodos utilizados na programação orientada a objetos.

A programação estruturada tem grande importância comercial e acadêmica, pois geralmente, nas instituições de Ensino Superior, os estudantes iniciam o seu aprendizado de programação por esse paradigma.

5.1.1 Caracterização

A programação estruturada é um modelo de programação de computadores que estabelece uma disciplina no desenvolvimento de algoritmos, independentemente da LP na qual a solução será codificada, ou da complexidade do problema computacional.

Esse paradigma guia os desenvolvedores na criação de estruturas simples em seus programas, usando um número reduzido de estruturas de codificação com especial destaque para a modularização ou subprogramação.

À medida que os problemas se tornam maiores e mais complexos, sempre é possível simplificar a solução dividindo o programa em partes menores, seguindo o lema dividir para conquistar, chamadas de subprogramas ou módulos baseados em procedimentos e funções.

No paradigma de programação estruturada, o fluxo de execução do programa é evidente, não são admitidos saltos condicionais (comando Goto) e todos os programas podem ser construídos utilizando apenas três estruturas essenciais: estrutura sequencial, estrutura condicional e estrutura repetição.

- **Estrutura sequencial**: usada para a entrada e saída de dados e informações.

- **Estrutura condicional**: utilizada para a tomada de decisão ao longo do fluxo de execução do programa, conforme exemplo em C# a seguir:

```
int mediaAluno;
if (mediaAluno >= 7)
    Console.WriteLine("Aluno aprovado com média maior ou igual 7")
else
    Console.WriteLine("Aluno reprovado na disciplina")
```

- **Estrutura repetição**: empregada em trechos do código que necessitam de execuções constantes, conforme exemplo em C# a seguir:

```
while (mediaAluno >=7)
     Console.WriteLine("Média do aluno maior ou igual a 7")
}
```

Lembrete

Esse paradigma possui uma estrutura organizada em módulos ou subprogramas, que são denominados procedimentos ou funções, conforme foi visto na seção sobre subprogramas.

A diferença entre função e procedimento está no fato de que uma função sempre deve retornar um resultado para a unidade chamadora. Por exemplo, uma função pode fazer um cálculo e retornar o seu resultado.

Já um procedimento é um subprograma que executa determinada tarefa e não retorna nenhuma informação para a unidade chamadora.

O estado de um programa é o conceito fundamental da programação estruturada. Ele representa a situação das variáveis em determinado instante de tempo no fluxo de execução do programa. Esse estado pode ser alterado por meio da manipulação das variáveis e dos chamados funções e procedimentos.

Um programa para ser considerado estruturado deve possuir o fluxo de execução claro, facilitando o acompanhamento dos detalhes de sua execução ao longo das diversas linhas de código.

No paradigma estruturado não devemos utilizar saltos incondicionais, como na LP Basic o comando Goto, que permite desviar o fluxo de execução para qualquer parte do código. Algumas LPs possibilitam ao desenvolvedor mudar inesperadamente o fluxo de execução do programa por meio do comando Goto.

Para exemplificar o uso do comando Goto (Ir para), o trecho de código demonstrado a seguir apresenta a utilização de saltos incondicionais na LP Visual Basic (VB), na qual, de acordo com o valor do número informado na variável numeroEntrada, o programa continua sua execução pelo desvio de fluxo 1 ou desvio de fluxo 2. O uso dos saltos pode ser visualizado nas linhas 5, 6 e 9, quando o fluxo de execução da aplicação é transferido para o rótulo predeterminado pelo programador em uma linha específica.

```
1. Sub ExemploUsoComandoGoTo()

2. Dim numeroEntrada As Integer = 1

3. Dim retornaTexto As String

4. ' Avalia o número informado na variável numeroEntrada e se este número for 1 segue pelo fluxo
   1 senão pelo fluxo 2.

5. If numeroEntrada = 1 Then GoTo Fluxo1 Else GoTo Fluxo2

6. Fluxo1:

7. retornaTexto = "O número informado é igual a 1"

8. GoTo SegueFluxoPrincipal

9. Fluxo2:

10. retornaTexto = " O número informado é diferente de 1".

11. GoTo SegueFluxoPrincipal

12. SegueFluxoPrincipal:
```

```
13. ' Imprime o valor da variável retornaTexto na janela de Depuração.

14. Debug.WriteLine(retornaTexto)

15. End Sub
```

Inúmeras discussões podem ser encontradas na literatura a respeito da utilização de saltos incondicionais, sendo que a maior parte delas não considera sua utilização como uma boa prática de programação.

5.2 Linguagens imperativas (Basic, Pascal, C)

Conforme já vimos, o paradigma imperativo fundamenta-se na ideia da computação como um processo que é realizado por uma sequência de alterações no estado da memória do computador. O programa e as variáveis são armazenados juntos e contêm uma série de instruções que permitem obter os valores de entrada, processar cálculos, atribuir valores a variáveis, além de realizar desvios de fluxo para outros pontos do programa por meio de uma série de comandos, de acordo com a LP empregada. Por ser de fácil entendimento, é amplamente utilizado em cursos introdutórios de programação.

No mercado existem muitas outras linguagens que dão suporte ao paradigma imperativo, como Basic, Pascal e C.

Na sequência temos a sintaxe e um exemplo simples desenvolvido na linguagem Basic. Cada LP possui sua própria sintaxe, por exemplo, no Basic a declaração de uma variável é realizada com o comando Dim, da seguinte forma:

```
Sintaxe: Dim <nome_da_variavel> As tipo_variavel
Dim nome_cliente As String
```

No Basic, se a declaração da variável for feita nas primeiras linhas do programa fora de uma função ou procedimento, a variável poderá ser utilizada por todos os procedimentos, funções e objetos do programa.

Quando a declaração de uma variável é realizada em um objeto, função ou procedimento, a variável é válida apenas no escopo em que ela foi declarada.

O exemplo a seguir demonstra a implementação de um programa na linguagem Visual Basic. Observe que a variável mensagem está declarada no topo do código, o que significa que os dados armazenados nela podem ser acessados de qualquer parte do código. Então, o evento click do botão "CliqueMe" acessa o conteúdo da variável e exibe a mensagem para o usuário.

```
Dim mensagem As String

Private Sub Form_Load()
    mensagem = "Alo Mundo, Basic!"
End Sub

Private Sub CliqueMe_Click()
    MsgBox mensagem
End Sub
```

6 PRÁTICAS DE PROGRAMAÇÃO UTILIZANDO PROGRAMAÇÃO ESTRUTURADA: EXEMPLOS DE PROGRAMAS E IMPLEMENTAÇÕES PRÁTICAS (BASIC, PASCAL, C)

Demonstraremos a implementação de uma calculadora com quatro operações na linguagem C, utilizando a programação estruturada.

Observe os seguintes recursos e estruturas que são característicos da programação estruturada:

- Nesse programa, utilizam-se duas variáveis do tipo float: valor1 e valor2. Escolhemos o tipo citado por aceitar o ponto decimal, assim o usuário não precisa somente digitar números inteiros. Uma variável do tipo int denominada menuOpcoes, que aceita números inteiros, para o usuário digitar um número de 1 a 4 de acordo com a operação que deseja realizar no aplicativo: Somar, Subtrair, Multiplicar ou Dividir.

- A estrutura de repetição usa o laço Do while para controlar a exibição do Menu.

- A estrutura de decisão do comado switch testa sucessivamente o valor da lista de 0 a 4 e quando o valor coincide, os comandos associados àquela constante são executados. No nosso exemplo são a execução da operação escolhida e a impressão na tela do seu resultado.

- A utilização das funções printf e scanf para interagir com a entrada de dados do usuário. A função printf exibe um ou mais dados na tela, instruindo o usuário sobre os próximos passos. A função scanf captura o valor digitado pelo usuário no teclado.

- Para limpar a tela, é utilizado o comando system("cls || clear"), que limpa a tela de saída de programa para tudo que já foi escrito que, no nosso exemplo, é executado no MS-DOS.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
int menuOpcoes;
float valor1, valor2;
```

```
do
{
printf("\t\t\n Selecione o número correspondente a operação desejada\n");
printf("0. Sair Menu Opções\n");
printf("1. Somar\n");
printf("2. Subtrair\n");
printf("3. Multiplicar\n");
printf("4. Dividir\n");

printf("Operação: ");
scanf("%d", & menuOpcoes);

printf("Digite o primeiro número: ");
scanf("%f", &valor1);

printf("Digite o segundo número: ");
scanf("%f", &valor2);

switch(menuOpcoes)
{
case 0:
system("cls || clear");
printf("Saindo do menu de opções...\n");
break;
case 1:
system("cls || clear");
printf("%f + %f = %f \n", valor1, valor2, valor1 + valor2);
break;
case 2:
system("cls || clear");
printf("%f - %f = %f \n", valor1, valor2, valor1 - valor2);
break;
case 3:
system("cls || clear");
printf("%f * %f = %f", valor1, valor2, valor1 * valor2);
break;
case 4:
system("cls || clear");
printf("%f / %f = %f", valor1, valor2, valor1 / valor2);
break;
default:
system("cls || clear");
printf("Menu inválido! Digite um número de 0 a 4.\n");
}

} while(menuOpcoes);
```

Posteriormente, temos outro exemplo prático da programação estruturada em que implementamos uma simples calculadora com quatro operações, porém na linguagem Pascal. Nosso objetivo é demonstrar as principais características desse paradigma em duas LPs diferentes.

Observe os seguintes recursos e estruturas característicos da programação estruturada:

- O comando var marca o início da declaração de variáveis. A utilização de três variáveis nesse programa: valor1 e valor2 do tipo de dados real, escolhemos esse tipo por aceitar o ponto decimal, assim o usuário não precisa somente digitar números inteiros. A variável operacao do tipo char aceita textos ou string, dessa maneira atribuímos o símbolo da operação que o usuário quer realizar no aplicativo: (+) Somar, (-) Subtrair, (*) Multiplicar ou (/) Dividir.

- A estrutura begin/end marca o início e o fim de um bloco de instruções.

- A estrutura de decisão, o comando if, testa sucessivamente o símbolo digitado pelo usuário e quando o valor coincide, os comandos associados àquela constante são executados.

- A utilização das funções readln e writeln para interagir com a entrada de dados do usuário. A função writeln exibe um ou mais dados na tela, instruindo o usuário sobre os próximos passos. A função readln captura o valor digitado pelo usuário no teclado.

- O comando "clrscr" é utilizado para limpar a tela de saída do programa para tudo o que já foi escrito que, no nosso exemplo, é executado no MS-DOS.

A seguir um exemplo de calculadora simples em Pascal:

```
program calculadoraSimples;
uses crt;
var
valor1,valor2,resultado:real;
operacao:char;
begin
   clrscr;
   writeln('Entre com o primeiro valor:');
   readln(valor1);
   writeln('Entre com o segundo valor:');
   readln(valor2);
   writeln('Selecione uma operacao(+,-,*,/));
   readln(operacao);
   {operacao somar}
   if(operacao='+')then
       resultado:=valor1+valor2;
       {operacao subtrair}
   else if(operacao='-')then
```

```
        resultado=valor1-valor2;
        {operacao multiplicar}
    else if(operacao='*')then
        resultado=valor1*valor2;
        {operacao dividir}
    else if(operacao='/')then
        resultado=valor1/valor2;
        writeln(resultado:8:3);
end.
```

7 PROGRAMAÇÃO ORIENTADA A EVENTOS

O paradigma de programação orientada a eventos tem adquirido grande importância devido ao crescimento exponencial da utilização de aplicações mobile e Web, nas quais o usuário, para interagir com essas interfaces, faz uso de toques em botões, digita textos e seleciona diversas opções nos menus da aplicação.

No paradigma imperativo tradicional, o desenvolvedor de software é quem determina a sequência de comandos que alteram o estado atual do sistema, desde a entrada dos dados até atingir o estado final.

O paradigma de programação orientada a eventos é um modelo de desenvolvimento de software cujas partes do programa são executadas em momentos inesperados, não se controla a sequência na qual ocorrem os eventos de entrada de dados. Normalmente, eles são disparados por interações do usuário, por meio de componentes gráficos ou objetos, quando o software está em execução (SEBESTA, 2011).

Nele, as ações do programa são estruturadas através de eventos, que podem ser programados pelo desenvolvedor para atender aos requisitos de software, criando assim procedimentos específicos.

7.1 Conceitos fundamentais

Eventos em um software são as operações que o usuário realiza no programa. Os programas orientados a eventos não seguem um fluxo de controle padronizado, em vez disso, eles são codificados para reagir a qualquer sequência de eventos (TUCKER; NOONAN, 2009).

No processo de desenvolvimento de software, um evento ocorre em um momento inesperado e produz uma operação em uma unidade computacional preparada para tratar esses eventos (ETZION; NIBLETT, 2011).

Mas o que são eventos? Em uma fábrica, por exemplo, quando toca ou dispara a sirene na hora almoço, os funcionários interrompem as suas atividades e vão almoçar. Então, podemos dizer que a sirene é um evento que informa aos funcionários que chegou a hora do almoço e que eles podem se dirigir ao refeitório.

As aplicações desenvolvidas para a internet também têm controles acionados por eventos, por exemplo, um sistema de registro de estudante on-line deve ser preparado para interagir com o usuário, independentemente de qual seja a sua próxima ação: cadastrar um curso, alterar dados de um curso, escolher em qual classe vai assistir as suas aulas etc. Outro exemplo é um sistema de reservas web de uma companhia aérea, que também deve estar preparado para responder a várias sequências de eventos do usuário, como selecionar a data da viagem, o destino preferido ou o local da poltrona no avião.

Ainda, a programação orientada a eventos é utilizada em dispositivos mobile, sistemas de navegação em aviões e sistemas de segurança residencial. Nos aviões, os eventos podem reagir a respostas programadas em razão da mudança na direção e velocidade do vento ou temperatura, pois esses eventos não ocorrem em determinada ordem de maneira previsível (TUCKER; NOONAN, 2009).

São alguns exemplos de eventos:

- Toda vez que o usuário movimenta o mouse.
- Aperta um botão do mouse.
- Aciona uma tecla no teclado do computador.
- Digita em uma caixa de texto.
- Seleciona um item no menu.

Um evento é a notificação de que alguma coisa aconteceu. O desenvolvedor de software pode, por exemplo, utilizar esses eventos para elaborar um programa cujo processamento ocorra a partir da sua execução.

O exemplo mais comum de programa acionado por eventos, utilizado atualmente, é o daqueles que utilizam os componentes de interface gráfica de usuário (graphical user interface – GUI), encontrada na maioria dos computadores pessoais e laptops (TUCKER; NOONAN, 2009). Essas GUI usam elementos gráficos como: caixas de texto, botões, caixas de seleção, rótulos, entre outros.

Os componentes visuais de uma GUI são objetos que podem ser notificados por eventos desde que exista a interface apropriada para tratar o evento. Um exemplo bem simples é apertar um botão em um formulário web para exibir uma mensagem de boas-vindas para o usuário. Nesse contexto, no qual o tratamento de eventos está sendo apresentado, um evento é um objeto criado pelo sistema em tempo de execução em resposta a uma ação de usuário (SEBESTA, 2011).

Cada um desses elementos gráficos pode ser programado para reagir a determinados eventos de acordo com o objetivo e regras do software que está sendo projetado.

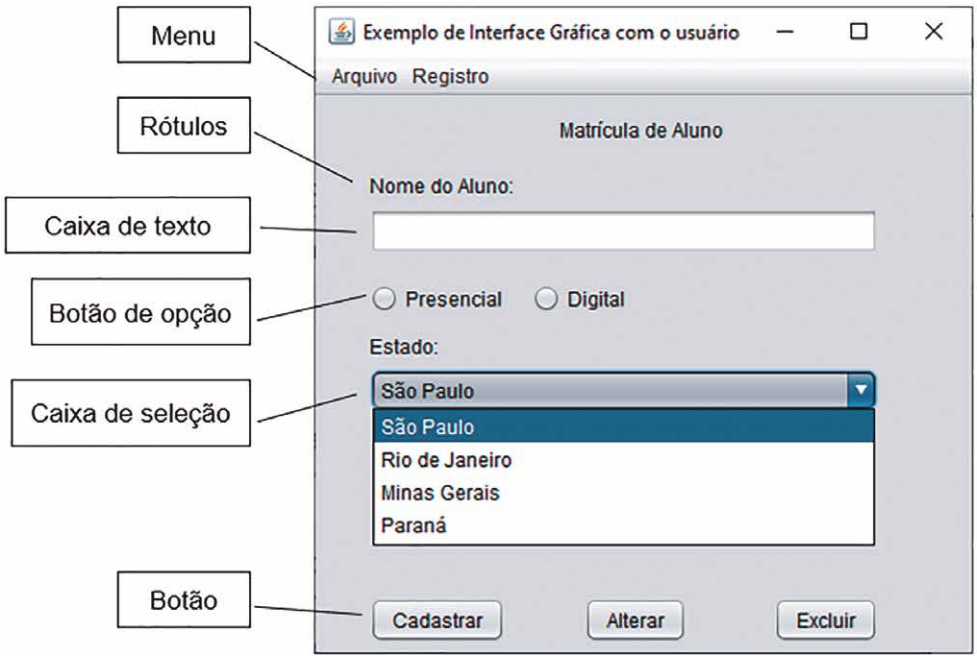


Figura 27 – Interface gráfica com o usuário (GUI)

Os componentes visuais são disparadores de eventos ou fontes de eventos. Na linguagem Java, os componentes que acionam eventos pertencem à classe JComponent. São alguns dos exemplos mais comuns: JTextField (Caixa de texto), JButton (botões que podem ser clicados pelo usuário) e JComboBox (Caixas de seleção). Cada um deles gera objetos associados a eventos conforme a interação com o usuário. No nosso exemplo, quando o usuário clicar no botão Cadastrar, este poderá gerar um objeto de evento da classe ActionEvent.

Lembrete

As aplicações gráficas permitem a criação de uma interface gráfica de usuário (graphical user interface – GUI) na qual o desenvolvedor pode definir componentes gráficos como caixa de textos, botões, barras de rolagem e menus, que facilitam a interação.

Os chamados Listeners, em tradução literal, ouvinte, são uma estrutura de programação utilizada para reconhecer, ouvir ou ficar à espera das ocorrências de eventos particulares que acontecem a determinado objeto. Para que um programa possa executar determinada ação ou algoritmo, o *Listener* serve para detectar o evento que foi disparado. A interface ActionListener é um ouvinte notificado quando o usuário clica em um botão na GUI, disparando um evento que possibilita executar um algoritmo, regra de negócio ou alguma decisão lógica.

Por exemplo, na linguagem Java, para detectar que um botão foi clicado pelo usuário, o programador adiciona neste botão o método addActionListener, a fim de que tal evento seja ouvido ou detectado pelo programa. O resultado é a exibição da mensagem conforme será mostrado na sequência.

A seguir um trecho de código com exemplo de Listener:

```
botaoDeAcao.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
            /*Quando o botão recebe a ação, neste local
            dispara o código que executa o algoritmo.
            */
            JOptionPane.showMessageDialog(null, "Item cadastrado com sucesso!");
      }
});
```

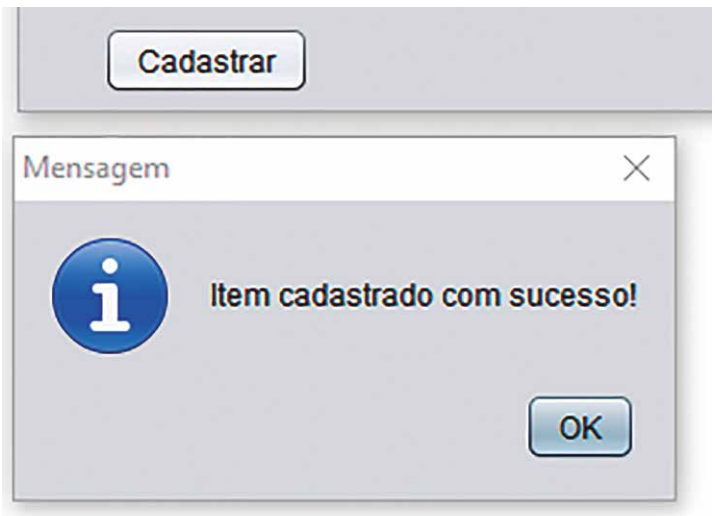


Figura 28 – Resultado da execução do código

Para responder aos eventos dos objetos, o desenvolvedor precisa implementar os métodos manipuladores chamados de handlers de evento. Esses métodos especiais são utilizados como tarefas para responder aos eventos detectados pelos listeners. Para cada classe de evento, é definido um nome de um manipulador que deve ser implementado por uma interface listener (TUCKER; NOONAN, 2009). Um tratador de evento é um trecho de código executado em resposta ao disparo de um evento. Ele habilita um programa a responder às ações do usuário.

No desenvolvimento de um programa orientado a eventos, são utilizados padrões da programação estruturada, por exemplo, os subprogramas ou procedimentos executados a partir do disparo ou chamada de um evento. Logo, para cada evento disparado pelo programa, o programador cria um conjunto de instruções, funções ou procedimentos que serão realizados para resolução de algum problema de software.

Observação

Listener é uma palavra da língua inglesa que significa "ouvinte" em português. Na programação é como se o sistema ficasse ouvindo, monitorando ou esperando alguma ação do usuário.

Na linguagem Java, para tratar os eventos em um determinado objeto, realizamos os seguintes procedimentos:

- Criamos o objeto na GUI, por exemplo, um botão Cadastrar. Esse componente, chamado na linguagem Java como JButton, possui um método denominado addActionListener(), que recebe como argumento do método entre parênteses a interface ActionListener. Portanto, quando o botão Cadastrar for clicado, esse Listener será executado.

- Adicionamos, no código-fonte, o ActionListener ao objeto que tratará este evento, utilizando o comando "new", instanciando um objeto ActionListener para implementar o método de tratamento.

- Implementamos o método actionPerformed, que durante o disparo de um evento realiza um determinado procedimento, que no nosso exemplo é executar o método de tratamento "cadastrarUsuario()" onde implementamos o nosso algoritmo ou lógica.

Desta forma, quando o usuário clicar no botão "Cadastrar", o programa executará a rotina fictícia "Cadastrar usuário". Os dados informados na GUI, como nome e estado da federação, serão, por exemplo, armazenados em um banco de dados.

