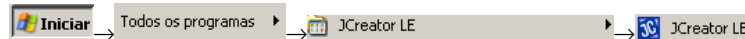


Exercício 0: Criando um primeiro programa no JCreator:

Para iniciar o JCreator aperte sempre o botão esquerdo do mouse, utilizando a seguinte sequência:



Após esta sequência irá aparecer a Figura 0.1.

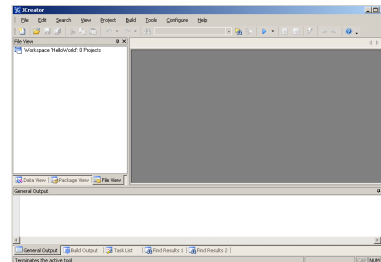
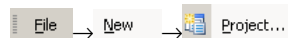


Figura 0.1: Ambiente de desenvolvimento JCreator.

Para criar um programa em Java deve-se empregar a seguinte sequência de botões:



Após esta sequência de comandos irá aparecer uma janela igual à Figura 0.2. Selecione nesta janela a opção "Empty Project", depois no campo nome digite "Lab1" e finalize a criação de um projeto. Observe que a imagem da janela no canto superior esquerdo da Figura 0.1 (Janela File View) será modificada para a Figura 0.3. Clique com o botão esquerdo sobre o ícone Lab1 e siga a sequência: Add, New Class e o nome da classe como HelloWorld.

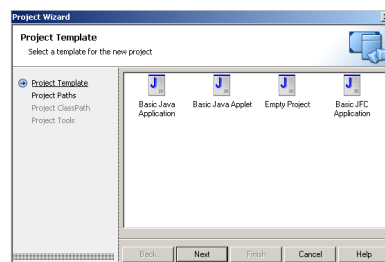


Figura 0.2: Opções de projeto no JCreator.



Figura 0.3: Detalhe da janela File View.

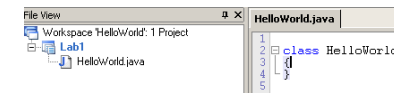


Figura 0.4: Criação da classe HelloWorld.

A Figura 0.4 mostra o que ocorre após a criação da classe HelloWorld. Observe na janela File View será incluído no Projeto Lab1 um arquivo de nome HelloWorld.java bem como será aberta uma janela onde pode ser inserido o código correspondente a classe HelloWorld do **PT1** (vide Figura 0.5).

```

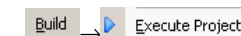
1: public class HelloWorld
2: { // Meu primeiro programa Java.
3:     public static void main( String args[] )
4:     {
5:         System.out.println("Hello World");
6:     }
7: }
  
```

Figura 0.5: PT1: Primeira classe em Java.

Após digitar o código do **PT1**, salve suas alterações e siga a sequência:



Se não houver erros de digitação, e conseqüentemente de compilação, então, selecione:



O resultado da execução deste programa deve ser tal como dado na Figura 0.6.

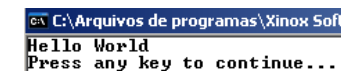


Figura 0.6: Resultado da execução de PT1.

Algumas observações importantes para o **PT1**:

- Um comentário pode ser inserido com // (uma linha) /* e */ (múltiplas linhas).
- As palavras-chave **public** e **class** introduzem declaração de uma classe definida pelo usuário. Por convenção todos os nomes de classes em Java iniciam com uma letra maiúscula, bem como a letra inicial de cada palavra (HelloWorld).
- As chaves { e } indicam tanto o corpo da classe (linhas 2 e 7) como o corpo do método main (linhas 4 e 6).

- A linha 3 (**public static void** main(**String** args[])) corresponde à declaração de um método. Classes podem ter um ou mais métodos. Para a execução de um aplicativo deve sempre existir pelo menos o método main. Como os métodos podem retornar informações, a palavra-chave **void** é empregada e nenhuma informação é devolvida. Este método possui ainda, entre parêntesis, parâmetros de entrada dados por **String** args[].
- O comando da linha 5 realiza a impressão de uma cadeia de caracteres contidas entre as aspas duplas, empregando um **objeto** de saída padrão, o **System.out**. Portanto, o método **System.out.println** exibe uma linha de texto na janela de comando. Observe que o cursor do teclado é posicionado para a próxima. É importante notar, também, que esta **instrução**, assim como todas as outras, é seguida de ponto-e-vírgula (;).

Exercício 1: O programa **P1** contém um arquivo Mensagem.class que contém uma classe Mensagem que possui um método mostraTexto1() que imprime uma string através do comando System.out.println tal como ilustrado na Figura 1.1. Além disso, o programa **P1** é composto do arquivo TestaMensagem.class que contém um método main() que cria um objeto da classe Mensagem e depois invoca o método mostraTexto1() (vide Figura 1.2), ilustrando o uso da classe Mensagem.

```
// Declaração da classe Mensagem com apenas um método.
// Esta classe deve estar em um arquivo Mensagem.class.
public class Mensagem
{
    // Exibe uma mensagem.
    public void mostraTexto1()
    {
        System.out.println(" O rato roeu a roupa do rei de Roma ");
    } // fim do método mostraTexto1().
} // fim da classe Mensagem.
```

Figura 1.1: Programa **P1** – Arquivo Mensagem.class.

```
// Declaração da classe de teste da classe Mensagem.
public class TestaMensagem
{
    public static void main( String args[] )
    {
        // Cria um objeto Mensagem e o atribui a minhaMensagem.
        Mensagem minhaMensagem = new Mensagem();

        // Chama método mostraTexto1().
        minhaMensagem.mostraTexto1();

    } // fim do método main.
} // fim de classe TestaMensagem.
```

Figura 1.2: Programa **P1** – Arquivo TestaMensagem.class.

Pede-se:

Item (A) Teste o programa **P1**.

Item (B) Insira um novo método mostrarTexto2() que solicita ao usuário uma string e depois a imprime. Para tanto, utilize os comandos e a sequência da Figura 1.3 e modifique o método main de TestaMensagem para chamar mostrarTexto2().

```
Scanner input = new Scanner(System.in); // Cria Scanner para leitura de dados.
System.out.println("Entre com a sua mensagem");
String novaMensagem = input.nextLine(); // lê uma linha de texto.
System.out.printf("Mensagem personalizada: \n %s ! \n", novaMensagem);
```

Figura 1.3: Comandos a serem empregados no método mostrarTexto2().

Não esqueça de incluir no início da classe TestaMensagem o comando que possibilita o uso do objeto Scanner: **import java.util.Scanner; // utiliza Scanner.**

Item (C) Construir um novo método mostrarTexto3() que imprime na tela uma caixa de diálogo empregando a classe JOptionPane. Para tanto, utilize o código contido na Figura 1.4.

```
// Colocar o comando a seguir no início da classe Mensagem,
// pois permite a importação da classe JOptionPane que permite
// o uso de caixas de diálogo.
import javax.swing.JOptionPane; // importa a classe JOptionPane
// fim main.
public void mostrarTexto3()
{
    // exibe uma caixa de diálogo.
    JOptionPane.showMessageDialog(null, "Mensagem \n em \n caixas ! ");
} // fim do método mostraTexto3().
```

Figura 1.4: Comandos para permitir a impressão de mensagens em caixas.

Item (D) Fornecer um novo método mostrarTexto4() que captura uma mensagem através de uma caixa de diálogo de entrada e depois a mostra em outra caixa de diálogo de saída. Para tanto, use os comandos da Figura 1.5.

```
// Pede para o usuário inserir o nome.
String name = JOptionPane.showInputDialog(" Qual o seu nome ? ");

// Cria a mensagem a ser armazenada na variável texto.
String texto = String.format("Bem-vindo %s ao mundo do Java !", name);

// Exibe a mensagem para cumprimentar o usuário pelo nome.
JOptionPane.showMessageDialog(null, texto);
```

Figura 1.5: Comandos a serem utilizados no método mostrarTexto4().

Exercício 2: O programa **P1** pode ser modificado de forma que a classe Mensagem tenha um campo para armazenar as mensagens fornecidas pelo usuário. Para tanto, é necessário modificar a classe Mensagem com a declaração do campo texto dentro do escopo da classe e ainda declarar dois métodos getMensagem e setMensagem. O primeiro método retorna o conteúdo do campo texto e o segundo possibilita a modificação do campo texto. As modificações descritas são fornecidas pela Figura 2.1. Observe, porém que os demais métodos deve ser alterados para mostrarem na Tela o conteúdo do campo texto.

```
// Declaração da classe Mensagem com apenas um método.
// Esta classe deve estar em um arquivo Mensagem.class.
public class Mensagem
{
    // Declara um campo texto para a classe Mensagem.
    private String texto;

    // Método para configurar a mensagem contida no campo texto.
    public void setTexto( String mens)
    {
        texto = mens;
    } // fim do método setMensagem().

    // Método para recuperar a mensagem contida no campo texto.
    public String getTexto()
    {
        return texto;
    } // fim do método getMensagem().

    // Demais métodos devem ser alterados !!!

} // fim da classe Mensagem.
```

Figura 2.1: Programa **P2** – Arquivo Mensagem.class.

Pede-se:

Item (A): Testar as modificações sugeridas na Figura 2.1. Para tanto, modifique também o programa main utilizando os comandos contidos na Figura 2.2.

```
// Declaração da classe de teste da classe Mensagem.
public class TestaMensagem
{
    public static void main(String args[])
    {
        // Cria Scanner para obter entrada a partir da janela de comando.
        Scanner input = new Scanner( System.in );
        // Cria um objeto Mensagem e o atribui a minhaMensagem.
        Mensagem minhaMensagem = new Mensagem();
        // Chama método que exibe valor inicial de texto e imprime na tela.
```

```
System.out.printf(" Valor inicial de texto: %s \n ", minhaMensagem.getTexto());

// Solicita a inserção de uma mensagem a ser armazenada em texto.
System.out.println("Entre com o texto: ");
String mensa = input.nextLine(); // lê uma linha com caracteres.
minhaMensagem.setTexto(mensa); // Atribui conteúdo de mensa para texto.
System.out.printf(" Valor final de texto: %s \n ", minhaMensagem.getTexto());

} // fim do método main.

} // fim de classe TestaMensagem.
```

Figura 2.2: Programa **P2** – Arquivo TestaMensagem.class.

Item (B): Em função das modificações detalhadas no **Item (A)** é necessário alterar os métodos mostrarTexto1(), mostrarTexto2(), mostrarTexto3() e mostrarTexto4() tal que sempre o conteúdo do campo texto é que seja exibido ou alterado. Por exemplo, para o método mostrarTexto1() será necessário realizar as alterações descritas na Figura 2.3. Use sempre os métodos getTexto() e setTexto().

```
// Exibe uma mensagem.
public void mostrarTexto1()
{
    System.out.printf(" %s \n ", getTexto());
} // fim do método mostraTexto1().
```

Figura 2.3: Programa **P2** – Alterações no arquivo Mensagem.class.

Exercício 3: O programa **P2** pode ainda incluir métodos que inicializam o campo texto automaticamente quando da criação de um objeto do tipo Mensagem. Tais métodos são denominados de construtores e possuem o mesmo nome da classe (no caso Mensagem). Para incluir construtores no programa **P2** utilize os comandos da Figura 3.1.

```
// Construtor vazio: insere uma mensagem padrão no campo texto.
public Mensagem()
{
    texto = " ";
} // fim do método Mensagem().

// Construtor com argumento: inicializa campo texto com String fornecida como
// argumento.
public Mensagem(String mensa)
{
    texto = mensa; // inicializa campo texto.
    // poderia ter usado também: setTexto(mensa);
} // fim do método mostraTexto1().
```

Figura 3.1: Programa **P2** – Inserindo construtores.

Deseja-se verificar as alterações que ocorrem no programa **P2** empregando os comandos da Figura 3.2 a serem inseridos no método main da classe TestaMensagem.

```
// Declaração da classe de teste da classe Mensagem.
public class TestaMensagem
{
    public static void main(String args[])
    {
        // Cria Scanner para obter entrada a partir da janela de comando.
        Scanner input = new Scanner( System.in );

        // Cria um objeto Mensagem e o atribui a minhaMensagem.
        Mensagem minhaMensagem = new Mensagem("Teste Construtor");

        // Chama método que exibe valor inicial de texto e imprime na tela.
        System.out.printf(" Valor inicial de texto: %s \n ", minhaMensagem.getText());

        // Solicita a inserção de uma mensagem a ser armazenada em texto.
        System.out.println("Entre com o texto: ");
        String mensa = input.nextLine(); // lê uma linha com caracteres.
        minhaMensagem.setText(mensa); // Atribui conteúdo de mensa para texto.
        System.out.printf(" Valor final de texto: %s \n ", minhaMensagem.getText());

    } // fim do método main.
} // fim de classe TestaMensagem.
```

Figura 3.2: Programa **P2** – Arquivo TestaMensagem.class que testa construtor.

Exercício 4: Criar uma classe **Conta** que representa a conta corrente de um dado cliente de um Banco. A classe **Conta** deve possuir os seguintes campos:

- nome – Nome do correntista (tipo **String**).
- saldo – valor disponível na conta corrente (tipo **double**).

Este programa ainda deve possuir os seguintes métodos:

Item (A): Para a classe **Conta**, criar um método que fornece valores iniciais "SemNome" e 1000.0 para os campos nome e saldo, respectivamente. Este método deverá ter o mesmo nome da classe e não deve ter nenhum tipo de retorno, nem mesmo **void**, ou seja, deverá ser como na Figura 4.1.

```
public Conta()
{
    nome = "String";
    saldo = 1000.0;
}
```

Figura 4.1: Método que inicializa os valores iniciais de um objeto da classe Conta.

Item (B): Para a classe **Conta**, criar um novo método que fornece valores iniciais nn e ss passados como parâmetros para inicializar os campos nome e saldo, respectivamente. Este método deverá ter o mesmo nome da classe e não deve ter nenhum tipo de retorno, nem mesmo **void**, ou seja, deverá ser como na Figura 4.2.

```
public Conta(String nn, double ss)
{
    nome = nn;
    saldo = ss;
}
```

Figura 4.2: Método que inicializa os valores iniciais de um objeto da classe **Conta**.

Item (C): Criar também um método **public void** mostrarConta() que ao ser invocado imprime o conteúdo dos campos de um objeto da classe **Conta**. Para tanto, empregue os trechos de código da Figura 4.3.

```
public void mostrarConta()
{
    System.out.printf("\n Nome: %s ", nome);
    System.out.printf("\n Saldo: %s ", saldo);
}
```

Figura 4.3: Método que mostra os valores dos campos de um objeto **Conta**.

Item (D): Criar uma classe TestaConta que possui um método main que cria dois objetos c1 e c2 da classe **Conta**. Teste os métodos criados nos itens anteriores. Para tanto, empregue os trechos de código da Figura 4.4.

```
// Declaração da classe de teste da classe Conta.
public class TestaConta
{
    public static void main(String args[])
    {
        // Cria dois objetos Conta e inicializa seus campos com 2 construtores.
        Conta c1 = new Conta();
        Conta c2 = new Conta("Ze Moleza", 5000.0);

        // Chama método que exibe o conteúdo de um objeto da classe conta.
        c1.mostrarConta();
        c2.mostrarConta();
    } // fim do método main.
} // fim de classe TestaMensagem.
```

Figura 4.4: Classe para testar objetos da classe conta.

Item (E): Criar uma método fazerDeposito na classe Conta que incrementa o valor contido no campo saldo de acordo com o valor passado com parâmetro para o método tal como dado na Figura 4.5.

```
public void fazerDeposito(double d)
{
    saldo = saldo + d;
}
```

Figura 4.5: Método que adiciona valor ao valor contido no campo saldo.

Item (F): Criar uma método fazerRetirada na classe Conta que reduz o valor contido no campo saldo de acordo com o valor passado com parâmetro para o método tal como dado na Figura 4.6.

```
public void fazerRetirada(double d)
{
    saldo = saldo - d;
}
```

Figura 4.6: Método que subtrai valor ao valor existente no campo saldo.

Item (G): Modificar o método **main** contido no item (D) de modo a testar os novos métodos da classe conta que foram fornecidos nos itens (E) e (F).

Item (H): Construir um diagrama UML da classe **Conta**. Lembrando que a notação UML é dada por:

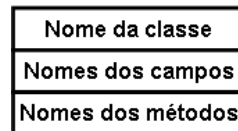


Figura 4.7: Notação UML para representar uma classe.

Exercício 5: Criar os diagramas UML e o código Java correspondente para as classes descritas na Tabela 5.1.

Classe	Campos (tipo)	Valor inicial do campo	Métodos
Lampada	estado (int)	0	•ligar() •desligar() •mostrarEstado()
Carro	velocidade (double)	0.0	•acelerar() •freiar() •mostrarVelocidade()
Time	pontosGanhos (int)	0	•ganharPartida() •empatarPartida() •perderPartida()

Relógio	hora (int)	6	•adicionarHora() •subtrairHora() •mostrarHora()
Telefone	numMinutos (int) preço (double)	0 2.0	•duracaoLigacao(int) •alterarPreço(double) •mostrarConta()

Tabela 5.1: Classes com seus respectivos campos e métodos.

A Tabela 5.2 fornece um melhor detalhamento das ações a serem realizadas para cada método de cada classe.

Classe	Método	Ação
Lampada	ligar()	Campo estado recebe o valor 1.
	desligar()	Campo estado recebe o valor 0.
	mostrarEstado()	Se estado igual a um mostra a mensagem, "Ligada"; senão mostra a mensagem "Desligada".
Carro	acelerar()	Campo velocidade é aumentado em 10.
	freiar()	Campo velocidade é reduzido em 10.
	mostrarVelocidade()	Mostrar o valor atual no campo velocidade.
Time	ganharPartida()	O campo pontosGanhos é aumentado em 3.
	empatarPartida()	O campo pontosGanhos é aumentado em 1.
	perderPartida()	O campo pontosGanhos é aumentado em 0.
Relógio	adicionarHora()	O campo hora é aumentado em 1, desde que não ultrapasse o valor de 23. Se ultrapassar o valor de hora deverá ser ajustado para o valor 0.
	subtrairHora()	Reduz o valor do campo hora em 1, desde que não ultrapasse o valor 0. Se ultrapassar o valor contido no campo hora será ajustado para 23.
	mostrarHora()	Mostra o valor contido no campo hora.
Telefone	duracaoLigacao(int a)	Adiciona o valor a (parâmetro de entrada do método) ao campo numMinutos.
	alterarPreço(double p)	Altera o campo preço para o valor contido no parâmetro p.
	mostrarConta()	Mostra o valor atual da conta telefônica, ou seja, o valor dado por: numMinutos*preço.

Para cada classe deverá ser criada uma classe Testa correspondente que irá realizar a chamada de todos os métodos, permitindo a verificação do correto funcionamento dos mesmos.

Exercício 6: Criar uma classe **Complex** tal que possua os seguintes campos:

- x – parte real do número complexo $z = x + i*y$ (**double**).
- y – parte imaginária do número complexo $z = x + i*y$ (**double**).

Dica: Releia os itens (A) e (B) do Exercício 4 que detalha como construir métodos que inicializam os campos de um objeto da classe. Estes métodos também são denominados de **construtores**.

Item (A): Criar um construtor **Complex** (**int** a, **int** b) que preenche os campos real (x) e imaginário (y) de um objeto do tipo **Complex**.

Item (B): Criar um método **void** mostrar() que mostra os campos de um objeto **Complex** que armazena um número complexo.

Item (C): Criar um método somar(**Complex** z₁) que realiza a soma de dois números complexos (objetos): o objeto do parâmetro de entrada e o objeto que invocou o método. O resultado é armazenado nos campos do objeto que invocou o método.

Item (D): Criar um método subtrair(**Complex** z₁, **Complex** z₂) que realiza a subtração de dois números complexos (objetos) e armazena o resultado no objeto que o invocou.

Item (E): Criar um método multiplicar(**Complex** z₁) que realiza a multiplicação de dois números complexos (objetos) e armazena o resultado no objeto que invocou o método. Para tanto, use a Equação (1):

$$z_3 = z_1 z_2 \leftrightarrow (x_3, y_3) = (x_1 x_2 - y_1 y_2, x_1 y_2 + x_2 y_1) \quad (1)$$

Exercício 7: Criar uma classe **Fracao** tal que possua os seguintes campos:

- num – campo com o valor do numerador da fração (**int**).
- dem – campo com o valor do denominador da fração (**double**).

Item (A): Criar um construtor **Fracao** (**int** a, **int** b) que preenche os campos num e dem de um objeto do tipo **Fracao**.

Item (B): Criar um método **void** mostrar() que mostra os campos de um objeto **Fracao**.

Item (C): Criar um método somar(**Fracao** z₁) que realiza a soma de duas frações (objetos): o objeto do parâmetro de entrada e o objeto que invocou o método. O resultado é armazenado nos campos do objeto que invocou o método.

Item (D): Criar um método subtrair(**Fracao** z₁) que realiza a subtração de dois números complexos (objetos) e armazena o resultado no objeto que o invocou.

Item (E): Criar um método multiplicar(**Fracao** z₁) que realiza a multiplicação de duas frações (objetos) e armazena o resultado no objeto que o invocou.

Exercício 8: Criar uma classe **Circulo** tal que represente as características geométricas de figura e possui os seguintes campos:

- raio – valor do raio.
- circ – valor da circunferência dada em função do raio r por: $C = 2\pi r$.
- area – valor da área dada em função do raio r por: $A = \pi r^2$.

Item (A): Criar um construtor **Circulo** (**int** r) que preenche os campos raio, circ e area de um objeto do tipo **area**.

Item (B): Criar um método **void** mostraI() que mostra os campos de um objeto **Circulo**.

Exercício 9: Criar uma classe **Integral** tal que forneça de forma aproximada o valor de uma integral definida em um intervalo [a, b]:

- a – limite inferior do intervalo de integração.
- b – limite superior do intervalo de integração.
- fa – valor da função no limite inferior do intervalo de integração.
- fb – valor da função no limite superior do intervalo de integração.
- value – valor aproximado da integral no intervalo [a, b]. A integral é aproximada por um trapézio tal como dado na Equação (2):

$$I = \int_a^b f(x) dx \approx \frac{(b-a)}{2} (f(a) + f(b)) \quad (2)$$

Item (A): Criar um construtor **Integral** (**int** a, **int** b, **int** fa, **int** fb) que preenche os campos a, b, fa, fb e value (calculado com a Equação (2)) de um objeto do tipo **Integral**.

Item (B): Criar um método **void** mostraI() que mostra os campos de um objeto **Integral**.

Exercício 1: Construtores são métodos especiais sem tipo de retorno (nem mesmo **void**) e de mesmo nome que a classe que são invocados quando da criação de um objeto (ou seja quando empregamos a palavra-chave **new**). O propósito dos **construtores** é fornecer valores iniciais para os campos de um objeto de classe no momento de sua criação. Além dos construtores, é importante definir em uma classe métodos através dos quais os valores dos campos de um objeto serão alterados. Por convenção usa-se **set** e **get** para alterar e retornar o valor contido em um campo de um dado objeto, respectivamente. A classe usina contida na Figura 1.1 ilustra esses conceitos.

```
public class Usina
{ // Definição dos campos.
  private String nome;
  private double potenciaMax;
  private double geracaoAtual;

  // Definição dos métodos.
  public Usina() // Construtor sem parâmetros
  { nome = "Sem Nome";
    potenciaMax = 0;
    geracaoAtual = 0;
  }

  public Usina(String s, double p, double g) // Construtor com parâmetros
  { nome = s;
    potenciaMax = p;
    geracaoAtual = g;
  }

  // Métodos para modificar e retornar os valores dos campos.
  public void setNome(String s)
  {nome = s;}

  public String getNome()
  {return nome;}

  public void setPotenciaMax(double p)
  {potenciaMax = p;}

  public double getPotenciaMax()
  {return potenciaMax;}

  public void setGeracaoAtual(double g)
  {geracaoAtual = g;}
```

```
public double getGeracaoAtual()
{return geracaoAtual;}

public void mostrarUsina()
{ System.out.printf("\n Nome: %s \n", nome);
  System.out.printf("Potencia Maxima (MW): %8.2f \n", potenciaMax);
  System.out.printf("Geração Atual (MW): %8.2f \n", geracaoAtual);
}

}
```

Figura 1.1: Código da classe Usina.

Para testar o código da classe Usina é necessário construir a classe TestaUsina da Figura 1.2.

```
public class TestaUsina
{ // Definição do método principal para executar o programa.
  public static void main(String args[])
  { // Criando objetos da classe Usina.
    Usina u1 = new Usina(); // Usa o primeiro construtor.
    u1.mostrarUsina();
    Usina u2 = new Usina("Tucuruí", 8340, 5000); // segundo construtor.
    u2.mostrarUsina();
    u1.setNome("Itaipu");
    u1.setPotenciaMax(14000);
    u1.mostrarUsina();
  }
}
```

Figura 1.2: Código da classe TestaUsina.

Exercício 2: O Exercício 1 pode ser aperfeiçoado como 4 modificações:

- (i) Os métodos **set** podem realizar uma validação dos valores passados por parâmetros antes de atribuir estes valores para os campos de um objeto. Por exemplo, o valor máximo de geração de uma usina não pode ser negativo.
- (ii) O construtor com parâmetro pode chamar métodos **set** de modo a inicializar os campos de um objeto de acordo com o processo de validação dos dados contido no método **set** correspondente.
- (iii) O método **mostrarUsina** pode obter os dados dos campos através dos métodos **get** correspondentes.
- (iv) Criar um método **copiarUsina(Usina u)** que copia os dados do campo de um objeto usina u para os campos do objeto que invocou o método.

As 4 modificações do Exercício 1 são ilustradas nas Figuras 2.1 e 2.2.


```

public class Usina
{ // Definição dos campos.
    private String nome;
    private double potenciaMax;
    private double geracaoAtual;

    // Definição dos métodos.
    public Usina() // Construtor sem parâmetros
    {
        setNome("Sem Nome");
        setPotenciaMax(0);
        setGeracaoAtual(0);
    }

    public Usina(String s, double p, double g) // Construtor com parâmetros
    {
        setNome(s);
        setPotenciaMax(p);
        setGeracaoAtual(g);
    }

    // Métodos para modificar e retornar os valores dos campos.
    public void setNome(String s)
    { nome = s; }

    public String getNome()
    { return nome; }

    public void setPotenciaMax(double p)
    { potenciaMax = p < 0.0 ? 0.0 : p; }

    public double getPotenciaMax()
    { return potenciaMax; }

    public void setGeracaoAtual(double g)
    { geracaoAtual = g < 0.0 ? 0.0 : g; }

    public double getGeracaoAtual()
    { return geracaoAtual; }

    public void mostrarUsina()
    {
        System.out.printf("\n Nome: %s \n", getNome());
        System.out.printf("Potencia Maxima (MW): %8.2f \n", getPotenciaMax());
        System.out.printf("Geração Atual (MW): %8.2f \n", getGeracaoAtual());
    }
}

```

```

public void copiarUsina(Usina u)
{
    setNome(u.getNome());
    setPotenciaMax(u.getPotenciaMax());
    setGeracaoAtual(u.getGeracaoAtual());
}
}

```

Figura 2.1: Código da classe Usina.

Para testar o código da classe Usina é necessário construir a classe TestaUsina da Figura 2.2.

```

public class TestaUsina
{ // Definição do método principal para executar o programa.
    public static void main(String args[])
    { // Criando objetos da classe Usina.
        Usina u1 = new Usina(); // Usa o primeiro construtor.
        u1.mostrarUsina();
        Usina u2 = new Usina("Tucuruí", 8340, 5000); // segundo construtor.
        u2.mostrarUsina();
        u1.copiarUsina(u2);
        u1.mostrarUsina();
        u1.setPotenciaMax(14000);
        u1.setNome("Itaipu");
        u1.mostrarUsina();
    }
}

```

Figura 2.2: Código da classe TestaUsina.

Exercício 3: Modificar a classe Usina do Exercício 2 de modo que seja possível controlar o número de objetos da classe usina que foram criados em um dado programa. Para tanto, é necessário empregar um campo de classe que é uma variável comum a todos os objetos criados. Tal tipo de variável é denominada de variável estática e usa-se a palavra reservada **static** para a sua criação. Toda variável declarada sem a palavra **static** é denominada de dinâmica, pois sua existência está associada à existência de um objeto. Para realizar o que é pedido é necessário realizar as modificações descritas nas Figura 3.1 e 3.2.

```

public class Usina
{

    // Definição dos campos.
    private String nome;
    private double potenciaMax;
    private double geracaoAtual;
    private static int numUsinas;
}

```



```

// Definição dos métodos.
public Usina() // Construtor sem parâmetros
{
    setNome("Sem Nome");
    setPotenciaMax(0);
    setGeracaoAtual(0);
    numUsinas++;
}

public Usina(String s, double p, double g) // Construtor com parâmetros
{
    setNome(s);
    setPotenciaMax(p);
    setGeracaoAtual(g);
    numUsinas++;
}

// Métodos para modificar e retornar os valores dos campos.
public void setNome(String s)
{
    nome = s;
}

public String getNome()
{
    return nome;
}

public void setPotenciaMax(double p)
{
    potenciaMax = p < 0.0 ? 0.0 : p;
}

public double getPotenciaMax()
{
    return potenciaMax;
}

public void setGeracaoAtual(double g)
{
    geracaoAtual = g < 0.0 ? 0.0 : g;
}

public double getGeracaoAtual()
{
    return geracaoAtual;
}

public void mostrarUsina()
{
    System.out.printf("\n Nome: %s \n", getNome());
    System.out.printf("Potencia Maxima (MW): %8.2f \n", getPotenciaMax());
    System.out.printf("Geração Atual (MW): %8.2f \n", getGeracaoAtual());
    System.out.printf("Numero total de usinas: %2d \n", numUsinas);
}

public void copiarUsina(Usina u)
{
    setNome(u.getNome());
    setPotenciaMax(u.getPotenciaMax());
    setGeracaoAtual(u.getGeracaoAtual());
}
}

```

Figura 3.1: Código da classe Usina.

```

public class TestaUsina
{
    // Definição do método principal para executar o programa.
    public static void main(String args[])
    {
        // Criando objetos da classe Usina.
        Usina u1 = new Usina(); // Usa o primeiro construtor.
        u1.mostrarUsina();
        Usina u2 = new Usina("Tucuruí", 8340, 5000); // segundo construtor.
        u2.mostrarUsina();
        u1.copiarUsina(u2);
        u1.mostrarUsina();
        u1.setPotenciaMax(14000);
        u1.setNome("Itaipu");
        u1.mostrarUsina();
    }
}

```

Figura 3.2: Código da classe TestaUsina.