```
jButtonCadastrar.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
           cadastrarUsuario();
        }
    });
```

Cada componente visual tem um conjunto de eventos disponíveis que pode ser utilizado pelo programador para estabelecer o fluxo do programa, de acordo com os seus objetivos de desenvolvimento. Especificamos a seguir alguns dos mais utilizados:

- **Eventos da Janela** (**WindowListener**): são disparados quando uma janela pode ser aberta (Open), fechada (Close), maximizada (maximize), minimizada (minimize) ou alterada de tamanho (Resize).

- **Eventos de Teclado** (**KeyListener**): são disparados quando uma tecla é pressionada, ou é solta: (Down, Press, Up).

- **Eventos do Mouse** (**MouseListener**, **MouseMotionListener**): são disparados por cliques do mouse ou por movimentos de mouse, por exemplo, o mouse entrando ou saindo de determinada área de um componente: Click, MouseHover, MouseLeave, MouseMove.

O quadro a seguir exibe uma lista de eventos, listeners e métodos disponíveis na linguagem Java.

Quadro 11 – Alguns exemplos de eventos na linguagem Java

Event	Interface listener	Método
WindowEvent	WindowListener	windowClosing(WindowEvent e) windowClosed(WindowEvent e) windowOpened(WindowEvent e) windowIconified(Window Event e) windowDeIconified(Window Event e) windowActivatebd(Window Event e) windowDeactivated(Window Event e)
keyEvent	KeyListener	keyPressed(KeyEvent event); keyReleased(KeyEvent event); keyTyped(KeyEvent event);

MouseEvent	MouseListener	mouseClicked(MouseEvent event);
		mouseEntered(MouseEvent event);
		mouseExited(MouseEvent event);
		mousePressed(MouseEvent event);
		mouseReleased(MouseEvent event);

Exemplo de aplicação

Além da interface ActionListener, existem outras que tratam objetos diferentes, por exemplo, a interface KeyListener, que fica ouvindo se alguma tecla do teclado foi pressionada. Faça uma pesquisa na internet sobre outros Listeners disponíveis na linguagem Java e qual a função de cada um deles.

7.2 Caracterização

Propriedades e eventos são características dos componentes visuais. Existem diferenças entre ambos. Não devemos confundi-los. Geralmente, em uma IDE (Visual Studio ou Eclipse), quando selecionamos um objeto as propriedades são disponibilizadas em uma guia e os eventos em outra.

Alguns dos exemplos de propriedades mais comuns nos componentes visuais são:

- Nome do componente (Name): um nome que identifica o componente no programa.
- Fonte (Font).
- Cor (Color, Background color).

- Altura (Height): com a largura define a dimensão do componente.

- Largura (Width): com a altura define a dimensão do componente.

É importante sabermos diferenciar nos componentes visuais os eventos que podem ser disparados para determinar o fluxo do programa de suas propriedades do objeto, por exemplo, cor, nome e tamanho.

Veja na sequência algumas das vantagens do paradigma orientado a eventos:

- Trata-se do paradigma adequado para trabalhar com interfaces gráficas como web e mobile. Permite que o usuário utilize a interface gráfica para disparar uma atividade sem que o sistema saiba previamente o momento da sua execução. Um exemplo é o envio de e-mails na interface do programa MS-Outlook, como mostrado a seguir:

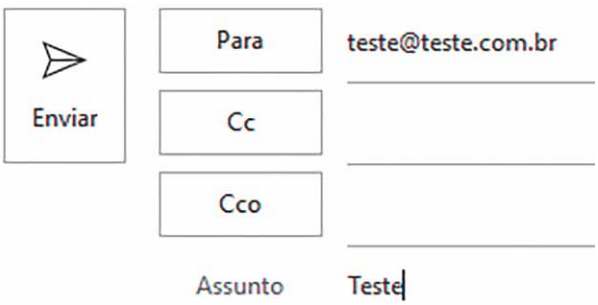


Figura 29 – Exemplo de interface gráfica para enviar e-mail

- É amplamente utilizado em interfaces gráficas, programação mobile e jogos. O programa pode executar ações utilizando pouco código-fonte, através dos eventos, por exemplo, em formulários, caixas de texto e botões. Também é possível usar diversos componentes e controles, inclusive de terceiros e aplicá-los nos formulários.

- Exige menos esforço para o desenvolvedor com relação à programação visual, pois as aplicações são criadas a partir de janelas gráficas com a utilização de ferramentas fornecidas pelas IDEs para o design de telas.

- Acontece através dos componentes da tela que disponibilizam eventos que fazem a interação com o usuário. Por meio dos tratadores de eventos, o desenvolvedor orquestra a execução do programa.

- Permite que o programador projete visualmente a aplicação de acordo com requisitos do produto de software, tornando a programação mais flexível.

- Possibilita a programação dos componentes por meio de um enorme leque de eventos que atendem aos diversos requisitos do que se deseja que o programa faça e interage com o usuário.

7.3 Linguagens orientadas a eventos (Delphi, Visual Basic)

Algumas LPs que dão suporte a paradigmas orientados a eventos são: Delphi e Visual Basic.

É uma linguagem que fornece suporte tanto para o paradigma orientado a eventos quanto ao paradigma orientado a objetos. Os dados do programa são utilizados pelos objetos e são operacionalizados por ferramentas que fazem parte do próprio programa.

Como exemplo de aplicação Delphi, temos um procedimento que faz a soma de dois números e exibe o resultado em um campo chamado resultado. O código foi inserido no evento click do botão somar. Depois, o usuário informa os dois valores e realiza um clique no botão somar. O evento clique do botão dispara o procedimento que captura o valor digitado no campo Primeiro Número, soma com o valor digitado no campo Segundo Número e exibe o resultado da soma no campo Resultado.

No exemplo foi feita a conversão de texto (String) para número inteiro utilizando a função StrToInt. Ela deve ser realizada, pois, para utilizar a função soma, os valores das variáveis precisam ser do tipo numérico, e não texto. A conversão também precisa ser realizada em outras LPs devido ao fato de os dados provenientes de objetos Windows virem em formato texto. Depois que os dois valores são somados, é realizada a conversão novamente para texto com a função IntToStr, pois o resultado será exibido em um objeto MS-Windows:

```
Procedure TForm1.SomarClick(Sender: TObject);
begin
      resultado.Text:=
      IntToStr(StrToInt(ValorDoPrimeiroNumero.Text)+
      IntToStr(StrToInt(ValorDoSegundoNumero.Text);
end;
```

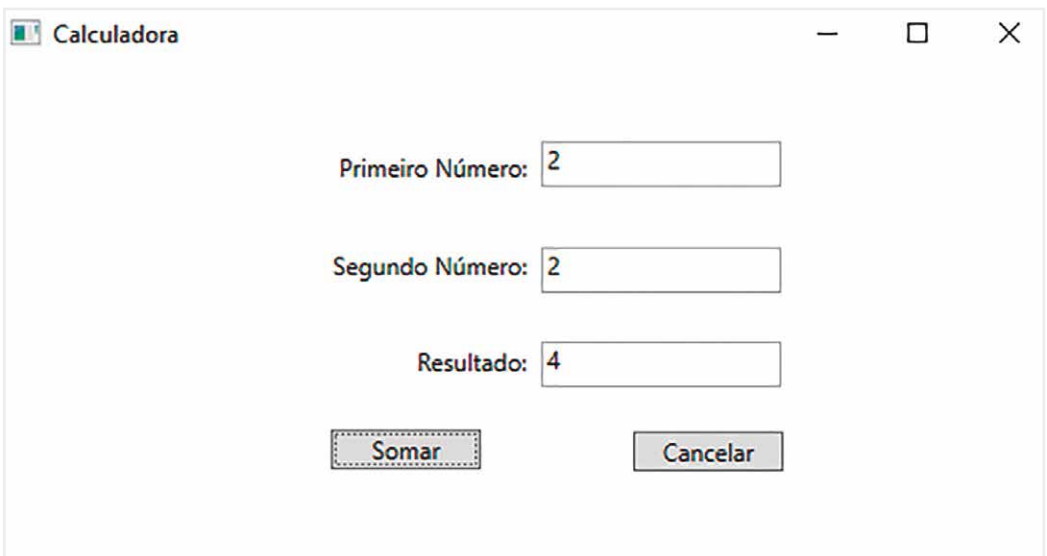


Figura 30 – Exemplo de calculadora em Delphi

Saiba mais

A linguagem Delphi pode ser utilizada para o desenvolvimento de aplicativos do tipo desktop, cliente/servidor e web. Inicialmente, foi criada pela empresa Borland, mas atualmente é mantida pela empresa Embarcadero. A fim de conhecê-la melhor, acesse o site a seguir:

EMBARCADERO. *Delphi 11*. [s.d.]. Disponível em: https://bit.ly/3ud3QeO. Acesso em: 21 jan. 2022.

A linguagem Visual Basic oferece suporte ao paradigma orientado a eventos, possui uma IDE própria totalmente gráfica, facilitando muito a construção de GUIs. Pertence à empresa Microsoft e pode ser utilizada para desenvolvimento de aplicativos do tipo desktop e cliente/servidor.

Como exemplo de aplicação Visual Basic, temos o formulário "Cadastro de Vendedor". Nele, ao clicar no botão Cadastrar, o evento produz uma caixa de mensagem informando o status "Vendedor cadastrado com sucesso", conforme a figura a seguir. Esse código foi inserido no evento click do botão Cadastrar:

```
Private Sub Cadastrar_Click()
    MsgBox ("Vendedor cadastrado com sucesso")
End Sub
```

Figura 31 – Exemplo do cadastro de vendedor em Visual Basic

7.4 Práticas de programação utilizando programação orientada a eventos (Delphi, Visual Basic)

Seguem alguns exemplos para evidenciar as principais características de programas e implementações práticas utilizando o paradigma orientado a eventos. O primeiro deles mostra o funcionamento de uma aplicação simples, apenas para fins didáticos, uma calculadora com quatro operações, desenvolvida na LP Delphi, conforme a figura a seguir.

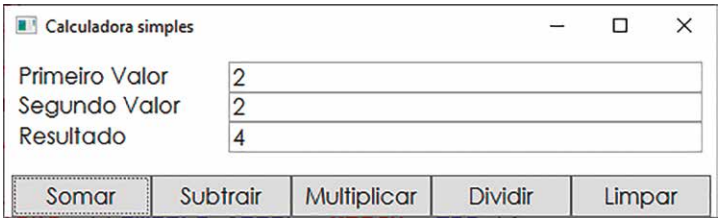


Figura 32 – Calculadora simples na linguagem Delphi

Para criar uma calculadora básica e simples em Delphi, utilizaremos componentes nativos e código simples de fácil compreensão, somente para favorecer o entendimento do uso de eventos.

Em um formulário, criaremos a parte visual da calculadora e alteraremos a propriedade name.

- **Inserir 5 botões no form, e alterar a propriedade Caption de cada botão para sua respectiva operação**: Somar, Subtrair, Multiplicar, Dividir e Limpar.
- **Inserir 3 labels**: Primeiro Valor, Segundo Valor e Resultado.

No evento OnClick de cada botão, criaremos um procedimento que verificará se os campos primeiroNumero e segundoNumero estão preenchidos. Em caso afirmativo, então será realizada a seguinte operação:

```
Procedure Calculadora.SomarClick(Sender: TObject);
Begin
        if (primeiroNumero.Text <> '') and (segundoNumero.Text <> '') then
   resultado.Text:=IntToStr(StrToInt(primeiroNumero.Text)+IntToStr(StrToInt(segundoNumero.Text);
end;

Procedure Calculadora.SubtrairClick(Sender: TObject);
Begin
        if (primeiroNumero.Text <> '') and (segundoNumero.Text <> '') then
      resultado.Text := IntToStr(StrToInt(primeiroNumero.Text) -IntToStr(StrToInt(segundo
Numero.Text);
end;

Procedure Calculadora.MultiplicarClick(Sender: TObject);
begin
        if (primeiroNumero.Text <> '') and (segundoNumero.Text <> '') then
   resultado.Text:=IntToStr(StrToInt(primeiroNumero.Text)*IntToStr(StrToInt(segundoNumero.Text);
end;

Procedure Calculadora.DividirClick(Sender: TObject);
Begin
        if (primeiroNumero.Text <> '') and (segundoNumero.Text <> '') then
   resultado.Text:=IntToStr(StrToInt(primeiroNumero.Text)/IntToStr(StrToInt(segundoNumero.Text);
end;
```

```
Procedure Calculadora.LimparClick(Sender: TObject);
begin
    primeiroNumero.Text := '';
    segundoNumero.Text := '';
    resultado.Text := '';
end;
```

Após implementar os códigos mostrados, nossa calculadora está terminada. Execute o programa e veja-a em funcionamento.

O exemplo anterior ilustra como o paradigma da orientação a eventos aliado à utilização de IDEs facilita a construção de interfaces gráficas (GUIs).

Implementaremos uma aplicação simples, seguindo os mesmos moldes do exemplo anterior, porém agora na LP Visual Basic. Desenvolveremos um aplicativo que calcula o índice de massa corpórea (IMC). O usuário informa seu peso e altura e o programa aplica a fórmula do IMC. O IMC é um cálculo que avalia se a pessoa está com seu peso ideal em relação à altura. Conforme o resultado do cálculo do IMC, o indivíduo pode saber se está com o peso ideal, acima ou abaixo dele.

Nesse exemplo de programa, além do evento iniciado pelo clique no botão, programaremos uma opção que permite ao usuário acessar o procedimento por meio de eventos de teclado. Observe, na figura a seguir, que, no botão com o texto Calcular IMC, a letra C está sublinhada. Isso significa que se o usuário acionar respectivamente as teclas alt+c do teclado, ele executará o botão. Perceba que no botão Sair a letra S está sublinhada, significando que ela também pode ser acionada pelo teclado (pela combinação de teclas alt+s).

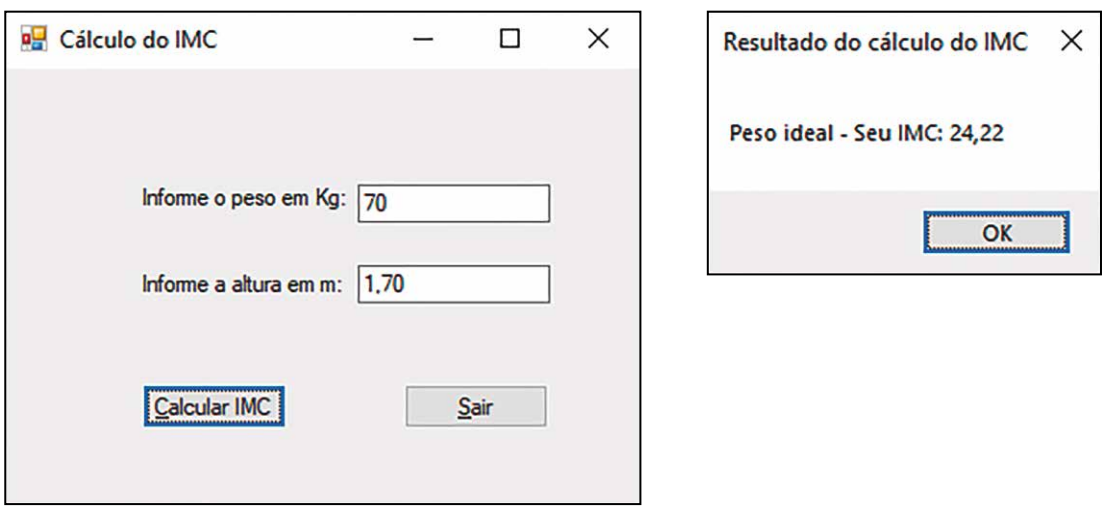


Figura 33 – Exemplo de programa em Visual Basic

Veja a seguir o código com os devidos comentários. Em uma LP, comentários são trechos do código-fonte ignorados pelo compilador ou interpretador, servindo para auxiliar na leitura e no

entendimento do código-fonte. De acordo com a sintaxe da linguagem Visual Basic, eles correspondem aos trechos do código-fonte localizados após as aspas simples (') até o final da linha. Trata-se de um recurso que o desenvolvedor pode utilizar para documentar o código-fonte durante a programação.

```
'Classe do formulário principal
Public Class Form1
'Evento clique do botão
   Private Sub BtnCalcularIMC_Click(sender As Object, e As EventArgs) Handles BtnCalcularIMC.Click
'Declaração de variáveis
      Dim peso, altura, imc As Double
      Dim mensagem As String
'Converte os valores inseridos pelo usuário em número com ponto flutuante (double - Dbl)
      peso = CDbl(txtPeso.Text)
      altura = CDbl(txtAltura.Text)
'Realiza o cálculo do IMC: peso dividido pela altura ao quadrado.
      imc = peso / (altura * altura)