Exercício 4: Para empregar um estrutura de seleção na Linguagem Java o comando **if-else** pode ser empregado. Por exemplo, o código da função da Figura 4.1 indica se um aluno foi ou não aprovado em uma matéria.

```

if (notaFinal >= 50)
{
    System.out.printf("%s Aprovado em %s \n", aluno, disciplina);
}
else
{
    System.out.println("%s Reprovado em %s \n", aluno, disciplina);
}

```

Figura 4.1: Exemplo de utilização de estrutura de seleção em Java.

Para testar o uso da estrutura de seleção da Figura 4.1, deve ser construída a classe Aluno, descrita em notação UML, tal como dado na Figura 4.2.

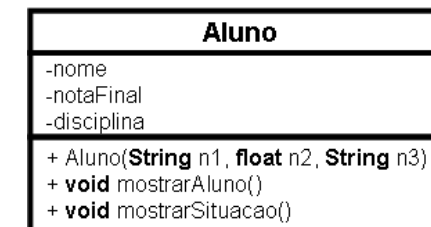


Figura 4.2: Campos e métodos da classe Aluno.

O método **void** mostrarAluno() deve mostrar o conteúdo dos campos nome, notaFinal e disciplina. Já o método **void** mostrarSituacao() deve mostrar, baseado no código da Figura 4.1, a situação de um aluno em uma disciplina. Deve ser criada, ainda, uma classe TestaAluno que lê as informações de um aluno e cria um objeto da classe Aluno e testa os métodos **void** mostrarAluno() e **void** mostrarSituacao(). Para tanto, as informações do aluno podem ser lidas via teclado tal como ilustrado na Figura 4.3 (Para nota use números como 7,8).

```
import java.util.Scanner; // Utiliza Scanner.
Scanner input = new Scanner(System.in); // Cria variável input do tipo Scanner.
System.out.println("Entre com o nome do Aluno: ");
String n1 = input.nextLine(); // Lê uma linha de texto.
System.out.println("Entre com a nota do Aluno: ");
float n2 = input.nextFloat(); // Lê o valor da nota.
input.nextLine(); // Lê o enter após o número inteiro.
System.out.println("Entre com o nome da Materia: ");
String n3 = input.nextLine(); // Lê uma linha de texto.
// Inserir código para a criação de um objeto da classe Aluno,
// bem como código relativo a chamada de seus métodos.
```

Figura 4.3: Entrada de dados para a classe TestaAluno no método main().

Exercício 5: Criar uma classe Ordem cujos campos são variáveis **double**: x, y e z. A classe Ordem deverá possuir os métodos indicados na Tabela 2.1.

Método	Descrição
public Ordem(double a, double b, double c)	Construtor com parâmetros.
public void mostrarMaior()	Indica qual campo possui o maior valor e qual é esse valor.
public void mostrarMenor()	Indica qual campo possui o menor valor e qual é esse valor.
public void mostrarCrescente()	Mostra em ordem crescente os valores contidos em x, y e z.
public void mostrarDecrescente()	Mostra em ordem decrescente os valores contidos em x, y e z.

Tabela 5.1: Métodos da classe Ordem e sua respectiva descrição.

Criar uma classe TestaOrdem que cria um objeto da classe ordem e testa todos os métodos da Tabela 2.1. Na classe Ordem é útil empregar o comando **if-else**, tal como dado no código da função **void** mostrarMaior() dado na Figura 5.1.

```
public void mostrarMaior()
{
    // String que indica qual campo é o maior: x, y e z.
    String nome = "x";
    // Inicialmente x é considerado o maior.
    float maior = x;
    if (maior < y)
    { maior = y;
      nome = "y"; }
    if (maior < z)
    { maior = z;
      nome = "z"; }
    System.out.printf("Maior campo %s: %f ", nome, maior);
}
```

Figura 5.1: Método void mostrarMaior() da classe Ordem que usa **if-else**.

Exercício 6: Criar uma classe Formas tal como especificado no diagrama UML da Figura 6.1.

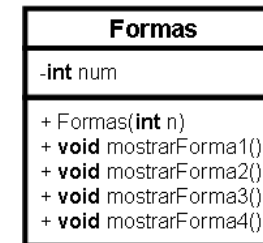


Figura 6.1: Campos e Métodos da classe Formas.

O construtor da classe Formas possui um parâmetro que informa o valor que deve ser inserido no campo num. O valor do campo num será empregado para cada um dos métodos mostrarForma(). Especificamente, cada método deverá apresentar uma Figura diferente tal como descrito na Tabela 6.1 e ilustrado na Figura 6.2.

Método	Figura a ser Apresentada
void mostrarForma1()	Figura 6.2 (A)
void mostrarForma2()	Figura 6.2 (B)
void mostrarForma3()	Figura 6.2 (C)
void mostrarForma4()	Figura 6.2 (D)

Tabela 6.1: Correspondência entre os métodos e as figuras que eles devem apresentar.

<pre> 1 1 2 1 2 3 ... 1 2 3 4 ... n n ... 4 3 2 1 ... 3 2 1 2 1 1 </pre> <p>(A)</p>	<pre> 1 2 1 3 2 1 ... n ... 4 3 2 1 1 2 ... n-2 n-1 n ... n-2 n-1 n n-1 n n </pre> <p>(B)</p>

1 2 3 4 ... n	n ... 4 3 2 1
1 2 3	3 2 1
1 2	2 1
1	1
1	1
2 1	1 2
3 2 1	1 2 3
N ... 4 3 2 1	1 2 3 4 ... n
(C)	(D)

Figura 6.2: Figuras a serem exibidas por cada um dos métodos da classe Formas.

Criar uma classe TestaFormas que cria um objeto da classe Formas e testa os métodos descritos na Tabela 6.1. Além disso, TestaFormas deverá solicitar ao usuário um valor n (Teste valores de n = 4 e 5).

Exercício 7: Criar uma classe CalculoPi tal como especificado no diagrama UML da Figura 7.1.

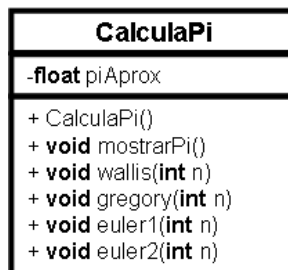


Figura 7.1: Diagrama UML da classe CalculoPi.

O construtor CalculoPi() deverá inicializar o campo pi com o valor zero. Já o método **void** mostrarPi() mostra o conteúdo do campo pi. Os demais métodos possuem como parâmetro de entrada um valor inteiro que corresponde ao número de termos a serem empregados para calcular o aproximado valor de pi. Cada método utiliza uma fórmula tal como indicado na Tabela 7.1.

Método	Aproximação a ser utilizada para calcular o valor de pi
wallis	$\frac{\pi}{2} = \frac{2.2}{1.3} \cdot \frac{4.4}{3.5} \cdot \frac{6.6}{5.7} \cdot \frac{8.8}{7.9} \dots$
gregory	$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \dots$
euler1	$\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2} + \dots$
euler2	$\frac{\pi^4}{90} = \frac{1}{1^4} + \frac{1}{2^4} + \frac{1}{3^4} + \frac{1}{4^4} + \frac{1}{5^4} + \dots$

Tabela 7.2: Fórmulas a serem empregadas para o cálculo de Pi em cada método.

Um exemplo de código para o método **void** wallis(int n), e que ilustra a utilização do comando **for**, é dado na Figura 7.3.

```

public void wallis(int n)
{
    piAprox = 1;
    for (int i=1; i <= n; i++)
    {
        // Usar conversão explícita para converter a variável i em float
        // e poder usar a mesma no cálculo de um valor float.
        piAprox = piAprox * (float)((2*i)*(2*i))/(float)((2*i-1)*(2*i+1));
    }
    piAprox = 2*piAprox;
}
  
```

Figura 7.3: Código do Método **void** wallis(int n) empregando o comando **for**.

Exercício 8: Criar uma classe Materia tal como especificado no diagrama UML da Figura 8.1.

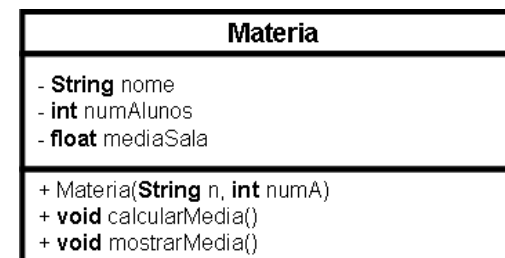


Figura 8.1: Campos e métodos da classe Materia.

Pede-se

Item (A): O método construtor da classe Materia possui dois parâmetros tal que o primeiro é preenche o campo nome e o segundo preenche o campo num. O método **void** calcularMedia() emprega um comando **for** que lê numA valores de notas dos alunos. Terminada a leitura dos dados, é calculada a média dos alunos na matéria e este valor é armazenado no campo mediaSala. Posteriormente, o método **void** mostrarMedia() deve ser empregado para mostrar o valor da média. Teste o funcionamento da classe Materia através de uma classe TestaMateria que contém um objeto da classe Materia.

Item (B): Modifique o método **void** calcularMedia() para que os dados das notas dos alunos sejam valores reais gerados no intervalo [0,10]. Para tanto, empregue o código descrito na Figura 8.2.

```

import java.util.Random; // Permite o uso de objetos da Classe Random.
  
```

```

int nota;
// Cria um objeto da classe Random que funciona como gerador aleatório.
Random randomNumbers = new Random();

// Laço para gerar tantos números aleatórios quanto for o número de alunos.
for (int i=1; i <= numAlunos; i++)
{
    nota = randomNumbers.nextDouble(); // Gera número real em [0, 1].
    nota = 10*nota; // As notas devem variar no intervalo [0, 10].
    // Acrescentar mais código relativo ao cálculo da média...
}

```

Figura 8.2: Geração de números aleatórios empregando um objeto Random.

Item (C): Modifique o método **void** calcularMedia() para que sejam contabilizadas e impressas na tela o número de notas em cada uma das 5 categorias descritas na Tabela 8.1. Para tanto, será necessário empregar o comando **if-else**.

Intervalo de nota	Categoria
10,0-8,0	A
8,0-6,0	B
6,0-5,0	C
5,0-3,0	D
3,0-0,0	E

Tabela 8.1: Possíveis categorias associadas aos valores das notas dos alunos.

Exercício 9: Construir uma classe Adivinho cujo diagrama UML é dado na Figura 9.1.

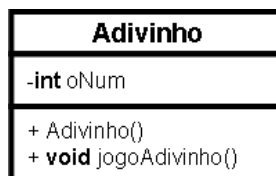


Figura 9.1: Campos e métodos da classe Adivinho.

O construtor desta classe gera um número aleatório inteiro no intervalo [0, 100] e armazena no campo oNum. Para tanto, empregar randomNumbers.nextInt() ao invés de randomNumbers.nextDouble() (vide Figura 8.2). Além disso, o método **void** jogoAdivinho() deve realizar perguntas ao usuário até que este adivinhe qual foi o número digitado. Se o número fornecido pelo usuário for maior que oNum, a mensagem "Número maior que o gerado" deverá ser fornecida. Se o número for menor que oNum, a mensagem "Número menor que o gerado". Depois da mensagem deverá ser pedido outro número para o usuário. Se o usuário acertar, então, a mensagem "O número é

bom !" deverá ser fornecida e o método termina. Criar uma classe TestaAdivinho que emprega um objeto da classe Adivinho para testar o método void jogoAdivinho().

Exercício 10: Criar uma classe BouncingBall tal como especificado no diagrama UML da Figura 10.1.

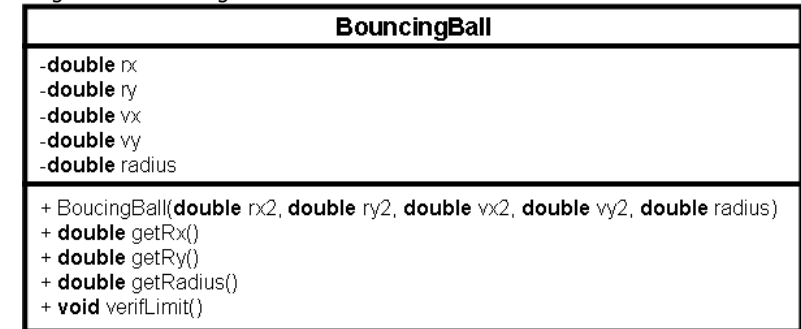


Figura 10.1: Campos e métodos da classe BouncingBall.

Uma descrição detalhada dos campos e dos métodos está nas Tabelas 10.1 e 10.2.

Campo	Descrição
rx	Posição no eixo x do objeto BouncingBall.
ry	Posição no eixo y do objeto BouncingBall.
vx	Velocidade no eixo x do objeto BouncingBall.
vy	Velocidade no eixo y do objeto BouncingBall.
radius	Raio do objeto BouncingBall.

Tabela 10.1: Descrição dos campos da classe BouncingBall.

Método	Descrição
BouncingBall	Construtor que inicializa os campos do objeto.
getRx	Retorna o valor do campo rx.
getRy	Retorna o valor do campo ry.
getRy	Retorna o valor do campo ry.
verifLimit	Verifica se o próximo passo do objeto BouncingBall está dentro do limite da área de desenho da figura. Se não estiver, então, inverte a velocidade e depois calcula uma nova posição.

Tabela 10.2: Descrição dos métodos da classe BouncingBall.

Uma sugestão de código para o método **void** verifLimit() é dado na Figura 10.2.

```

// Verifica se a próxima posição de uma bola está dentro da área
// de desenho. Se não estiver, corrige a velocidade de

```

```
// modo a permanecer dentro da área de desenho.
if (Math.abs(rx + vx) + radius > 1.0) vx = -vx;
if (Math.abs(ry + vy) + radius > 1.0) vy = -vy;
// Atualizando a posição de modo a permanecer na área de desenho.
rx = rx + vx;
ry = ry + vy;
```

Figura 10.2: Código para o método verifLimit.

Além disso, a classe deverá ter um método main tal como dado na Figura 10.3

```
public static void main(String args[])
{
    // Simula o movimento de uma bouncing ball.
    StdDraw.setXscale(-1.0, 1.0);
    StdDraw.setYscale(-1.0, 1.0);
    BouncingBall b1 = new BouncingBall(.48,.86,.015,.023,.05);
    while (true)
    { b1.verifLimit();
      StdDraw.setPenColor(StdDraw.RED);
      StdDraw.filledCircle(b1.getRx(), b1.getRy(), b1.getRadius());
      b1.verifLimit();
      StdDraw.show(20);
      StdDraw.clear();
    }
}
```

Figura 10.3: Método main para testar a classe BouncingBall.

Com as informações e código acima deseja-se criar uma simulação com 3 bolas cujos parâmetros são dados na Tabela 10.3.

Bola	rx	ry	vx	vy	radius
1	.48	.86	.015	.023	.05
2	.28	.16	.005	.013	.01
3	.78	.46	.035	.063	.03

Tabela 10.3: Dados dos objetos BouncingBall a serem testados na simulação.

Por último, será necessário adicionar a classe StdDraw que encontra-se disponível no site:

<http://www.cs.princeton.edu/introcs/35purple/StdDraw.java.html>

Informações detalhadas sobre esta classe em:

<http://www.cs.princeton.edu/introcs/35purple/>

Exercício 11: Criar uma classe NaveEspacial tal como especificado no diagrama UML da Figura 11.1.

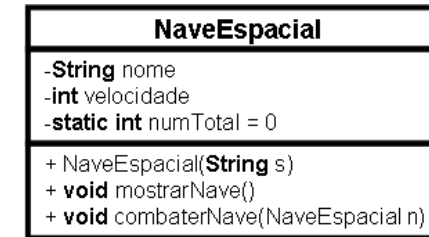


Figura 11.1: Campos e métodos da classe NaveEspacial.

Item (A): Elaborar o construtor da classe que deverá empregar o parâmetro n como valor para o campo nome de um objeto NaveEspacial. Além disso, o campo velocidade de ter um valor aleatório inteiro contido no intervalo [5, 10]. Cada vez que um objeto NaveEspacial for criado deverá ser incrementado o valor de numTotal. Para testar o construtor, criar uma classe TestaNaveEspacial que cria 4 objetos NaveEspacial. Após a criação de cada objeto NaveEspacial, o método **void** mostrarNave() deverá ser chamado e deverá mostrar os valores dos campos nome, velocidade e numTotal. Se mais de 3 objetos do tipo NaveEspacial forem criados, então, a mensagem "Superpopulação de Naves" deverá ser exibida e os campos nome e velocidade do novo objeto deverão ter "Nave com Defeito" e 0, respectivamente. Quando o método **void** mostrarNave() encontrar um objeto Nave cujo campo velocidade é igual a zero, então, deverá mostrar dois valores: número total de naves criadas e o número de naves operantes (campo velocidade é diferente de 0).

Item (B): Implementar o método **void** combaterNave(NaveEspacial n) que verifica qual entre dois objetos NaveEspacial possui maior valor no campo velocidade. A que possuir maior valor é dita vencedora. Imprimir o nome da nave vencedora.

Exercício 1: Construir uma Classe **Vetor** cujo diagrama UML é dado na Figura 1.1.

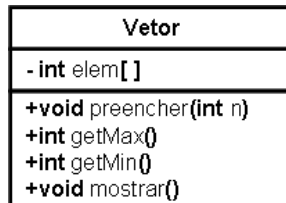


Figura 1.1: Campos e métodos da classe Vetor.

A classe vetor possui um campo elem que é um vetor de inteiros. Este campo é preenchido quando o método preencher é chamado. Portanto, os construtores com e sem parâmetros da classe Vetor chamam este método. Uma melhor descrição dos métodos e como eles se relacionam é fornecida na Tabela 1.1.

Método	Descrição
Vetor()	Construtor sem parâmetros. Deve realizar a leitura do valor correspondente ao número n de elementos do objeto da classe Vetor e só depois chamar o método preencher().
Vetor(int n)	Método que recebe um parâmetro inteiro n com o número de elementos de um objeto da classe Vetor . Deve chamar o método preencher() para inicializar o campo elem .
preencher(int n)	Método que recebe um parâmetro inteiro que define o número de elementos a serem armazenados no campo elem .
int getMax()	Retorna o maior elemento contido no campo elem .
int getMin()	Retorna o menor elemento contido no campo elem .
mostrar()	Método que exibe todos os elementos contidos no campo elem . Para tanto, usa a informação contida no campo elem.length (contendo o número de elementos do vetor) que todo vetor na linguagem Java automaticamente possui ao ser criado.

Tabela 1.1: Métodos da classe **Vetor** e sua descrição.

Além disso, será necessário construir uma classe **TestaVetor** que usa dois objetos da classe **Vetor** para verificar o funcionamento dos métodos da classe **Vetor**. A descrição das classes **Vetor** e **TestaVetor** está contida nas Figuras 1.2 e 1.3, respectivamente.

```

import java.util.Scanner;

public class Vetor
{
    // Campo da classe e um vetor de inteiros.
    int elem[]; // nao foi definido o seu tamanho ainda.

    // Definicao dos construtores.
    public Vetor() // Sem parametros.
    {
        // Criando um objeto Scanner para leitura dos dados.
        Scanner input = new Scanner(System.in);
        // Realizando a leitura via teclado no numero de
        // elementos a serem armazenados em elem.
        System.out.println("Digite o numero de elementos: ");
        int n = input.nextInt();
        // Chamada ao metodo preencher.
        preencher(n);
    }

    public Vetor(int n) // Com parametros.
    {
        // Nao precisa ler n e ja chama preencher().
        preencher(n);
    }

    // Metodo para preencher o campo elem com valores
    // inteiros empregando um objeto Scanner.
    public void preencher(int n)
    {
        // Importante: Antes de preencher o vetor, devemos garantir que foram
        // criados n espacos na memoria do tipo int e que eles estao associados
        // ao campo elem[].
        elem = new int[n];

        Scanner input = new Scanner(System.in);
        for (int i=0; i < elem.length; i++)
        {
            System.out.printf("\n Forneça o %d - esimo valor: ",i+1);
            elem[i] = input.nextInt();
        }
    }
}
  
```

```

// Metodo para achar o maior valor contido em elem.
public int getMax()
{
    // Declarando variavel cujo valor sera retornado.
    int aux = 0;
    // Verifica se o vetor tem elementos !
    if (elem.length > 0)
    {
        // Inicializa o valor maximo com o primeiro do vetor.
        aux = elem[0];
        // Laco para achar maior valor.
        for (int i=1; i < elem.length; i++)
            if (aux < elem[i])
                aux = elem[i];
    }
    // O vetor nao tem elemento e retorna uma
    // mensagem e um valor padrao: zero.
    else
    {
        System.out.println("\n Vetor vazio ! \n");
        aux = 0;
    }
    // Retorna o valor contido em aux.
    return aux;
}

// Metodo para achar o menor valor contido em elem.
public int getMin()
{
    // Declarando variavel cujo valor sera retornado.
    int aux = 0;
    // Verifica se o vetor tem elementos !
    if (elem.length > 0)
    {
        // Inicializa o valor maximo com o primeiro do vetor.
        aux = elem[0];
        // Laco para achar menor valor.
        for (int i=1; i < elem.length; i++)
            if (aux > elem[i])
                aux = elem[i];
    }
    // O vetor nao tem elemento e retorna uma mensagem e valor padrao:0
    else
    {
        System.out.println("\n Vetor vazio ! \n");
        aux = 0;
    }
    // Retorna o valor contido em aux.
    return aux;
}

```

```

// Metodo para mostrar na tela os elementos armazenados no campo elem.
public void mostrar()
{
    // Metodo para mostrar os valores contidos em elem.
    for (int i=0; i < elem.length; i++)
    {
        System.out.printf("\n v[%d] = %d ",i,elem[i]);
    }
}

} // Fim da classe.

```

Figura 1.2: Código da classe **Vetor** contendo seu campo e seus métodos.

```

public class TestaVetor
{
    public static void main(String args[])
    {
        // Criando um primeiro objeto vetor !
        Vetor v1 = new Vetor();

        // Obtendo estatísticas acerca dos valores contidos em um vetor.
        System.out.printf("\n Maior valor = %d ",v1.getMax());
        System.out.printf("\n Menor valor = %d ",v1.getMin());

        // Imprimindo os elementos de um vetor !
        v1.mostrar();

        // Criando um novo vetor com 4 elementos !
        Vetor v2 = new Vetor(4);

        // Obtendo estatísticas e imprimindo os elementos.
        System.out.println(" Maior valor = "+v2.getMax());
        System.out.println(" Menor valor = "+v2.getMin());
        v2.mostrar();
    }
}

```

Figura 1.3: Código da classe **TestaVetor**.

Algumas sugestões de testes numéricos são fornecidas na Tabela 1.2.

Teste	V1	V2
1	[5 -10 1]	[9 8 7 6]
2	[-10 -20 10]	[5 7 9 11]

Tabela 1.2: Sugestões de valores para testar o programa.

Exercício 2: Refazer o Exercício 1 incorporando dois novos métodos: setElem e getElem. Os dois métodos estão descritos na Tabela 2.1.

Método	Descrição
setElem(int i, int a)	Método que modifica o elemento elem[i] para o valor a, desde que o índice i não exceda os limites do vetor, ou seja, não seja negativo nem maior ou igual que elem.length .
int getElem(int i)	Método que retorna o conteúdo de elem[i] desde que o índice i não exceda os limites do vetor, ou seja, não seja negativo nem maior ou igual que elem.length .

Tabela 2.1: Novos métodos da classe **Vetor** e sua descrição.

Modifique todos os métodos da classe Vetor de modo a empregar os novos métodos **set** e **get**, bem como modifique a classe **TestaVetor** de modo a testar os novos métodos criados.

```
import java.util.Scanner;

public class Vetor
{
    // Campo da classe e um vetor de inteiros.
    int elem[]; // nao foi definido o seu tamanho ainda.

    // Definicao dos construtores.
    public Vetor() // Sem parametros.
    {
        // Criando um objeto Scanner para leitura dos dados.
        Scanner input = new Scanner(System.in);
        // Realizando a leitura via teclado no numero de
        // elementos a serem armazenados em elem.
        System.out.println("Digite o numero de elementos: ");
        int n = input.nextInt();
        // Chamada ao metodo preencher.
        preencher(n);
    }

    public Vetor(int n) // Com parametros.
    {
        // Nao precisa ler n e ja chama preencher().
        preencher(n);
    }

    // Método para modificar o elemento contido em elem[i],
```

```
// desde que o índice i esteja dentro de um intervalo
// válido, ou seja, entre o zero e menor que elem.length.
public void setElem(int i, int a)
{
    // Verificando se o indice esta dentro dos limites.
    if ((i >= 0)&&( i < elem.length))
        elem[i] = a;
    else
        System.out.println("Índice fora dos limites do vetor !");
}

// Método para retornar o elemento contido em elem[i],
// desde que o índice i esteja dentro de um intervalo
// válido, ou seja, entre o zero e menor que elem.length.
public int getElem(int i)
{
    // Verificando se o indice esta dentro dos limites.
    if ((i >= 0)&&( i < elem.length))
        return elem[i];
    else
    {
        System.out.println("Índice fora dos limites do vetor !");
        return 0;
    }
}

// Metodo para preencher o campo elem com valores
// inteiros empregando um objeto Scanner.
public void preencher(int n)
{
    // Importante: Antes de preencher o vetor, devemos garantir que foram
    // criados n espacos na memoria do tipo int e que eles estao associados
    // ao campo elem[].
    elem = new int[n];

    Scanner input = new Scanner(System.in);
    for (int i=0; i < elem.length; i++)
    {
        System.out.printf("\n Forneça o %d - esimo valor: ",i+1);
        setElem(i, input.nextInt());
    }
}

// Metodo para achar o maior valor contido em elem.
```

```

public int getMax()
{
    // Declarando variavel cujo valor sera retornado.
    int aux = 0;
    // Verifica se o vetor tem elementos !
    if (elem.length > 0)
    {
        // Inicializa o valor maximo com o primeiro do vetor.
        aux = getElem(0);
        // Laco para achar maior valor.
        for (int i=1; i < elem.length; i++)
            if (aux < getElem(i))
                aux = getElem(i);
    }
    // O vetor nao tem elemento e retorna uma
    // mensagem e um valor padrao: zero.
    else
    {
        System.out.println("\n Vetor vazio ! \n");
        aux = 0;
    }
    // Retorna o valor contido em aux.
    return aux;
}

// Metodo para achar o menor valor contido em elem.
public int getMin()
{
    // Declarando variavel cujo valor sera retornado.
    int aux = 0;
    // Verifica se o vetor tem elementos !
    if (elem.length > 0)
    {
        // Inicializa o valor maximo com o primeiro do vetor.
        aux = getElem(0);
        // Laco para achar menor valor.
        for (int i=1; i < elem.length; i++)
            if (aux > getElem(i))
                aux = getElem(i);
    }
    // O vetor nao tem elemento e retorna uma mensagem e valor padrao:0
    else
    {
        System.out.println("\n Vetor vazio ! \n");
        aux = 0;
    }
    // Retorna o valor contido em aux.
    return aux;
}

// Metodo para mostrar na tela os elementos armazenados no campo elem.

```

```

public void mostrar()
{
    // Metodo para mostrar os valores contidos em elem.
    for (int i=0; i < elem.length; i++)
    {
        System.out.printf("\n v[%d] = %d ",i,getElem(i));
    }
}

} // Fim da classe.

```

Figura 1.2: Código da classe **Vetor** contendo seu campo e seus métodos.

```

public class TestaVetor
{
    public static void main(String args[])
    {
        // Criando um primeiro objeto vetor !
        Vetor v1 = new Vetor();
        // Obtendo estatísticas acerca dos valores contidos em um vetor.
        System.out.printf("\n Maior valor = %d ",v1.getMax());
        System.out.printf("\n Menor valor = %d ",v1.getMin());
        // Imprimindo os elementos de um vetor !
        v1.mostrar();

        // Criando um novo vetor com 4 elementos !
        Vetor v2 = new Vetor(4);
        v2.setElem(0,-1000);
        v2.setElem(-1,-1000);
        v2.setElem(10,89);
        // Obtendo estatísticas e imprimindo os elementos.
        System.out.println(" Maior valor = "+v2.getMax());
        System.out.println(" Menor valor = "+v2.getMin());
        v2.mostrar();
    }
}

```

Figura 1.3: Código da classe **TestaVetor**.

Exercício 3: Construir uma Classe **Sequencia** cujos campos e métodos são dados na Figura 3.1.

Sequencia
- double elem[]
+ Sequencia(int nTermos) + boolean verifLength(int num) + void Fibonacci(int num) + void mostrarTermos(int num)

Figura 3.1: Campos e métodos da Classe **Sequencia**.

Para testar a classe **Sequencia** empregar uma classe **TestaSequencia** apropriada. Dada a classe da Figura 3.1, pede-se:

Item (A): Implementar os métodos de acordo com a Tabela 3.1.