'De acordo com o resultado, informa qual é estado em relação ao peso ideal
      If imc < 16 Then
         mensagem = "Magreza severa"
      ElseIf imc < 17 Then
         mensagem = "Magreza moderada"
      ElseIf imc < 18.5 Then
         mensagem = "Magreza leve"
      ElseIf imc < 25 Then
         mensagem = "Peso ideal"
      ElseIf imc < 30 Then
         mensagem = "Sobrepeso"
      ElseIf imc < 35 Then
         mensagem = "Obsidade nível 1"
      ElseIf imc < 40 Then
         mensagem = "Obsidade nível 2"
      Else
         mensagem = "Obsidade nível 3"
      End If
'Exibe o resultado para o usuário
      MsgBox(mensagem & " - Seu IMC: " & FormatNumber(imc, 2),, "Resultado do cálculo do IMC")
   End Sub
'Evento click do botão Sair
   Private Sub BtnSair_Click(sender As Object, e As EventArgs) Handles btnSair.Click
      End
   End Sub
End Class
```

Após implementar os códigos mencionados, basta executar o programa para vê-lo funcionar. Observe que não utilizamos nenhum código para programar o evento de teclado, isso foi feito na própria IDE do Visual Basic. Para tanto, na propriedade Text do botão, colocamos os nomes: &Calcular IMC e &Sair. O símbolo & comercial, indica para o programa que o usuário pode acessar o botão por meio do evento de teclado.

8 PROGRAMAÇÃO ORIENTADA A OBJETOS

O paradigma de programação orientada a objetos (POO) é um dos paradigmas mais requisitados pelas empresas de software na atualidade. Desde o final da década de 1980, a POO começou a ser considerada uma abordagem de desenvolvimento de software poderosa e prática.

Por meio de uma metodologia clara e objetiva, ela fornece todo o suporte para que o desenvolvedor possa entregar um programa com custo, prazo e qualidade adequados aos processos de desenvolvimento de software.

O POO está incluído em uma classificação maior, a das LPs imperativas que se baseiam em um modelo entrada-processamento-saída. Objetos possuem um estado associado aos seus atributos. Atributos funcionam como uma variável, podendo sofrer alterações ao longo da vida do objeto associado. Isso é similar ao funcionamento de um computador com a arquitetura de von Neumann, no qual uma sequência de instruções atua na memória, mudando o seu estado. Assim, objetos podem sofrer alterações de estado, caracterizando o paradigma imperativo. Por outro lado, no paradigma puramente funcional, essa mudança de estado não deveria ser permitida.

8.1 Conceitos fundamentais

A partir dos anos 1980, surgiu um problema quando programadores e projetistas de linguagem começaram a perceber que um número enorme de aplicações não era bem resolvido pela aplicação do paradigma imperativo da decomposição funcional e abstração de dados. Para solucioná-lo, surgiu a programação orientada a objetos (POO) como um estilo popular de programação no qual a decomposição de objeto se tornou uma preocupação central, em vez da decomposição funcional e da abstração de dados (TUCKER; NOONAN, 2009).

A POO é uma maneira específica de pensar (um paradigma), independentemente do tamanho do programa. Mas ela é especialmente útil para programas maiores.

Um modo importante de entender o funcionamento de um programa orientado a objetos é perceber que ele é uma coleção de objetos, que interagem entre si e se comunicam através da troca de mensagens. Cada um deles pode ser visto como uma máquina separada que possui dados e realiza operações sobre os dados. Uma das vantagens da POO é que vários objetos podem ser instâncias distintas de uma mesma classe, sendo que as operações são as mesmas, operando sobre diferentes dados (TUCKER; NOONAN, 2009).

Para que o desenvolvedor de software possa usufruir dos benefícios da POO, são necessários o entendimento e a aplicação prática de um conjunto de conceitos teóricos, que são a base da orientação a objeto. Eles devem ser aplicados durante a codificação que faz parte do processo de desenvolvimento de software.

A POO é considerada uma abordagem de desenvolvimento mais intuitiva, pois ela parte do princípio de que o mundo é composto de uma coleção de objetos. Quando o desenvolvedor projeta um sistema, ele deve partir do princípio de que todos os objetos existentes no mundo real precisam ser representados no computador de maneira análoga. Esse conjunto de objetos modelados interage entre si no sistema.

Por exemplo, temos um requisito de software que é calcular o valor do imposto ICMS sobre um produto vendido. No sistema, podemos representar o imposto como um objeto, o produto como outro e, para programar o método que fará o cálculo desse imposto, precisamos fazer com que os dois objetos interajam para poder calcular o ICMS.

Classes e objetos

A POO apoia o encapsulamento dos tipos de dados com suas funções e a ocultação de informação da abstração de dados. Nela, toda a estrutura do programa é organizada em classes, que é uma declaração de tipo que encapsula constantes, variáveis e funções para manipulação dessas variáveis (TUCKER; NOONAN, 2009).

Uma classe é um tipo de dado abstrato e um mecanismo para o definir em um programa. Nos conceitos da POO, as variáveis locais de uma classe são conhecidas como variáveis de instância e suas inicializações são feitas por meio de funções especiais chamadas de construtores, ao passo que suas finalizações (liberação de memória, eventualmente fechamento de conexão com o banco de dados ou arquivo etc.) são conhecidas por destruidores, e outras funções são implementadas por métodos (TUCKER; NOONAN, 2009).

As classes que podem ser representadas semanticamente por um retângulo contendo três divisões são:

- **Nome da classe**: carro.
- **Propriedades ou atributos da classe**: marca do carro, placa do carro.
- **Métodos da classe**: ligar o carro() e brecar o carro().

Quadro 12 – A classe do carro

Carro
+ Marca do carro
+ Placa do carro
+ Ligar o carro()
+ Brecar o carro()

Uma classe é como uma planta de um prédio ou, no nosso exemplo, um projeto de um carro. A partir de uma planta, podemos construir vários edifícios do mesmo tipo. A partir do projeto de um carro, podemos construir vários deles. Por meio das classes, especificamos as propriedades (atributos) e os métodos (comportamentos) para um conjunto de objetos semelhantes, que possuem semelhantes propriedades e comportamentos. Então, as classes agrupam ou são uma abstração para os objetos que são similares, pois possuem as mesmas propriedades (atributos) e comportamentos (métodos).

Na POO, os atributos são utilizados para armazenar os dados, ao contrário da programação estruturada, em que as variáveis estão "soltas" dos subprogramas ou funções, enquanto na POO elas estão "amarradas". A codificação do comportamento do programa é realizada pelos métodos, que, na programação estruturada similarmente, seria executada pelos subprogramas contendo funcionalidades específicas.

Em resumo, as classes permitem definir um tipo de objeto. Diz-se que todo objeto é uma instância de uma classe. Para instanciar um objeto, geralmente é utilizado na LP o comando new. Por exemplo, o comando a seguir cria um objeto chamado Luiz, do tipo Pessoa, na memória do computador:

- Pessoa Luiz = new Pessoa();

Os objetos são representados por determinada classe e diferenciam-se pelos valores de seus atributos. Cada um deles possui uma identidade única e uma existência própria. São diferentes mesmo que suas propriedades e comportamentos sejam iguais. Por exemplo: o carro do Luiz possui um identificador único, e é uma instância da classe Carro.

Na POO, o foco da programação está nos objetos que trocam mensagens entre si, através da execução dos métodos que realizam os algoritmos ou atividades necessárias ou serviços requisitados pelas mensagens.

Os objetos se comunicam trocando mensagens entre eles, similarmente como uma máquina (objeto) que pode estar executando uma mensagem de cada vez. Esse objeto base passa uma mensagem para outra máquina receptora, ela invoca um método, que pode ser uma função, e envia os parâmetros, que são o conteúdo da mensagem apropriados para o método e, então, aguarda até que a máquina receptora retorne uma resposta, que é o chamado valor de retorno.

Uma linguagem baseada em objetos normalmente se caracteriza como um conjunto de comportamentos que permite a um programa criar qualquer quantidade de instâncias de um tipo de dado abstrato. Trata-se de uma forma de pensar em programação, pois os dados tornam-se ativos não sofrendo mais passivamente o efeito das funções.

Nas LPs que dão suporte ao POO, as classes desempenham dois papéis fundamentais: em primeiro lugar, apresentam os tipos de um objeto, por exemplo, tipo pessoa ou tipo televisor, que definem os métodos e a sua visibilidade. Depois, elas possibilitam a verificação de tipo, por exemplo, em Java, que é uma linguagem fortemente tipada, essas verificações são realizadas em tempo de compilação, enquanto

em uma linguagem como Smalltalk, que é dinamicamente tipada, as verificações são feitas em tempo de execução (TUCKER; NOONAN, 2009).

Em uma linguagem fortemente tipada, existem duas maneiras de executar a verificação de tipo: os tipos de dados dos parâmetros da mensagem devem ser verificados em relação aos parâmetros formais do método e o tipo de retorno do método precisa ser verificado em relação ao tipo esperado da mensagem (SEBESTA, 2011).

> A verificação de tipo é a fase da análise semântica ou verificador de tipo. Esta fase é responsável por garantir que as regras de semântica de tempo de compilação da linguagem sejam impostas com as seguintes condições:
>
> - Que todos os identificadores referenciados no programa sejam declarados.
> - Que os operandos para cada operador possuam um tipo apropriado.
> - Que as operações de conversões envolvidas, por exemplo, inteiro para float, sejam inseridas onde for necessário (TUCKER; NOONAN, 2009, p. 47).

Observação

O **tempo de compilação** ocorre quando o programa é compilado, os comandos e as expressões desses programas são convertidos em sequências de instruções em linguagem de máquina equivalente para que ele se torne um executável.

Já o **tempo de execução** acontece quando o programa está sendo executado, ou seja, depois que ele foi compilado, os valores são atribuídos às variáveis, como na execução da atribuição x = 1 (TUCKER; NOONAN, 2009).

Exemplo: a execução do método Brecar o Carro, faz com que o veículo pare de se locomover em determinado instante, então os objetos são um agrupamento de atributos e métodos (comportamentos comuns) e:

- Representam uma instância da classe na memória do computador.
- Significam uma entidade abstraída do mundo real, que possui várias características e pode executar ações.
- No mundo real, diferentes objetos podem possuir semelhantes atributos e operações.

Quadro 13 – Classe × Objeto

Classe	Objeto
Carro	**BFGHJK123456: Carro**
+ Marca do carro	+ Marca do carro: Corsa
+ Placa do carro	+ Placa do carro: AAA1234
+ Ligar o carro()	+ Ligar o carro()
+ Brecar o carro()	+ Brecar o carro()

Atributos e métodos

Os atributos representam a segunda divisão no nosso diagrama de classe. Os valores dos atributos ou propriedades definem o estado do objeto. Esse conjunto de propriedades estabelece as informações ou dados que configuram ou caracterizam um objeto e armazenam seus valores.

Observe na sequência exemplos de propriedades para um objeto Carro:

- Ano modelo do carro: 2015.
- Cor do carro: azul.
- Proprietário: Luiz Roberto.

A terceira divisão no nosso diagrama de classe representa os métodos, também conhecidos no jargão de TI como assinaturas dos métodos das classes. Por exemplo: Ligar o carro (), Brecar o carro ().

Na POO, os métodos definem o comportamento do objeto, que oferece e recebe serviços dos demais objetos por meio de troca de mensagens. Cada objeto possui seu próprio conjunto de serviços ou operações que podem ser requisitados por outros. Um objeto pode enviar uma mensagem, ou chamada de um método a outro objeto, contendo a identificação dos objetos e, em determinados casos, os seus parâmetros. Esse conceito é muito semelhante à chamada de procedimentos e funções no paradigma estruturado.

Observe na sequência exemplos de métodos para um objeto Carro:

- Ligar o carro ().
- Brecar o carro ().
- Acender os faróis ().
- Trocar de marcha ().

Encapsulamento

Uma linguagem é considerada orientada a objetos se ela suportar um mecanismo de encapsulamento que esconda informações para definir tipos abstratos de dados, herança e métodos virtuais.

> Encapsulamento é um mecanismo que permite que constantes, tipos, variáveis, métodos, entre outros, fiquem relacionados logicamente, podendo ser agrupados em uma nova entidade como: classes, pacotes e procedimentos. A abstração de dados estende a noção de tipo, possibilitando ao programador utilizar um mecanismo de encapsulamento para definir novos tipos de dados, além daqueles fornecidos pela LP, dentro da aplicação que está desenvolvendo (TUCKER; NOONAN, 2009, p. 310).

O encapsulamento de dados em uma aplicação permite restringir o acesso a variáveis e métodos da classe, ou até mesmo à própria classe. Para isso, é criada uma estrutura de restrição do acesso direto a alguns dos componentes de um objeto, por meio de métodos que podem ser utilizados por qualquer outra classe, sem causar inconsistências no desenvolvimento de um código. Os detalhes de como o código foi implementado ficam ocultos ao usuário da classe, que passa a utilizar os serviços dela sem saber como isso ocorre internamente. Somente é apresentada ao programador uma lista das funcionalidades existentes (FURGERI, 2015).

Para garantir o princípio do encapsulamento, os atributos de uma classe somente podem ser acessados única e exclusivamente por meio de métodos públicos, garantindo maior segurança no acesso aos dados da classe.

Utiliza-se como padrão os métodos manipuladores Set para atribuir um valor a propriedade da classe e Get para recuperar um valor da propriedade da classe. Em geral, deve-se usar o modificador 'private' para os atributos e 'public' para os métodos.

O processo conhecido como encapsulamento determina que, para acessar esses tipos de modificadores, é necessário criar métodos setters e getters. Atualmente, por se tratar de um padrão de desenvolvimento de software fortemente estabelecido, diversos frameworks e IDEs já criam por padrão e de forma automática essa estrutura, bastando selecionar quais as propriedades, que o próprio framework se incube de criar todo o código para dar suporte aos getters e setters, reduzindo o esforço de programação.

Lembrete

Framework é um termo inglês que, em sua tradução direta, significa estrutura. Na programação de software, um framework de desenvolvimento fornece partes comuns de códigos genéricos, evitando que os desenvolvedores percam tempo lidando com funções e bibliotecas comuns e repetitivas.

Na sequência temos como exemplo a classe Pessoa. Nela, basta o desenvolvedor codificar os dois atributos, nome e RG, e solicitar ao framework para criar os getters e setters, que automaticamente serão criados os quatro métodos getNome(), setNome(String nome). getRg() e setRg(int rg).

Consta a seguir um exemplo de implementação de encapsulamento automático:

```
public class Pessoa {
   private String nome;
   private int rg;
//getters e setters criados automaticamente pelo framework

 public String getNome() {
       return nome;
   }

  public void setNome(String nome) {
       this.nome = nome;
   }

 public void setRg(int rg) {
       this.rg = rg;
   }


 public int getRg() {
       return rg;
   }
}
```

Uma vez criada a classe Pessoa (com a letra P em maiúscula), que contém o encapsulamento, somente será possível inserir dados no objeto pessoa (com letra p minúscula), ou obter dados deste objeto por meio dos métodos getters e setters. Para inserir os dados no objeto pessoa, utilizam-se os métodos SetNome e SetRG. A fim de recuperar os dados no objeto pessoa para imprimir no console por meio do método println, usam-se os métodos getNome e getRG.

A seguir um exemplo de implementação acessando o encapsulamento:

```
public class usaPessoa {

   public static void main(String[] args) {
      Pessoa pessoa = new Pessoa();
      pessoa.setNome("Luiz Roberto");
      pessoa.setRg(123456789);
```

```
        System.out.println(funcionario.getNome());
        System.out.println(funcionario.getRg());

    }
}
```

Herança

Trata-se de um recurso no qual uma subclasse herda as propriedades (atributos) e métodos da superclasse, também chamada de classe Pai ou classe Mãe, aumentando as possibilidades de reutilização de código, diminuição do tempo e do custo de desenvolvimento.

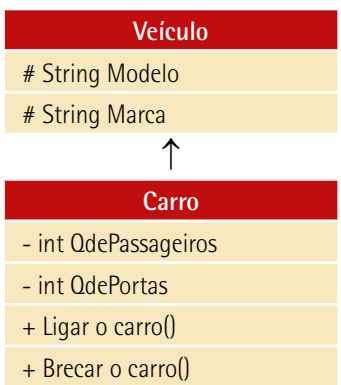
Uma vez criada, uma classe pode ser reutilizada no mesmo programa ou em outro.

A subclasse ou classe filha, além de herdar da classe Pai, pode implementar seus próprios atributos e métodos, ou mesmo redefinir métodos da classe Pai com novas implementações, promovendo inúmeras possibilidades de reutilização e polimorfismo (vide a explicação desse conceito mais adiante).

A herança é chamada de simples quando ela herda apenas de uma classe e de múltipla quando herda de mais de uma classe.

Veja um exemplo no quadro a seguir. Nele, a classe Carro, por meio do mecanismo de herança, pode utilizar os atributos e métodos da classe Veículo. Observe a ponta de seta que sai da classe Carro e vai em direção à classe Veículo, simbolizando a herança entre elas.

Quadro 14 – Exemplo de herança



A seguir, mostraremos um exemplo da implementação da classe Carro em Java. Observe que ela utiliza o comando extends para estender a superclasse Veículo, acessando e podendo utilizar a partir desse momento todos os seus atributos e métodos. Veja o uso das referências this e super, o termo "this" faz associação às propriedades do objeto corrente, enquanto "super" tem relação com as propriedades da superclasse. Por meio do método construtor da classe Carro, cadastramos em ambas as classes a informação referente ao carro.

```
public class Veiculo{
  public String modelo;
  public String marca;
  public Veiculo(String modelo, String marca) {
    this.modelo = modelo;
    this.marca = marca;
  }
}

public class Carro extends Veiculo{
  public int qdePassageiros;
  public int qdePortas;
  public Carro(String modelo, String marca, int qdePassageiros, int qdePortas) {
    super(modelo);
    super(marca);
    this.qdePassageiros = qdePassageiros;
    this. qdePortas = qdePortas;
  }
}

public class usaCarro {

   public static void main(String[] args) {
      Carro carro = new Carro("Renegade", "Jeep", 5, 4);
   }
}
```

Visibilidade (restrições de acesso para atributos e métodos)

Os atributos e métodos possuem visibilidade. No quadro a seguir, veremos os símbolos #, de visibilidade protegida, antes de cada atributo da classe Veículo, os símbolos -, de visibilidade privada, antes de cada atributo da classe Carro e os +, de visibilidade pública, antes de cada método dessa classe. Isso significa que:

- as propriedades Modelo e Marca podem ser utilizadas pelas classes Carro e Veículo;
- as propriedades QdePassageiros e QdePortas somente podem ser acessadas pela classe Carro;
- os métodos Ligar o carro e Brecar o carro são públicos e podem ser acessados de qualquer parte do sistema.

Quadro 15 – Visibilidade de atributos e métodos

A seguir um exemplo de implementação no código-fonte utilizando a linguagem Java de herança:

```
public class Veiculo {
   protected String modelo;
   protected String marca;
}

public class Carro extends Veiculo {
    private int qdeDePassageiros;
         private int qdeDePortas;
    public void ligarCarro(){
    }

   public void brecarCarro(){
    }
}
```

Polimorfismo

É a capacidade de um objeto de assumir diferentes formas. Através do uso das técnicas de encapsulamento e herança, novas implementações podem ser adicionadas às classes.

A partir do momento que uma classe foi herdada, os métodos herdados dela podem ser redefinidos para um novo algoritmo, permitindo a reutilização de código e consequentemente a redução no tempo e custo do desenvolvimento.

O polimorfismo permite implementar a generalização e a herança à especialização. Cada classe pode assumir o mesmo comportamento da superclasse, assim uma classe da hierarquia pode assumir diferentes formas (funcionalidades) de acordo com as classes de nível superior. O uso do polimorfismo gera uma economia de código-fonte. Por meio desse recurso, uma classe mais genérica (a superclasse) pode assumir diferentes comportamentos, gerando objetos distintos, dependendo de certas condições.

Na prática, o polimorfismo permite também que vários métodos possam existir com o mesmo nome, porém com implementações diferentes (FURGERI, 2015).

A seguir demonstraremos um exemplo de implementação de polimorfismo em Java no qual a superclasse Pessoa possui o método polimórfico exibeNomeClasse (FURGERI, 2015). Em seguida, as subclasses PessoaFisica e PessoaJuridica sobrescrevem ou redefinem o método herdado, porém em cada uma delas esse método imprime uma mensagem diferente, referente à sua classe. Na classe principal, é declarado um objeto do tipo Pessoa, porém ele é inicializado com null, o que significa que nesse ponto ele ainda não foi criado. De acordo com a opção escolhida pelo usuário no programa, o objeto Pessoa é criado a partir de uma das classes disponíveis na estrutura: Pessoa, PessoaFisica ou PessoaJuridica. Todas as classes da estrutura possuem o mesmo método exibeNomeClasse que é sobrescrito, imprimindo a mensagem respectiva, demonstrando aqui o polimorfismo, pois se tratam de objetos diferentes com comportamentos específicos (FURGERI, 2015).

```
public class Pessoa {
    public void exibeNomeClasse(){
        System.out.println("Super Classe Pessoa");
    }
}

public class PessoaFisica extends Pessoa {
    public void exibeNomeClasse(){
        System.out.println("Classe Pessoa Física");
    }
}

public class PessoaJuridica extends Pessoa {
    public void exibeNomeClasse(){
        System.out.println("Classe Pessoa Jurídica");
    }
}
import javax.swing.JOptionPane;
public class PessoaPolimorfica {
    public static void main(String[] args) {
        Pessoa pessoa = null;
      int classePessoa = Integer.parseInt(JOptionPane.showInputDialog("Digite um número de 1 a 3"));
        switch(classePessoa){
            case 1: pessoa = new Pessoa(); break;
            case 2: pessoa = new PessoaFisica(); break;
            case 3: pessoa = new PessoaJuridica(); break;
            default:
                System.out.println("Tipo pessoa não informado");
                System.exit(0);
```

```
        }
        pessoa.exibeNomeClasse();
    }
}
```

Classes abstratas

Segundo Sebesta (2011), uma classe que possui ao menos um método abstrato é chamada de classe abstrata. Ela não permite ser instanciada, ou seja, não podem ser criados objetos a partir dela. Os seus métodos são apenas declarados, mas não definidos, isto é, não têm implementação. Uma subclasse, ao criar uma instância de uma classe abstrata, deve implementar, ou seja, definir o algoritmo para todos os métodos abstratos herdados.

Uma classe abstrata é utilizada como base para a elaboração de outras classes. Por exemplo, as classes Pessoa Física e Pessoa Jurídica possuem atributos comuns como nome e telefone, porém elas também têm atributos que somente pertencem a elas, como CPF para pessoa física e CNPJ para pessoa jurídica. Portanto, podemos criar a classe abstrata pessoa com os atributos comuns a ambas e um método abstrato sem implementação para gravar uma pessoa, com o seguinte trecho de código desenvolvido em Java:

```
public abstract class Pessoa {
      // Atributos da classe
         String nome;
         String telefone;
      // Método abstrato
      public abstract void GravarPessoa();
}
```

A classe abstrata ainda pode ser utilizada para definir um comportamento padrão para outras classes, no nosso exemplo o método GravarPessoa poderá ser usado por qualquer outra classe que herde a classe abstrata Pessoa, como demonstra o seguinte trecho de código:

```
public class PessoaFisica extends Pessoa {
   String CPF;
   @Override
   public void GravarPessoa() {
      nome = "Luiz";
      telefone = "11111111";
      CPF = "000.000.000-00";
      System.out.println("Nome: " + nome + ", Telefone: " + telefone + " CPF: " +     CPF + ". Gravado
com sucesso!");
   }
}
```

```
public class PessoaJuridica extends Pessoa {
   String CNPJ;
   @Override
   public void GravarPessoa() {
      nome = "Grupo Empresarial Luiz";
      telefone = "11111111";
      CNPJ = "000.000.000/0001-00";
      System.out.println("Nome: " + nome + ", Telefone: " + telefone + " CNPJ: " + CNPJ + ". Gravado com sucesso!");
   }
}
```

Observe que o atributo CPF foi declarado na subclasse PessoaFisica, enquanto o CNPJ foi na subclasse PessoaJuridica. Ambas as subclasses utilizam os atributos nome e telefone que foram declarados na classe abstrata Pessoa e implementam (codificam) o método abstrato GravarPessoa, que é comum às duas subclasses. A classe abstrata Pessoa somente define o que as subclasses devem implementar (FURGERI, 2015).

Para exemplificar o funcionamento de uma classe abstrata, criaremos uma classe de teste, conforme o seguinte trecho de código:

```
public static void main(String[] args) {
   PessoaFisica pf = new PessoaFisica();
   pf.GravarPessoa();

   PessoaJuridica pj = new PessoaJuridica();
   pj.GravarPessoa();
}
```

A execução desse programa imprime a seguinte mensagem:

- Nome: Luiz, Telefone: 11111111 CPF: 000.000.000-00. Gravado com sucesso!

- Nome: Grupo Empresarial Luiz, Telefone: 11111111 CNPJ: 000.000.000/0001-00. Gravado com sucesso!

Interface

De acordo com Tucker e Noonan (2009, p. 582), "uma interface encapsula uma coleção de constantes e assinaturas de métodos abstratos. Uma interface não pode incluir variáveis, construtores ou métodos não abstratos".

A interface define apenas a especificação de uma classe, e não a maneira como ela será implementada (codificada). Ela funciona de maneira semelhante a classes abstratas, mas não permite a implementação de nenhum método, pois apenas declara um ou mais métodos (comportamento) que a classe que implementa a interface deve ter (FURGERI, 2015).

Uma classe implementa uma interface, ela não herda como é feito na classe abstrata. Logo, uma classe pode implementar uma ou mais interfaces (SEBESTA, 2011).

Nos projetos de software, a interface é muito útil para que os desenvolvedores sejam obrigados a seguir um padrão de projeto, por exemplo, a de nomenclatura de métodos, pois as classes que implementam uma interface são obrigadas a implementar os métodos definidos na interface.

O seguinte trecho de código, desenvolvido em Java, exemplifica a declaração de uma interface para dois métodos, Somar e Subtrair. Observe que eles possuem apenas a especificação, mas não têm implementação (codificação):

```
public interface ICalculadora {
    public int Somar(int valor1, int valor2);
    public int Subtrair(int valor1, int valor2);
}
```

No caso do Java, observe que é utilizada a palavra reservada interface no lugar de class. Conforme já dissemos, a classe Calculadora que implementa a interface ICalculadora é obrigada a implementar os métodos definidos na interface, codificando o algoritmo necessário para que o método cumpra com o seu objetivo, conforme o seguinte trecho de código:

```
public class Calculadora implements ICalculadora{
    @Override
    public int Somar(int valor1, int valor2) {
        return valor1 + valor2;
    }
    @Override
    public int Subtrair(int valor1, int valor2) {
      return valor1 - valor2;
    }
}
```

A classe calculadora poderia também implementar outros métodos além dos definidos na interface. No caso do Java, foi utilizada a palavra reservada implements para vincular a classe à interface.

Para exemplificar o funcionamento de uma interface, criaremos uma classe de teste, conforme o seguinte trecho de código:

```
public static void main(String[] args) {
    ICalculadora calculadora = new Calculadora();
    int resultadoSoma = calculadora.Somar(2, 2);
    System.out.println(resultadoSoma);

   int resultadoSubtracao = calculadora.Subtrair(2, 2);
   System.out.println(resultadoSubtracao);
}
```

8.2 Caracterização e comparação

A POO permite a modularização e a separação da aplicação e dos objetos em camadas. Um programa OO pode ter um, ou mais objetos, e ser criado (instanciado) ou destruído em tempo de execução.

Na programação estruturada, a aplicação possui uma estrutura na qual as funcionalidades (procedimentos e funções) e os dados (as variáveis) são organizados em um único bloco.

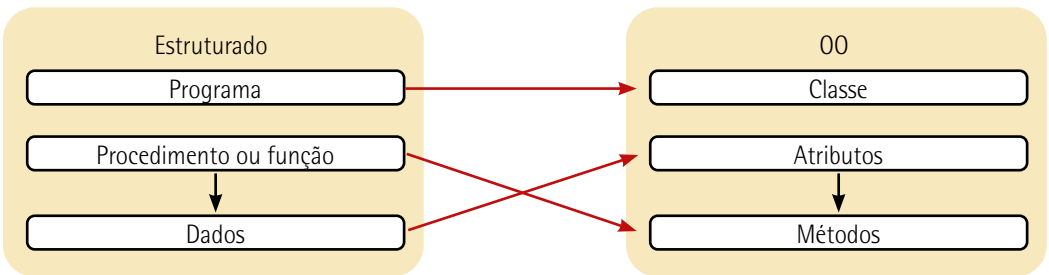


Figura 34 – Estrutura dos paradigmas estruturado e OO

O pensamento de um desenvolvedor OO deve se voltar para a própria forma de olhar a estrutura de um programa. Enquanto na programação estruturada o desenvolvedor se preocupa em dividir o programa em módulos com procedimentos e funções, na programação OO a modularização é realizada para a resolução do problema, por meio das classes. O quadro a seguir mostra a diferença de conceitos entre os paradigmas orientados a objetos e a estruturada.

Quadro 16 – Diferenças de conceitos entre paradigmas OO e estruturada

Programação OO	Programação estruturada
Atributos ou propriedades	Variáveis
Métodos das classes	Procedimentos ou funções
Troca de mensagens entre objetos	Chamadas a procedimentos e funções
Abstração de dados criados pelo desenvolvedor	-
Polimorfismo	-
Herança	-
Criação de objetos como instâncias da classe	-

Os dados na programação estruturada são acessados na memória do computador por meio dos procedimentos e funções, de forma totalmente dividida. No paradigma orientado a objetos, os dados são encapsulados nos objetos e protegidos pelos métodos deles. O acesso aos atributos ou dados é realizado apenas pelos métodos implementados na própria classe, aplicando-se o conceito de encapsulamento.

Na programação estruturada, os procedimentos realizam operações com os dados compartilhados. Os dados são acessados diretamente referenciados pelos nomes das variáveis. Na programação OO, as operações são realizadas por meio de um conjunto de objetos que interagem entre si através de "trocas de mensagens" ou execução dos métodos existentes em cada objeto. Um serviço é solicitado de um objeto a outro através de uma mensagem, conforme a figura a seguir.

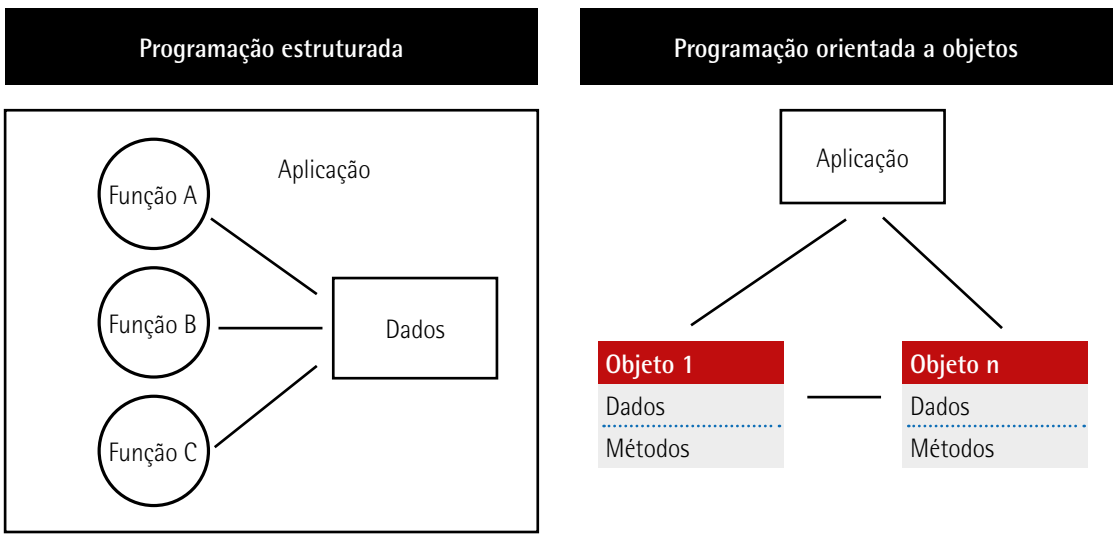


Figura 35 – Dados nos paradigmas estruturado e OO

8.3 Linguagens orientadas a objetos (Smalltalk, C++, Java, C#)

Entre as LPs orientada a objetos temos a linguagem Smalltalk, na qual seus componentes são considerados objeto: classes, métodos, números etc. Nessa linguagem, não existem tipos primitivos como acontece em outras linguagens orientadas a objetos. Em Smalltalk, números, strings e caracteres são implementados como classes. Programas escritos na linguagem Smalltalk podem ser codificados em ambientes de desenvolvimento modernos (IDE) como Pharo, Squeak e VisualWorks. Algumas linguagens que dão suporte à orientação a objetos e seus respectivos anos de criação são:

- Smalltalk (1972).
- Ada (1983).
- Eiffel (aproximadamente 1985).
- C++ (1986).

- Object-Pascal (1986).
- Common Lisp (1986).
- Java (1991).
- C# (2000).

Exemplo de aplicação

Para exemplificar os conceitos essenciais do paradigma de programação orientada a objetos nas LPs Smalltalk, resolveremos um problema contido no site Beecrowd. Ele disponibiliza um repositório de problemas de programação no qual o programador pode desenvolver um algoritmo. O site testa a solução e informa se ela está correta ou não.

1) Leia 2 valores de ponto flutuante de dupla precisão A e B, que correspondem a duas notas de um aluno. A seguir, calcule a média do aluno, sabendo que a nota A tem peso 3.5 e a nota B tem peso 7.5 (a soma dos pesos portanto é 11). Assuma que cada nota pode ir de 0 até 10.0, sempre com uma casa decimal.

Adaptado de: https://bit.ly/3ux1kAx. Acesso em: 24 jan. 2022.

Segue o trecho de código com a solução em Smalltalk:

```
| nota1 nota2 mediaPond |
nota1 := UIManager default request: 'Entre com a primeira nota'.
nota2 := UIManager default request: 'Entre com a segunda nota'.
mediaPond := (( nota1 * 3.5) + ( nota2 * 7.5)) / (3.5 + 7.5).
Transcript show: mediaPond; cr.
```

Veja a seguir funcionalidades comentadas sobre os comandos do Smalltalk utilizadas no trecho de código do exemplo anterior:

- As variáveis são declaradas utilizando pipes (|).
- Para atribuirmos um valor a uma variável, necessitamos utilizar dois-pontos igual (:=).
- O Smalltalk possui um objeto chamado de Transcript, similar a uma tela de console, onde é possível exibir as mensagens com o comando Transcript show.
- O método cr solicita ao Transcript que pule de linha.

2) Faça uma pesquisa na internet com o objetivo de descobrir outras funcionalidades úteis da linguagem Smalltalk e para se aprofundar na LP orientada a objetos.

Saiba mais

A solução para este problema foi desenvolvida por meio do uso do software Squeak, que pode ser encontrado no seguinte endereço:

Disponível em: https://squeak.org/. Acesso em: 24 jan. 2022.

Resumo

Nesta unidade, estudamos os principais paradigmas utilizados no mercado: paradigma estruturado, orientado a eventos e orientado a objetos.

No paradigma estruturado, vimos que o programa e as variáveis são armazenados em um mesmo lugar. O programa contém um conjunto de comandos para manipular variáveis, realizar cálculos, receber e enviar dados para o usuário ou outros programas. Trata-se de um paradigma eficiente, dominante e bem estabelecido no mercado, facilita a modelagem das aplicações do mundo real, possui fácil entendimento e por isso é bastante utilizado em cursos introdutórios de programação.

Como suas desvantagens, percebemos que ele pode dificultar a legibilidade do código, facilitando a geração de códigos confusos quando o tratamento dos dados é misturado ao comportamento do programa, dificultando a sua manutenção. Algumas das LPS que dão suporte a esse paradigma são: Basic, C, PHP, Pascal e Python.

Entendemos que o paradigma orientado a eventos torna a programação de computadores mais flexível, pois permite ao desenvolvedor projetar visualmente a aparência do software de acordo com as necessidades do projeto.

Os objetos possuem uma série de eventos que podem ser programados quando forem ativados pelo usuário ou por outro sistema. O desenvolvedor tem mais controle sobre as ações do usuário e o fluxo de execução do programa.

As vantagens do paradigma orientado a eventos são a flexibilidade e a utilização de interfaces gráficas, o que torna a programação mais simples e intuitiva, facilitando o desenvolvimento do software. Algumas das linguagens que dão suporte a esse paradigma incluem: Visual Basic, Java e Python.

Observamos que o paradigma orientado a objetos teve uma ótima aceitação pelo mercado de software, pois permite produtividade e qualidade na construção de sistemas simples e complexos. Ele possui técnicas, métodos, e ferramentas que possibilitam conduzir o processo de desenvolvimento de software desde a análise de requisitos até a implementação do software em produção.

Demonstramos que a utilização da POO permite a melhoria de comunicação entre o time de desenvolvedores e clientes, uma vez que a mesma notação é utilizada em todas as fases. Seu uso proporciona ao desenvolvedor dedicar mais tempo à fase de análise e entendimento do negócio do cliente, preocupando-se com a essência do sistema, o que reduz a quantidade de erros, diminuindo a duração nas etapas de codificação, teste e implementação.

Vimos que as principais linguagens que dão suporte a OO são: Smalltalk, Ada, Eiffel, Object Pascal, Common Lisp, C++, C#, Java e Python.

Exercícios

Questão 1. Considere o programa simples mostrado a seguir, escrito na LP Java.

```
class Numero {
   int valor;
   Numero(int valor){
      valor = valor + 30;
      this.valor=valor;
   }
   public void imprime(int valor){
      valor = 30;
      System.out.println("Resultado: " + this.valor);
   }
}
public class Main {
   public static void main(String[] args) {
      int valor = 20;
      Numero num = new Numero(40);
      num.imprime(50);
   }
}
```

Assinale a alternativa que corresponde à saída deste programa após a sua execução.

A) Resultado: 20.

B) Resultado: 30.

C) Resultado: 40.

D) Resultado: 50.

E) Resultado: 70.

Resposta correta: alternativa E.

Análise das alternativas

A) Alternativa incorreta.

Justificativa: ainda que o número 20 seja associado a uma variável local chamada "valor", ela não é utilizada em nenhum ponto do programa.

B) Alternativa incorreta.

Justificativa: a variável "valor" do método "imprime" é igual a 30. No entanto, a saída na tela não é dela, mas do atributo "valor" da classe "Numero". Ainda que o atributo e a variável local do método "imprime" possuam o mesmo nome, elas correspondem a entidades diferentes.

C) Alternativa incorreta.

Justificativa: o número 40, passado pelo construtor da classe "Numero", é somado a outro maior do que zero antes de ser associado ao atributo "valor" da classe "Numero".

D) Alternativa incorreta.

Justificativa: o número 50 é passado como argumento para o método "imprime", mas não é utilizado.

E) Alternativa correta.

Justificativa: inicialmente, adiciona-se 30 ao argumento do construtor da classe "Numero". Isso corresponde à operação 30+40=70. Posteriormente, esse valor é associado ao atributo "valor". Finalmente, ele é utilizado pelo método "println", que será mostrado na tela concatenado com a cadeia de caracteres "Resultado:".

Questão 2. Suponha que se queira criar um programa no qual existem duas classes. A primeira classe corresponde a uma letra normal do alfabeto. A segunda classe corresponde apenas às vogais. As vogais também são letras, mas são um conjunto mais restrito: a, e, i, o, u. O programa deve considerar tanto letras maiúsculas quanto letras minúsculas. Para criar esse programa, deseja-se utilizar o conceito de herança, sendo que a classe correspondente às vogais pode ser derivada da classe correspondente às letras mais genéricas. Foi solicitado a um desenvolvedor que escrevesse um programa que ilustrasse essa hierarquia na linguagem Java, e ele escreveu o seguinte código-fonte:

```
class Letra {
  protected char valor;
  Letra(char valor){
     if(Letra.ehValido(valor)){
        this.valor=valor;
     }else{
        throw new IllegalArgumentException("Não é uma letra!");
     }
  }
  protected static boolean ehValido(char simbolo){
    return Character.isLetter(simbolo);
  }
  public char retornaSimbolo()
  {
     return this.valor;
  }
}
class Vogal extends Letra {
  Vogal(char valor){
     super(valor);
     if(Vogal.ehValido(valor)){
        this.valor=valor;
     }else{
        throw new IllegalArgumentException("Não é uma vogal!");
     }
  }
  protected static boolean ehValido(char simbolo){
     String vogais="AEIOUaeiou";
     for(int pos=0;pos<vogais.length();pos++){
       if(vogais.charAt(pos)==simbolo){
          return true;
       }
     }
     return false;
  }
}
class Main {
  public static void main(String[] args) {
    System.out.println("Inicio 1");
    Letra s1= new Letra('r');
    Letra s2= new Vogal('E');
    System.out.println(s1.retornaSimbolo());
    System.out.println(s2.retornaSimbolo());
    Letra s3= new Vogal('q');
    System.out.println(s3.retornaSimbolo());
  }
}
```

Com base nesse código, avalie as afirmativas.

I – O código apresentado compila, mas, ao ser executado, gera uma exceção, disparada pela linha "throw new IllegalArgumentException("Não é uma vogal!");", chamada pelo construtor da classe Vogal. No método "main", isso corresponde à linha "Letra s3= new Vogal('q');". Esse comportamento é esperado, uma vez que a letra "q" não é uma vogal.

II – O programa apresentado não compila.

III – O programa compila e executa normalmente, sem gerar nenhuma exceção. A saída do programa é: "rEq".

É correto o que se expõe em:

A) I, apenas.

B) II, apenas.

C) III, apenas.

D) II e III, apenas.

E) I, II e III.

Resposta correta: alternativa A.

Análise das afirmativas

I – Afirmativa correta.

Justificativa: o construtor da classe "Vogal" chama o construtor da classe "Letra" pelo comando "super(valor)". Com isso, o programa verifica, inicialmente, se o valor passado como argumento é uma letra e, posteriormente, se o valor é uma vogal. Como a letra q não é uma vogal, o construtor da classe "Vogal" dispara uma exceção.

II – Afirmativa incorreta.

Justificativa: o programa apresentado compila, pois segue a sintaxe da linguagem Java.

III – Afirmativa incorreta.

Justificativa: a execução desse programa gera uma exceção, disparada pelo construtor da classe "Vogal", invocado na linha "Letra s3= new Vogal('q');", uma vez que a letra q não é uma vogal. Contudo, essa exceção não é tratada pelo código.

REFERÊNCIAS

Textuais

DAURICIO, J. S. *Algoritmos e lógica de programação*. Londrina: Educacional, 2015.

ETZION, O.; NIBLETT, P. *Event processing in action*. Stamford: Manning, 2011.

FURGERI, S. *Java 8* – Ensino didático: desenvolvimento e implementação de aplicações. São Paulo: Érica, 2015.

INVERTEXTO. *Código binário*. [s.d.]. Disponível em: https://bit.ly/3s20A38. Acesso em: 21 jan. 2022.

SEBESTA, R. W. *Conceitos de linguagens de programação*. 9. ed. Porto Alegre: Bookman, 2011.

TIOBE. *Tiobe index for January 2022*. Tiobe, 2022. Disponível em: https://bit.ly/3KQmxLq. Acesso em: 21 jan. 2022.

TUCKER, A. B.; NOONAN, R. E. *Linguagens de programação*: princípios e paradigmas. 2. ed. São Paulo: McGraw Hill, 2009.

UNIT-CONVERSION.INFO. *Convert text to binary*. [s.d.]. Disponível em: https://bit.ly/3FjeaUq. Acesso em: 11 jan. 2022.

VAREJÃO, F. M. *Linguagens de programação*: conceitos e técnicas. Rio de Janeiro: Elsevier/Campus, 2004.

Interativa

Informações:
www.sepi.unip.br ou 0800 010 9000