Método	Descrição
Sequencia	Construtor com um parâmetro inteiro nTermos. Este parâmetro irá definir o tamanho do vetor de elementos do campo elem. Se nTermos for menor ou igual a 2, deverá aparecer uma mensagem dizendo qual o valor de nTermos e que este será mudado para 3. Após isso, será realizada alocação dinâmica do vetor elem para o número contido em nTermos. Por fim, o vetor elem será preenchido com valores iguais a zero.
verifLength	Método que recebe um parâmetro inteiro com o número de inteiros num e verifica se num é maior que elem.length ou ainda se num é menor ou igual a zero. Se um desses dois casos ocorrer, então, o valor false deverá ser retornado. Caso contrário, retorna true .
Fibonacci	Método que recebe um parâmetro inteiro que define o número de termos da sequência de Fibonacci a serem calculados e armazenados no vetor contido no campo elem, tal que: $\begin{cases} elem[0] = elem[1] = 1 \\ elem[i] = elem[i-1] + elem[i-2], i = 2, \dots, n \end{cases}$ É importante que antes de realizar os cálculos o método verifLength seja chamada para verificar se o valor num passado por parâmetro é adequado. Se não for, então, alterar para o valor padrão elem.length .
mostrarTermos	Método que recebe um parâmetro inteiro num que define o número de elementos, contidos no vetor do campo elem, a serem mostrados. É importante que antes de realizar os cálculos o método verifLength seja chamada para verificar se o valor num passado por parâmetro é adequado. Se não for, então, alterar para o valor padrão elem.length .

Tabela 3.1: Métodos da classe **Sequencia** e sua descrição.

Item (B): Implementar o método PA que têm como parâmetros três valores inteiros n, q e a_1 . Este método atribui ao vetor contido no campo elem n valores de uma progressão aritmética cuja fórmula é dada por: $a_i = a_0 + (i-1)q$.

Item (C): Implementar o método PG que têm como parâmetros três valores inteiros n, q e a_1 . Este método atribui ao vetor contido no campo elem n valores de uma progressão geométrica cuja fórmula é dada por: $a_i = a_0 q^{i-1}$.

Item (D): Implementar o método somarPA que têm como parâmetros três valores inteiros n, a_1 e a_n . Este método atribui ao vetor contido no campo elem n valores de uma soma de uma progressão aritmética cuja fórmula é dada por: $S_i = i(a_0 + a_i) / 2$.

Item (E): Implementar o método somarPG que têm como parâmetros três valores inteiros n, q e a_1 . Este método atribui ao vetor contido no campo elem n valores de uma soma de uma progressão geométrica cuja fórmula é dada por:

$$S_i = \frac{a_0(q^{i-1} - 1)}{q - 1}.$$

Item (F): Implementar o método fatorial que tem como parâmetro um valor inteiro n. Este método atribui ao vetor contido no campo elem n valores da função fatorial. Por exemplo, o elemento elem[0] deve conter o fatorial de 0, o elemento elem[1] o fatorial de 1 e assim por diante. Para calcular o valor do fatorial de cada número é útil empregar a seguinte definição:

$$n! = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{c.c.} \end{cases}$$

Item (G): Implementar o método calcularPi que tem como parâmetro um valor inteiro n. Construa um programa que coloque no vetor elem os n primeiros valores das sucessivas aproximações do valor de π utilizando a seguinte fórmula:

$$\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots \right)$$

Exercício 4: Construir uma classe **VetorInt** cujos campos e métodos são dados na Figura 4.1.

VetorInt	
- int elem[]	
+ VetorInt(int ntam) + void mostrarVetor() + int prodIntVetor(VetorInt v1) + void multiplicarVetor(VetorInt v1) + void gerarBinario() + int distanciaHamming1(VetorInt v1) + void simularDado() + void mostrarHistograma(int numSorteios)	

Figura 4.1: Campos e métodos da Classe **VetorInt**.

Dada a classe da Figura 4.1, pede-se:

Item (A): Implementar os métodos tal como dado na Tabela 4.1.

Método	Descrição
VetorInt	Construtor com um parâmetro inteiro nTam. Este parâmetro irá definir o tamanho do vetor de elementos do campo elem. Se nTam for menor ou igual a 1, deverá aparecer uma mensagem dizendo qual o valor de nTermos e que este será mudado para 2. Após isso, será realizada alocação dinâmica do vetor elem para o número contido em nTam. Por fim, o vetor elem será preenchido com valores iguais a zero.
mostrarVetor	Método que mostra os elementos do vetor elem empregando um laço e o comando elem.length.

Tabela 4.1: Métodos da classe **VetorInt** e sua descrição.

Item (B): Implementar o método **void** multiplicarVetor(**VetorInt** v1) que realiza a multiplicação entre dois vetores v1 e v2 e armazena o resultado em v2. Por exemplo, a chamada v2.multiplicarVetor(v1) realiza a multiplicação de v2 com v1 e armazena o resultado em v2. Deve ser observado que a multiplicação elemento a elemento é como dado na Figura 4.2.

1	2	3	4	5	6	7	8
v1[0]	v1[1]	v1[2]	v1[3]	v1[4]	v1[5]	v1[6]	v1[7]
*	*	*	*	*	*	*	*
2	2	2	2	2	2	2	2
v2[0]	v2[1]	v2[2]	v2[3]	v2[4]	v2[5]	v2[6]	v2[7]
=	=	=	=	=	=	=	=
2	4	6	8	10	12	14	16
v2[0]	v2[1]	v2[2]	v2[3]	v2[4]	v2[5]	v2[6]	v2[7]

Figura 4.2: Operação a ser realizada por **void** multiplicarVetor(**VetorInt** v1).

Item (C): Implementar o método **int** prodIntVetor(**VetorInt** v1) que retorna o resultado de um produto interno entre dois vetores. Lembrar que o resultado k do produto interno de dois vetores v e t de dimensões $1 \times n$ é calculado através da seguinte fórmula: $k = \langle v, t \rangle = \sum_{i=1}^n v_i t_i$.

Item (D): Implementar o método **void** gerarBinario() que faz com que um vetor com n posições tenha valores ou zero ou 1 gerados aleatoriamente. Para tanto, será necessário empregar o código descrito na Figura 4.3.

```
import java.util.Random; // Permite o uso de objetos da Classe Random.

// Cria um objeto da classe Random que funciona como gerador aleatório.
Random randomNumbers = new Random();

// Laço para gerar n números aleatórios.
for (int i=1; i <= elem.length; i++)
{
    elem[i] = randomNumbers.nextInt(2); // Gera número inteiro 0 ou 1.
}
```

Figura 4.3: Geração de números aleatórios empregando um objeto Random.

Item (E): Implementar o método **int** distanciaHamming1(**VetorInt** v) que retorna a distância de **Hamming** entre dois vetores v e t gerados aleatoriamente (através do método **void** gerarBinario(), por exemplo). Por exemplo, a chamada t.distanciaHamming1(v) calcula a distância de **Hamming** entre v e t . Por exemplo, sejam os vetores v e t de tamanho 10, então, a distância de **Hamming** é obtida como dado na Figura 4.4.

Valor	1	0	1	0	0	0	0	0	0
Índice	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]
Comparação	↔	↔	↔	↔	↔	↔	↔	↔	↔
Valor	1	1	0	0	1	0	1	0	0
Índice	t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]
Diferença	0	1	1	0	1	0	1	0	0

Figura 4.4: Cálculo da **distância de Hamming** para v e t com tamanho 10.

Para os valores dos vetores binários v e t apresentados anteriormente a **distância de Hamming** é igual a quatro, pois os elementos $v[1]$, $v[2]$, $v[4]$ e $v[6]$ são diferentes (ou seja, quatro componentes dos vetores diferem entre si).

Item (F): Implementar o método **void** simularDado() que faz com que um vetor com n posições tenha valores inteiros no intervalo $[1,6]$ gerados aleatoriamente. Para tanto, será necessário empregar o código descrito na Figura 4.5.

```
import java.util.Random; // Permite o uso de objetos da Classe Random.

// Cria um objeto da classe Random que funciona como gerador aleatório.
Random randomNumbers = new Random();

// Laço para gerar n números aleatórios.
for (int i=1; i <= elem.length; i++)
{
    elem[i] = 1 + randomNumbers.nextInt(6); // Gera número inteiro em [1,6].
}
```

Figura 4.5: Geração de números aleatórios empregando um objeto Random.

Item (G): Implementar o método **void** mostrarHistograma(int numSorteios) que apresenta sob a forma de um histograma os valores contidos dentro de um vetor cujos elementos tenham sido gerados a partir do método **void** simularDado(). Para tanto, você deverá ter um vetor auxiliar que irá contabilizar o número de vezes que uma face foi sorteada. A partir dos valores contidos neste vetor auxiliar será possível empregar o código da Figura 4.6 para gerar uma figura tal como dado na Figura 4.7 (para usar o comando StdDraw.line você precisará do arquivo StdDraw.java disponível em www.feg.unesp.br/~anibal).

```
public void mostrarHistograma(int numSorteios)
{
    int aux[]={0,0,0,0,0,0};
    for (int i=0; i < elem.length; i++)
    { aux[(int)elem[i]]++; }
    StdDraw.setXscale(-1, 6);
    StdDraw.setYscale(0, numSorteios);
    StdDraw.setPenRadius(.5/6);
    for (int k=1; k <= 6; k++)
    { StdDraw.line(k, 0, k, aux[k]);
      // System.out.printf("Face %d:[%d]",k,aux[k]);
    }
}
```

Figura 4.6: Código para o método **void** mostrarHistograma(int numSorteios).

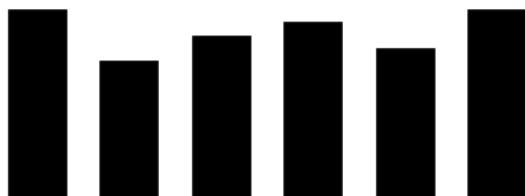


Figura 4.7: Histograma gerado empregando a classe StdDraw.

Exercício 5: Use um vetor para resolver o seguinte problema. Uma empresa paga seus vendedores com base em comissões. O vendedor recebe um valor fixo de R\$ 500,00 por mês mais 10 por cento de suas vendas brutas daquele mês. Por exemplo, um vendedor que teve vendas brutas de R\$ 3000,00 em um mês recebe R\$500,00 mais 10 por cento de R\$ 3000,00, ou seja, um total de R\$ 800,00. Escreva um programa (usando um vetor de contadores) que determine quantos vendedores (crie um aleatoriamente um vetor de vendedores de tamanho 10) receberam salários nos seguintes intervalos de valores (considere que o salário de cada vendedor é truncado para que seja obtido um valor inteiro) como dado na Tabela 5.1.

Faixa 1	Faixa 2	Faixa 3	Faixa 4	Faixa 5
500-999	1000-1499	1500-1999	2000-2999	3000-em diante

Tabela 5.1: Possíveis faixas de salário para cada vendedor.

Para tanto, será necessário criar uma classe Vendedor cujo diagrama UML é dado na Figura 5.1.

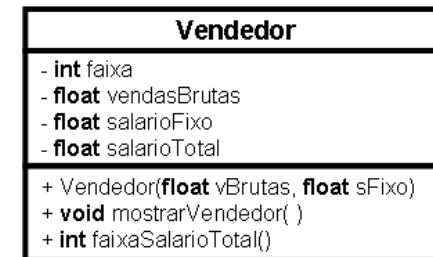


Figura 5.1: Campos e métodos da Classe **Vendedor**.

Observar que o construtor da classe **Vendedor** possui dois parâmetros que inicializam os campos vendasBrutas e salarioFixo, respectivamente. Além disso, com base nos valores contidos nestas variáveis o campo faixa (observe a Tabela 5.1) e salarioTotal também são conhecidos ao final do construtor. Para saber em qual faixa o salário de um dado vendedor está, basta empregar o método **int** faixaSalarioTotal().

Exercício 6: Refazer o **Exercício 5**, mas considerando que para cada Faixa de valor de vendas existe um percentual de comissão como dado na Tabela 6.1.

	Faixa 1 (500-1999)	Faixa 2 (2000-2999)	Faixa 3 (3000-em diante)
Comissão	10%	12%	14%

Figura 6.1: Comissão para cada faixa de valor de vendas.

Exercício 7: Para determinar o volume ϕ [hm³] de água contido em uma usina hidrelétrica a partir da altura x [m] do reservatório deve-se usar um polinômio cota por volume tal como dado na Figura 7.1.

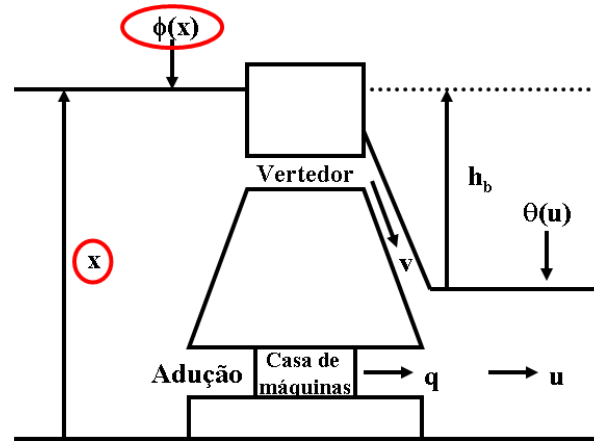


Figura 7.1: Dados de uma usina hidrelétrica.

Sejam os polinômios cota por volume das usinas da Tabela 7.1 e os valores de altura do reservatório da Tabela 5.2, construir um programa que calcule, utilizando a forma de Horner, os volumes de cada usina para cada valor de altura do reservatório.

Usina	Coeficientes do Polinômio (Altura [m] x Volume [hm ³])				
	C ₀	C ₁	C ₂	C ₃	C ₄
Furnas	736	3,19x10 ⁻³	-1,6x10 ⁻⁷	5,07x10 ⁻¹²	-6,50x10 ⁻¹⁷
Emborcação	568	1,45x10 ⁻²	-1,2x10 ⁻⁶	5,83x10 ⁻¹¹	-1,12x10 ⁻¹⁵
Ilha Solteira	293	3,60x10 ⁻³	-1,8x10 ⁻⁷	5,87x10 ⁻¹²	-7,50x10 ⁻¹⁷
Água Vermelha	350	5,50x10 ⁻³	-3,3x10 ⁻⁷	9,47x10 ⁻¹²	0,00x10 ⁻¹⁷

Tabela 7.1: Coeficientes dos polinômios cota por volume para 4 usinas hidrelétricas.

Usina	Altura [m]
Furnas	20
Emborcação	15
Ilha Solteira	18
Água Vermelha	30

Tabela 7.2: Alturas a serem utilizadas para calcular o volume de cada usina.

Para resolver este problema você deverá criar duas classes cujos diagramas UML são fornecidos nas Figuras 7.2 e 7.3.

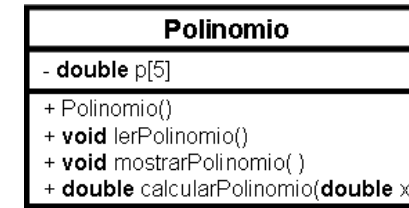


Figura 7.2: Campos e métodos da Classe **Polinômio**.

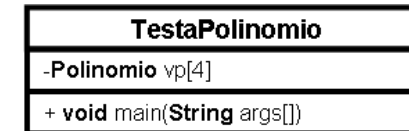


Figura 7.3: Campos e métodos da Classe **TestaPolinômio**.

Os principais passos para resolver este problema são:

- (1) Criar a classe Polinômio.
- (2) Criar a classe TestaPolinômio que cria um vetor de polinômios (tamanho 4) cujos dados são fornecidos na Tabela 7.1. Observe que toda vez que um objeto da classe Polinômio for criado, o construtor Polinômio deve chamar o método **void lerPolinomio()** que preenche o campo **p[]** empregando um método **nextDouble** (empregue a classe **Scanner**).
- (3) Ainda na classe TestaPolinômio, depois de criar todos os polinômios (se quiser pode criar um vetor de objetos da classe **Polinômio**), mande calcular o Volume através do método **double calcularVolume(double x)** para cada objeto **Polinômio** de modo a calcular o valor do polinômio em uma dada altura x .
- (4) Para calcular o valor do polinômio é necessário chamar um outro método denominado **Horner**. Este método deverá empregar a definição da Figura 7.4.

O cálculo de um polinômio $p(x) = c_0x^0 + c_1x^1 + c_2x^2 + \dots + c_nx^n$ em um dado ponto x pode ser realizado de forma mais eficiente se for utilizada a forma dos parênteses encaixados ou algoritmo de Horner. Neste caso, reescreve-se o polinômio tal como:

$$p(x) = c_0x^0 + c_1x^1 + c_2x^2 + \dots + c_nx^n$$

$$\rightarrow$$

$$p(x) = c_0 + x(c_1 + x(c_2 + \dots + x c_n))$$

Figura 7.4: Método de Horner para realizar o cálculo de um polinômio em x .

Exercício 8: A partir de um experimento de física são obtidos n dados experimentais de modo que cada medida i do experimento pode ser associada a um ponto P_i de coordenadas (x_i, y_i) tal como mostrado na Figura 8.1.

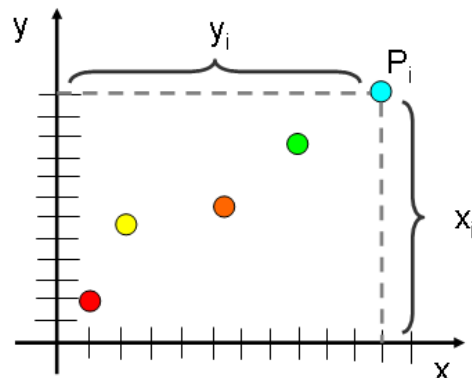


Figura 8.1: Representação dos dados experimentais obtidos.

Seja um conjunto de n pontos tal como dado na Tabela 8.1.

x	0.5	1.0	2.0	3.0	4.0
y	25	24	20	18	15

Tabela 8.1: Dados experimentais obtidos.

A partir deste conjunto de dados pretende-se obter o coeficiente angular **a** e o coeficiente independente **b** da reta $y = ax + b$ de modo que esta reta é tal que minimiza os desvios entre os pontos experimentais e a reta y obtida. Os valores de **a** e **b** podem ser obtidos por meio das equações (1) e (2).

$$a = \frac{n \sum x_i y_i - (\sum x_i)(\sum y_i)}{n \sum x_i^2 - (\sum x_i)^2} \quad (1)$$

$$b = \frac{(\sum y_i)(\sum x_i^2) - (\sum x_i y_i)(\sum x_i)}{n \sum x_i^2 - (\sum x_i)^2} \quad (2)$$

Para resolver este problema será necessário seguir os seguintes passos:

(1) Criar a classe Ponto e TestaPonto cuja descrição de campos e métodos é dada no diagrama UML das Figuras 8.2 e 8.3, respectivamente.

(2) Criar a classe TestaPonto que deverá realizar as seguintes operações:

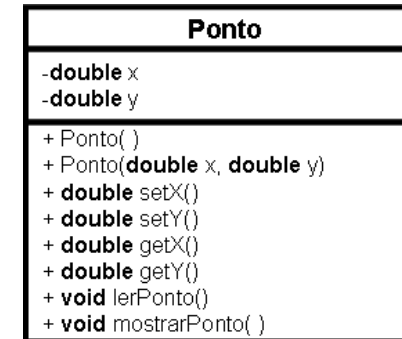


Figura 8.2: Campos e métodos da classe Ponto.

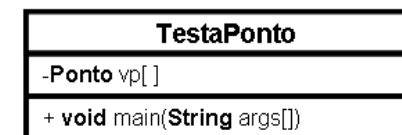


Figura 8.3: Campos e métodos da classe TestaPonto.

(2.1) Pedir o número de pontos para o usuário e criar um vetor de objetos Ponto de tamanho adequado.

(2.2) Ao se construir um objeto Ponto automaticamente será invocado o método leitura da classe Ponto que irá pedir para o usuário as coordenadas (x, y) de cada ponto experimental.

(2.3) Após completar todos os elementos do vetor, calcular as seguintes quantidades:

$$\sum_{i=1}^n x_i, \sum_{i=1}^n y_i, \sum_{i=1}^n x_i y_i \text{ e } \sum_{i=1}^n x_i^2.$$

(2.4) Empregar as quantidades calculadas no item (2.3) para encontrar os valores de **a** e **b** de acordo com as equações (1) e (2).

(2.5) Imprimir os valores de **a** e **b** obtidos.

Exercício 9: Vetores de caracteres podem ser utilizados para representar cadeias de DNA. As quatro bases encontradas em um DNA são: Adenina (A), Citosina (C), Guanina (G) e Timina (T). Dada as seqüências da Tabela 9.1, construir um programa que forneça quais as seqüências em que a Adenina aparece mais de 4 vezes.

Seqüência	Bases									
DNA 1	A	T	G	C	A	A	C	T	A	
DNA 2	G	G	C	C	A	A	T	A	T	
DNA 3	C	A	A	T	G	C	C	A	C	
DNA 4	T	T	G	C	C	C	T	T	C	

Tabela 9.1: Seqüências de DNA.

Para tanto, empregue a classes DNA e TestaDNA cuja descrição é dada nas Figuras 9.1 e 9.2, respectivamente.

DNA
-char seq[]
+ DNA() + void setDNA(int i, char c) + char getDNA(int i) + void lerDNA() + void mostrarDNA() + int frequenciaDNA(char b)

Figura 9.1: Campos e métodos da classe DNA.

TestaDNA
-DNA vd[]
+ void main(String args[])

Figura 9.2: Campos e métodos da classe TestaDNA.

Observe que:

- (1) O construtor sem parâmetros chama o método **lerDNA**.
- (2) Os métodos **set** e **get** devem realizar uma validação nos moldes do **Exercício 2**.
- (3) O método **lerDNA** realiza a leitura, via teclado, das bases a serem armazenadas no campo **seq**.
- (4) O método **mostrarDNA** deverá mostrar todas as bases armazenadas no campo **seq** de um objeto da classe **DNA**.
- (5) O método **frequenciaDNA** tem como parâmetro de entrada um caractere que corresponde a base com a qual se quer a freqüência que ela aparece no campo **seq** (que serve para armazenar a seqüência de bases de um DNA).

Exercício 10: Diz-se que uma dada seqüência de DNA possui um certo grau de similaridade de acordo com o número de bases que aparecem na mesma ordem. Por exemplo, dadas as duas seqüências da Figura 10.1 possuem similaridade 3.

Seqüência	Bases					
DNA 1	A	T	C	G	T	C
	↔	↔	↔	↔	↔	↔
DNA 2	A	C	T	G	T	T
	=	=	=	=	=	=
Similar	1	0	0	1	1	0

Figura 10.1: Comparação entre duas seqüências de DNA.

Modificar o **Exercício 9** de modo que um novo método, denominado **similarDNA**, deverá ter como parâmetro de entrada um objeto DNA e compara o campo **seq** do parâmetro de entrada com o campo **seq** do objeto que invocou o método. Por exemplo:

```
int s12 = d1.similar(d2);
```

O comando acima realiza uma comparação do conteúdo do campo **seq** de d1 com o conteúdo do campo **seq** de d2 nos moldes da descrição da Tabela 10.1. Depois, é retornado o valor de similaridade que será armazenado na variável s12.

Empregando o novo método, **similarDNA**, forneça os valores de similaridade entre as seqüências de DNA da Tabela 9.1, tal que preenche os valores faltantes da Tabela 10.2.

	DNA 1	DNA 2	DNA 3	DNA 4
DNA 1	9	a ₁₂	a ₁₃	a ₁₄
DNA 2	a ₂₁	9	a ₂₃	a ₂₄
DNA 3	a ₃₁	a ₃₂	9	a ₃₄
DNA 4	a ₄₁	a ₄₂	a ₄₃	9

Tabela 9.2: Informações de similaridade entre as seqüências da Tabela 9.1.

Exercício 11: Construir a classe Matriz com campos e métodos dados na Figura 11.1.

MatrizDouble
- double elem[][]
+ MatrizDouble(int nL, int nC) + void mostrarMatriz() + void prodExterno(double v1[], double v2[]) + void somarMatriz(MatrizDouble m2) + void multiplicarMatriz(MatrizDouble m2) + double[] multiplicarVetor(double v1[])

Figura 11.1: Campos e métodos da classe **MatrizFloat**.

Item (A): Construir o método construtor `MatrizDouble(double nL, double nC)` que aloca para o campo `elem` uma matriz do tipo **double** com `nL` linhas e `nC` colunas.

Item (B): Construir o método construtor **void** `mostrarMatriz()` que imprime os valores de uma matriz do tipo **double**, com `nL` linhas e `nC` colunas elementos, contidos no campo `elem`.

Item (C): Dados dois vetores v e t de dimensão n , construir o método **prodExterno** que calcula, e depois armazena o produto externo de v por t em uma matriz A , tal como descrito na Equação 1. Mostre a matriz A obtida.

$$\begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix} * \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{bmatrix} = \begin{bmatrix} v_1 * t_1 & v_1 * t_2 & \dots & v_1 * t_n \\ v_2 * t_1 & v_2 * t_2 & \dots & v_2 * t_n \\ \vdots & \vdots & \ddots & \vdots \\ v_n * t_1 & v_n * t_2 & \dots & v_n * t_n \end{bmatrix} \quad (1)$$

Item (D): Construir o método `somarMatriz` que realiza a soma de duas matrizes A e B , e armazena o resultado em uma matriz A . Por exemplo: `A.somarMatriz(B)`.

Item (E): Construir o método `somarMatriz` que realiza a multiplicação de duas matrizes A e B , e armazena o resultado em uma matriz A . Por exemplo: `A.multiplicarMatriz(B)`.

Item (F): Construir o método `multiplicarVetor` que realiza a multiplicação de uma matriz A por um vetor v e retorna o resultado em um vetor r .

Exercício 12: Um caractere pode ser digitalizado, bastando que uma grade seja aplicada a sua imagem. Depois, para cada área da grade por onde passe a linha correspondente ao desenho do caractere é associado o valor 1. Se a linha não passar por aquela área é associado o valor 0. O esquema de transformação é dado na Figura 12.1.

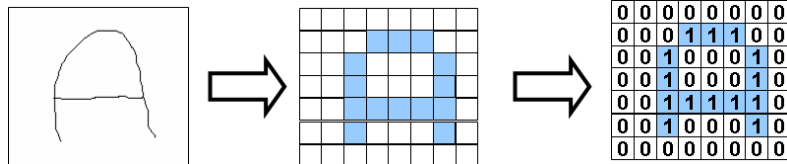


Figura 12.1: Esquema de digitalização e armazenamento de um caractere.

Supondo que as matrizes tenham dimensão 8×8 , construir um programa que:

- (A) Gera aleatoriamente os valores de duas imagens (matrizes 8×8).
- (B) Imprime as duas imagens (matrizes 8×8) tal como esquema da Figura 11.1.

(C) Calcula a similaridade entre as duas imagens aleatórias (dadas duas imagens associadas a matrizes A e B , a similaridade é o número elementos das matrizes tais que $a_{ij} == b_{ij}$).

(D) Comparar as duas matrizes geradas aleatoriamente com a matriz associada a letra A (a matriz correspondente a letra A está descrita na Figura 12.1). A comparação deve ser realizada de modo a determinar qual o grau de similaridade entre os padrões aleatórios e o padrão da letra A da Figura 12.1. Se a similaridade for maior que 90%, o programa deve fornecer uma mensagem dizendo que o padrão aleatório é a letra A .

Para construir o programa é necessário empregar a classe `Imagem` descrita na Figura 12.2.

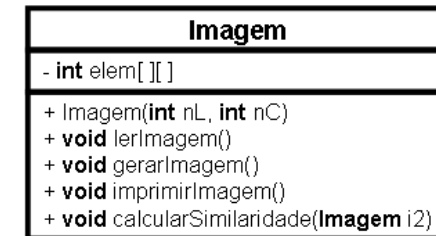


Figura 12.2: Diagrama UML da classe `Imagem`.

O detalhamento dos métodos é dado na Tabela 12.1.

Método	Descrição
<code>Imagem</code>	Construtor com parâmetros inteiros <code>nL</code> e <code>nC</code> que correspondem ao número de linhas e colunas da matriz associada ao campo <code>elem</code> . Inicialmente insere valores iguais a zero na matriz <code>elem</code> .
<code>lerImagem</code>	Permite a inserção de valores para cada elemento da matriz <code>elem</code> .
<code>gerarImagem</code>	Gera aleatoriamente os elementos contidos em <code>elem</code> .
<code>imprimirImagem</code>	Imprime o campo <code>elem</code> nos moldes da Figura 12.1.
<code>calcularSimilaridade</code>	Calcula a similaridade entre duas imagens (matrizes).

Tabela 12.1: Métodos da classe `Imagem` e sua descrição.

Exercício 1: Construir uma Classe **Horario** cujo diagrama UML é dado na Figura 1.1.

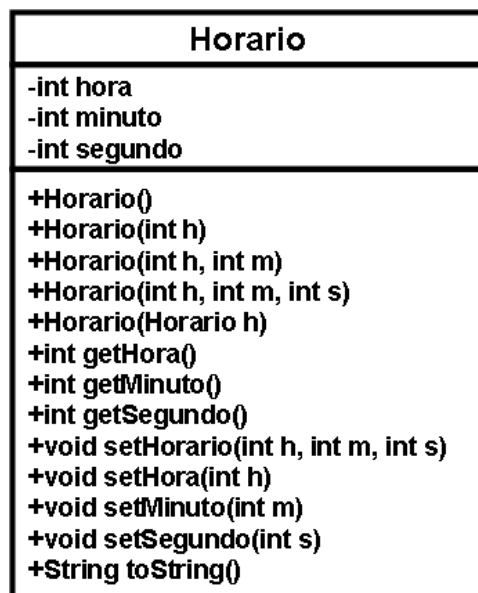


Figura 1.1: Campos e métodos da Classe **Horario**.

Dado diagrama UML da Figura 1.1, pede-se:

Item (A): Implementar os métodos tal como dado no detalhamento dos métodos dado na Tabela 1.1.

Método	Descrição
Horario()	Construtor sem parâmetros que inicializa os campos hora, minuto e segundo com o valor zero. Chama o método setHorario().
Horario(int h)	Construtor que inicializa o campo hora com o valor h e os demais com valor zero. Chama o método setHorario().
Horario(int h, int m)	Construtor que inicializa o campo hora com o valor h e o campo minuto com o valor m. Chama o método setHorario().
Horario(int h, int m, int s)	Construtor que inicializa o campo hora com o valor h, o campo minuto com m e o campo segundo com s. Chama o método setHorario().

Horario(Horario H)	Construtor que inicializa o campo hora com H.hora, o campo minuto com o valor contido em H.minuto e o campo segundo com o valor contido em H.segundo. Chama o método setHorario().
int getHora()	Método que retorna o valor contido no campo hora.
int getMinuto()	Método que retorna o valor contido no campo minuto.
int getSegundo()	Método que retorna o valor contido no campo segundo.
void setHorario(int h, int m, int s)	Método que é chamado por todos os construtores para modificar os campos hora, minuto e segundo com os valores h, m e s, respectivamente. Estas modificações são realizadas através de chamadas aos métodos setHora(), setMinuto() e setSegundo(), respectivamente.
void setHora(int h)	Método que modifica o valor do campo hora para o valor h.
void setMinuto(int m)	Método que modifica o valor do campo minuto para o valor m.
void setSegundo(int s)	Método que modifica o valor do campo segundo para o valor s.

Tabela 1.1: Métodos da classe **Horario** e suas descrições.

Item (B): Implementar mais métodos tal como dado no detalhamento dos métodos dado na Tabela 1.2.

Método	Descrição
void somarHoras(Horario h2)	Método que realiza a soma de dois objetos da classe hora h1 e h2 e armazena o resultado em h1. A chamada ao método pode ser realizada através do comando: h1.somarHoras(h2); Cuidado, o valor máximo de um horário é: 23:59:59. Se a soma for maior, mostrar uma mensagem apropriada e retornar 23:59:59.
void subtrairHoras(Horario h2)	Método que realiza a soma de dois objetos da classe hora h1 e h2 e armazena o resultado em h1. A chamada ao método pode ser realizada através do comando: h1.subtrairHoras(h2); Cuidado, o valor mínimo de um horário é: 00:00:00. Se a subtração for menor, mostrar uma mensagem apropriada e retornar 00:00:00.
int numSegundos()	Método que retorna o número de segundos contidos em um dado objeto da classe Horario .
void mostrarAMPM()	Método que mostra o conteúdo dos campo segundo, minuto e hora em notação militar, ou seja, se a hora for 23:40:01 será mostrada, com este método, da seguinte forma: 11:40:01 PM. Se a hora for 11:40:01 (antes do meio-dia), então, será mostrada da seguinte forma: 11:40:01.

Tabela 1.2: Mais métodos da classe **Horario** e suas descrições.

Exercício 2: Construir uma classe **Data** cujos campos e métodos são dados na Figura 2.1.

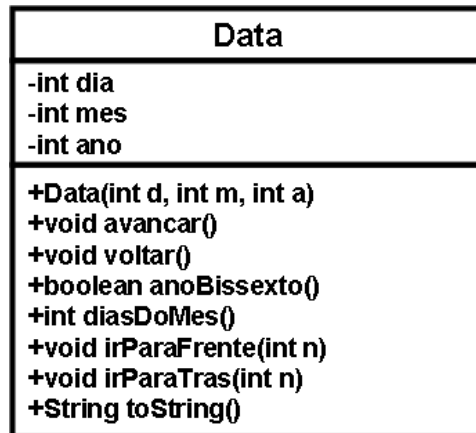


Figura 2.1: Campos e métodos da Classe **Data**.

Para implementar os métodos da Figura 2.1 segue o detalhamento dos métodos dado na Tabela 2.1.

Método	Descrição
Data(int d, int m, int a)	Construtor com três parâmetros inteiros d, m e a que inicializam os campos dia, mês e ano, respectivamente, através do método setData().
setData(int d, int m, int a)	Método responsável por modificar os campos dia, mes e ano com os valores d, m e a, respectivamente. Se os valores d, m e a não forem válidos, então, usar os valores 1, 1 e 1900, respectivamente.
void avancar()	Método que deve avançar em 1 dia a data atual, mantendo os campos da classe em estado coerente.
void voltar()	Método que reduz em 1 dia a data atual, mantendo os campos da classe em estado coerente.
boolean anoBissexto()	Método que retorna true se O campo ano contiver um ano tal que: (1) Seja divisível por 4, mas não por 100. (2) Seja divisível por 100 e 400.
int diaDoMes()	Método que retorna o número de dias do mês, considerando se o ano é bissexto ou não.
void irParaFrente(int n)	Método que incrementa em n dias a data contida nos campos dia, mes e ano, mantendo estado coerente.
void irParaTras(int n)	Método que decrementa em n dias a data contida nos campos dia, mes e ano, mantendo estado coerente.

Tabela 2.1: Métodos da classe **Data** e suas descrições.

Exercício 3: Construir uma classe **HistoricoChuvvas** cujo diagrama UML é dado na Figura 3.1 e métodos são descritos na Tabela 3.1.

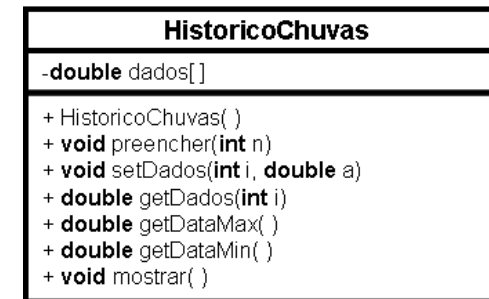


Figura 3.1: Diagrama UML da classe **HistoricoChuvvas**.

Método	Descrição
HistoricoChuvvas()	Construtor sem parâmetros que pede ao usuário o número n de dados a serem inseridos no campo data e chama o método preencher.
void preencher(int n)	Método responsável por preencher os n elementos do campo (vetor) data. Para tanto, emprega um laço onde usa o método setDados e um método nextDouble de um objeto da classe Scanner.
void setDados(int i, double a)	Método que tentar inserir o valor a na posição i do campo data desde que i seja um índice válido. Senão imprime uma mensagem de erro.
double getDados(int i)	Método que retorna o valor contido em data[i], desde que i seja um índice válido. Senão, imprime uma mensagem de erro e retorna o valor zero.
double getDataMax()	Método que retorna o maior valor do histórico de chuvas armazenado em data.
double getDataMin()	Método que retorna o menor valor do histórico de chuvas armazenado em data.
void mostrar()	Método que mostra todos os campos os elementos contidos no campo data, empregando um laço onde o método getDados é chamado.

Tabela 3.1: Métodos da classe **HistoricoChuvvas** e suas descrições.

Pede-se ainda para construir uma classe TestaHistoricoChuvvas com o intuito de se testar o emprego de um objeto da classe HistoricoChuvvas.

Exercício 4: Deseja-se criar uma classe cujo diagrama UML é dado na Figura 4.1. Uma descrição dos métodos é dada na Tabela 4.1.

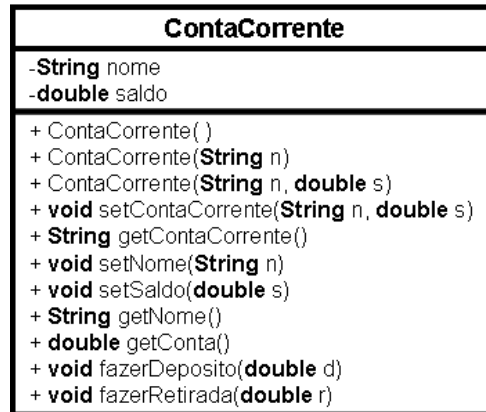


Figura 4.1: Diagrama UML da classe **ContaCorrente**.

ContaCorrente()	Construtor sem parâmetros que inicializa os campos nome e saldo com os valores "SemNome" e 0.0, respectivamente, usando o método setContaCorrente.
ContaCorrente(String n)	Construtor que inicializa o campo nome com n e o campo saldo com 0.0, usando setContaCorrente.
ContaCorrente(String n, double s)	Construtor que inicializa o campo nome com n e o campo saldo com s, usando setContaCorrente.
Void setContaCorrente(String n, double s)	Método que chama setNome e setSaldo para inicializar os campos nome e saldo com os valores n e s, respectivamente.
String getContaCorrente()	Método que usa os métodos getNome e getConta para retornar uma String contendo o conteúdo dos campos nome e saldo no seguinte formato: <nome> seu saldo atual é de <saldo>.
Void setNome(String n)	Método que modifica o campo nome para n.
void setSaldo(double s)	Método que modifica o campo saldo para s.
String getNome()	Método que retorna o conteúdo do campo nome.
double getConta()	Método que retorna o conteúdo do campo conta.
void fazerDeposito(double d)	Método que modifica o conteúdo do campo saldo para saldo + d (usar setSaldo).
void fazerRetirada(double r)	Método que modifica o conteúdo do campo saldo para saldo - d (usar setSaldo).

Tabela 4.1: Métodos da classe **ContaCorrente** e suas descrições.

Exercício 5: Deseja-se criar uma classe cujo diagrama UML é dado na Figura 5.1. Uma descrição dos métodos é dada na Tabela 5.1.

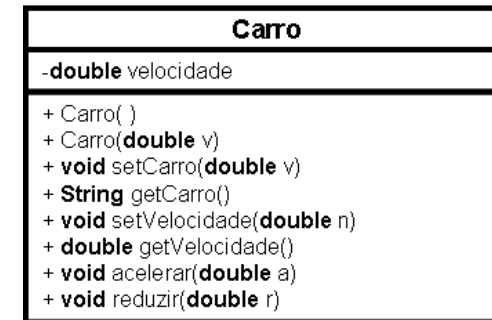


Figura 5.1: Diagrama UML da classe **Carro**.

Carro()	Construtor sem parâmetros que inicializa o campo velocidade com o valor 0.0, usando o método setCarro.
Carro(double n)	Construtor que inicializa o campo velocidade com 0.0, usando setCarro.
Void setCarro(double v)	Método que chama setVelocidade para inicializar o campo velocidade com o valor v.
String getCarro()	Método que usa getVelocidade e depois retorna uma String contendo o conteúdo do campo velocidade no seguinte formato: A velocidade atual do carro é de <velocidade>.
void setVelocidade(double n)	Método que modifica o campo velocidade para n, desde que n não seja negativa.
double getVelocidade()	Método que retorna o conteúdo do campo velocidade.
void acelerar(double a)	Método que modifica o conteúdo do campo velocidade para velocidade + a (usar setVelocidade), mas desde que o valor de a seja maior ou igual a zero e menor que 20. Senão, imprime a mensagem "Não foi possível acelerar" e mantém o valor atual do campo velocidade.
void reduzir(double r)	Método que modifica o conteúdo do campo velocidade para velocidade - r (usar setVelocidade), mas desde que o valor de r seja maior ou igual a zero e menor que 30. Senão, imprime a mensagem "Não foi possível reduzir" e mantém o valor atual do campo velocidade.

Tabela 5.1: Métodos da classe **Carro** e suas descrições.

Exercício 6: Deseja-se criar uma classe cujo diagrama UML é dado na Figura 6.1. Uma descrição dos métodos é dada na Tabela 6.1.

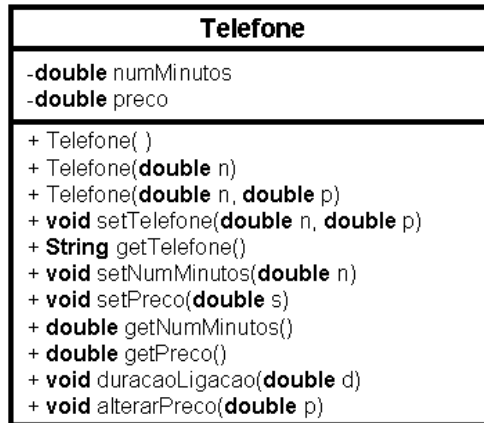


Figura 6.1: Diagrama UML da classe **Telefone**.

Telefone()	Construtor sem parâmetros que inicializa os campos numMinutos e preco com os valores 0.0 e 0.0, respectivamente, usando o método setTelefone.
Telefone(double n)	Construtor que inicializa o campo numMinutos com n e o campo preco com 0.0, usando setTelefone.
Telefone(double n, double p)	Construtor que inicializa o campo numMinutos com n e o campo preco com p, usando setTelefone.
void setTelefone(double n, double p)	Método que chama setNome e setSaldo para inicializar os campos numMinutos e preco com os valores n e p, respectivamente.
String getTelefone()	Método que usa os métodos getNumMinutos e getPreco para retornar uma String contendo o conteúdo dos campos numMinutos e preco no seguinte formato: Total de minutos <numMinutos> a um preço de <preco>.
void setNumMinutos(double n)	Método que modifica o campo numMinutos para n, desde que n seja maior ou igual a zero, senão imprime uma mensagem de erro e mantém o valor atual de numMinutos.
void setPreco(double s)	Método que modifica o preco para s desde que s seja maior ou igual a zero, senão imprime uma mensagem de erro e mantém o valor atual de preco.
String getNumMinutos()	Método que retorna o conteúdo do campo numMinutos.
double getPreco()	Método que retorna o conteúdo do campo preco.

void duracaoLigacao(double d)	Altera o campo preco para o valor contido no parâmetro d por meio do método setNumMinutos.
void alterarPreco(double p)	Altera o campo preco para o valor contido no parâmetro d por meio do método setPreco.

Tabela 6.1: Métodos da classe **Telefone** e suas descrições.

Exercício 7: Deseja-se criar uma classe cujo diagrama UML é dado na Figura 7.1. Uma descrição dos métodos é dada na Tabela 7.1.

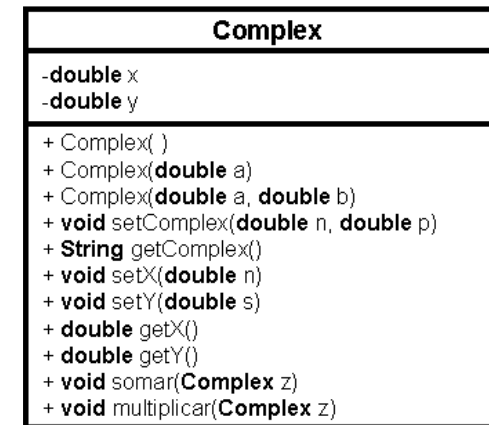


Figura 7.1: Diagrama UML da classe **Complex**.

Complex()	Construtor sem parâmetros que inicializa os campos x e y com os valores 0.0 e 0.0, respectivamente, usando o método setComplex.
Complex(double a)	Construtor que inicializa o campo x com a e o campo y com 0.0, usando setComplex.
Complex(double a, double b)	Construtor que inicializa o campo x com a e o campo y com b, usando setComplex.
void setComplex(double a, double b)	Método que chama setX e setY para inicializar os campos x e y com os valores a e b, respectivamente.
String getComplex()	Método que usa os métodos getX e getY para retornar uma String contendo o conteúdo dos campos x e y no seguinte formato: z=<x>+<y>*i.
void setX(double n)	Método que modifica o campo x para n.
void setY(double s)	Método que modifica o campo y para s.
String getX()	Método que retorna o conteúdo do campo x.
double getY()	Método que retorna o conteúdo do campo y.

void somar(Complex z)	Realiza a soma de dois números complexos (objetos): o objeto do parâmetro de entrada e o objeto que invocou o método. O resultado é armazenado nos campos do objeto que invocou o método.
void mutiplicar(Complex z)	Realiza a multiplicação de dois números complexos (objetos) e armazena o resultado no objeto que invocou o método. Para tanto, use a seguinte Equação: $z_3 = z_1 z_2 \Leftrightarrow (x_3, y_3) = (x_1 x_2 - y_1 y_2, x_1 y_2 + x_2 y_1)$

Tabela 7.1: Métodos da classe **Complex** e suas descrições.

Exercício 8: Deseja-se criar uma classe cujo diagrama UML é dado na Figura 8.1. Uma descrição dos métodos é dada na Tabela 8.1.

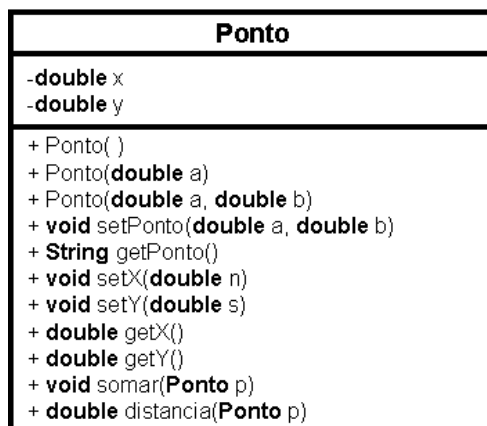


Figura 8.1: Diagrama UML da classe **Ponto**.

Ponto()	Construtor sem parâmetros que inicializa os campos x e y com os valores 0.0 e 0.0, respectivamente, usando o método setPonto.
Ponto(double a)	Construtor que inicializa o campo x com a e o campo y com 0.0, usando setPonto.
Ponto(double a, double b)	Construtor que inicializa o campo x com a e o campo y com b, usando setPonto.
void setPonto(double a, double b)	Método que chama setX e setY para inicializar os campos x e y com os valores a e b, respectivamente.
String getPonto()	Método que usa os métodos getX e getY para retornar uma String contendo o conteúdo dos campos x e y no seguinte formato: (x, y) = (<x>, <y>).
void setX(double n)	Método que modifica o campo x para n.

void setY(double s)	Método que modifica o campo y para s.
String getX()	Método que retorna o conteúdo do campo x.
double getY()	Método que retorna o conteúdo do campo y.
void somar(Ponto p)	Realiza a soma de dois pontos (objetos): o objeto do parâmetro de entrada e o objeto que invocou o método. O resultado é armazenado nos campos do objeto que invocou o método.
double distancia(Ponto p)	Calcular a distância entre dois pontos (objetos) e depois retorna o resultado. Para tanto, use a seguinte Equação: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Tabela 8.1: Métodos da classe **Ponto** e suas descrições.

Exercício 1: O diagrama da Figura 1.1 fornece um indicativo da evolução das estrelas (extraído de <http://astro.if.ufrgs.br/estrelas/escola.htm>):

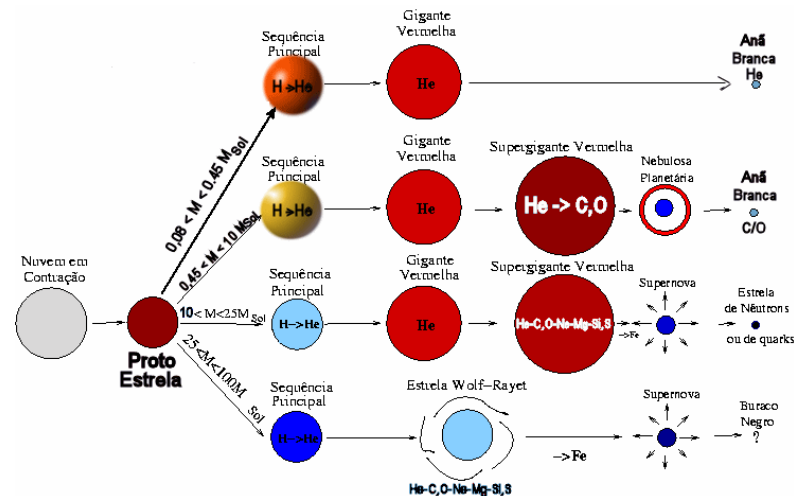


Figura 1.1: Evolução estelar para diferentes massas.

Tendo em vista a Figura 1.1, criar a hierarquia de classes dada na Figura 1.2:

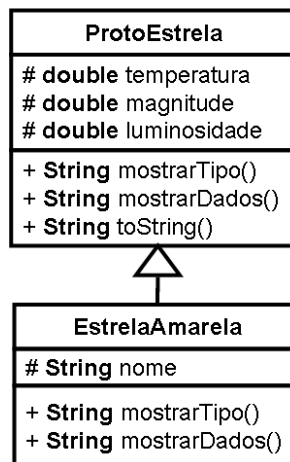


Figura 1.2: Hierarquia de classes.

Item (A): O método mostrarTipo() deverá mostrar o nome da classe, o método mostrarDados() deverá retornar o conteúdo de cada um dos campos (devidamente inicializados por construtores apropriados) e o campo nome deverá conter o nome do objeto (por exemplo, o nome da estrela do nosso Sistema Solar é "Sol").

Resolução:

```

public class ProtoEstrela
{
    protected double temperatura;
    protected double magnitude;
    protected double luminosidade;

    public ProtoEstrela() // Construtor sem parâmetros.
    {
        temperatura = 0.0;
        magnitude = 0.0;
        luminosidade = 0.0;
    }

    public ProtoEstrela(double a, double b, double c) // Sobrecarga.
    {
        temperatura = a;
        magnitude = b;
        luminosidade = c;
    }

    public String mostrarTipo() // Retorna a string com o nome da classe.
    {
        return "Sou uma Proto-Estrela";
    }

    public String mostrarDados() // Retorna string com valores nos campos.
    {
        return "Temperatura: " + temperatura +
            "Magnitude: " + magnitude +
            "Luminosidade: " + luminosidade;
    }

    public String toString() // Retorna string quando usar println.
    {
        return mostrarTipo() + mostrarDados();
    }
} // Fim Classe ProtoEstrela.
    
```

```

public class EstrelaAmarela extends ProtoEstrela // extends = herda de
{
    protected String nome;

    public EstrelaAmarela() // Construtor sem parâmetros.
    {
        temperatura = 0.0;
        magnitude = 0.0;
        luminosidade = 0.0;
        nome = "Desconhecida";
    }

    public EstrelaAmarela(double a, double b, double c, String s)
    {
        temperatura = a;
        magnitude = b;
        luminosidade = c;
        nome = s;
    }

    public String mostrarTipo() // Sobreposição: redefinindo o método dado
    { // na classe ProtoEstrela e que foi herdado.
        return "Nome: " + nome + "Sou uma EstrelaAmarela:";
    }
} // Fim da classe EstrelaAmarela.

```

```

public class TestaEstrelas
{
    public static void main( String args[] )
    {
        ProtoEstrela p = new ProtoEstrela();
        System.out.println(p);

        System.out.printf("\n\n");

        EstrelaAmarela e = new EstrelaAmarela(200000,1,10.0,"Sol");
        System.out.println(e);
    }
} // Fim da classe TestaEstrelas.

```

Item (B): Modificar a classe ProtoEstrela de modo que seus campos sejam **private**, bem como elaborar métodos **set** e **get** para modificar e retornar os valores contidos nos mesmos. Observe que esta modificação faz com que a classe EstrelaAmarela não tenha mais acesso direto aos campos declarados na classe ProtoEstrela, sendo, portanto, necessário empregar os métodos set nos construtores de EstrelaAmarela para inicializar temperatura, magnitude e luminosidade.

Resolução:

Na classe **ProtoEstrela** inserir os métodos:

```

public void setTemperatura(double a)
{temperatura = a;}

public void setMagnitude(double b)
{magnitude = b;}

public void setLuminosidade(double c)
{luminosidade = c;}

```

Na classe **EstrelaAmarela** modificar os construtores da seguinte forma:

```

public EstrelaAmarela()
{
    setTemperatura(0.0);
    setMagnitude(0.0);
    setLuminosidade(0.0);
    nome = "Desconhecida";
}

public EstrelaAmarela(double a, double b, double c, String s)
{
    setTemperatura(a);
    setMagnitude(b);
    setLuminosidade(c);
    nome = s;
}

```

Item (C): Modificar a classe ProtoEstrela de modo que o construtor sem parâmetros use o construtor com parâmetros por meio da palavra-chave **this**. Modificar, também, a classe EstrelaAmarela de modo que seus construtores empreguem os construtores da classe ProtoEstrela por meio da palavra-chave **super**.

Observação Importante: O uso das palavras-chave **this** e **super** irá funcionar independente do modificador de acesso nos campos da superclasse ser **protected** ou **private**. Isto decorre do fato de que **super** chama o **construtor** da **superclasse** e este tem total acesso aos campos.

Resolução:

Na classe **ProtoEstrela** modificar:

```
public ProtoEstrela() // Construtor chama construtor com parâmetros por this.
{
    this(0.0,0.0,0.0);
}
```

Na classe **EstrelaAmarela** modificar:

```
public EstrelaAmarela()
{
    super(0.0,0.0,0.0); // Chamada ao construtor da Classe ProtoEstrela.
    nome = "Desconhecida";
}

public EstrelaAmarela(double a, double b, double c, String s)
{
    super(a,b,c); // Chamada ao construtor da Classe ProtoEstrela.
    nome = s;
}
```

Exercício 2: O diagrama da Figura 2.1 fornece uma hierarquia de classes que relaciona os diferentes tipos de bichos de estimação.

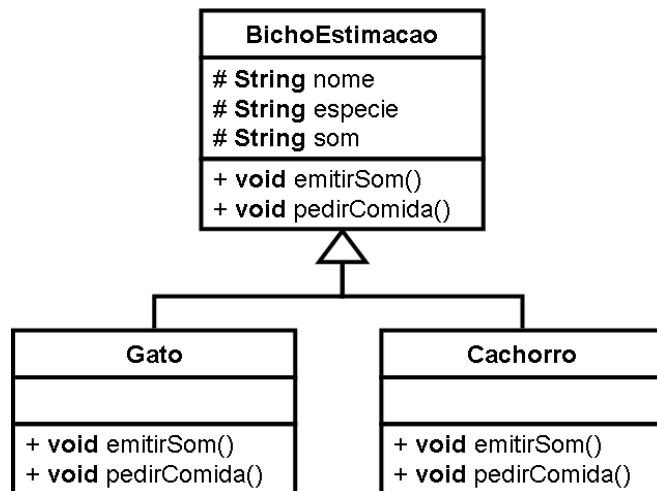


Figura 2.1: Hierarquia de classes para bichos de estimação.

A Tabela 2.1 fornece a mensagem que deve ser impressa por cada método de cada classe.

	void emitirSom()	void pedirComida()
Gato	Imprimir "Miau Miau"	Chamar o método void emitirSom() e depois imprimir "Gato faz olhar triste".
Cachorro	Imprimir "Au Au"	Chamar o método void emitirSom() e depois imprimir "Cachorro abana o rabo".

Tabela 2.1: Comportamento associado ao método de cada classe.

Exercício 3: O diagrama da Figura 3.1 fornece uma hierarquia de classes relacionadas com os diversos tipos de castelos.

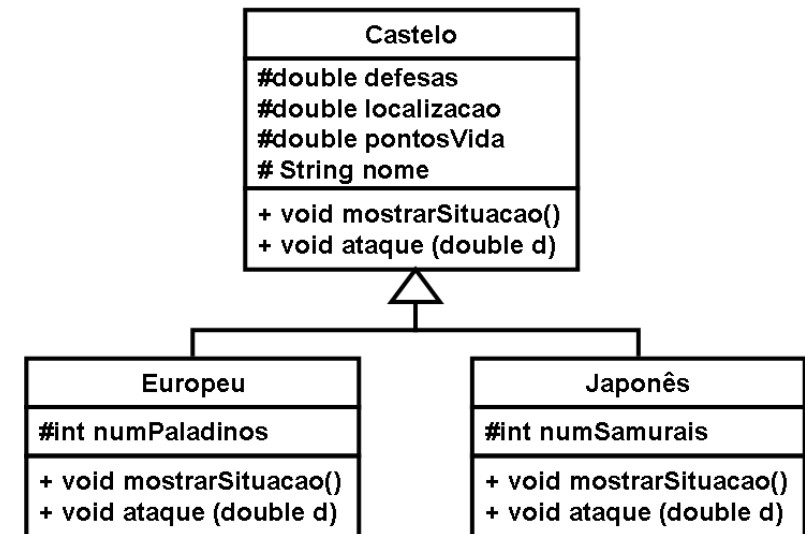


Figura 3.1: Hierarquia de classes para castelos.

A Tabela 3.1 indica como deve ser o comportamento de cada método em cada uma das classes:

	void mostrarSituacao()	void ataque(double d)
Castelo	Se o campo defesas for > 0.0, então, reduzir o valor deste campo, senão reduzir o valor do campo pontosVida.	Mostrar os valores contidos nos campos da classe. Se pontosVida == 0.0, então, mostrar também a mensagem o castelo foi destruído.
Europeu Japonês	Se o número de Paladinos ou Samurais for > 0, realizar um sorteio aleatório e eliminar entre 2 e 5 guerreiros. Senão, proceder como no método ataque da classe Castelo.	

Tabela 3.1: Comportamento de cada método para cada classe.

Exercício 4: O diagrama da Figura 4.1 fornece uma hierarquia de classes relacionadas com os diversos tipos de Karatekas. Já a Tabela 4.1 fornece os golpes que podem ser aplicados e qual a mensagem relativa ao dano que um dado golpe provoca (vide campo Dano). Por exemplo, se um método oizuki for chamado, então, um valor aleatório contido no intervalo $[1, \text{forca}/2]$ será gerado (Observe que forca deve ser maior ou igual que 4). Após o valor ser gerado será impresso na tela a mensagem: "Oizuki com dano %f deferido !". Além disso, o objeto que chamou o método terá o valor, contido no campo pontos de vida, reduzido de acordo com o estabelecido no campo Pontos de Vida da Tabela 4.1. Cuidado, pois se o Karateka tiver zero pontos de vida, então, a mensagem "Karateka cansado !" deverá ser apresentada. Por último a Tabela 4.2 mostra quais são os golpes que devem ser mostrados ao se invocar o método mostrarGolpes() de cada classe.

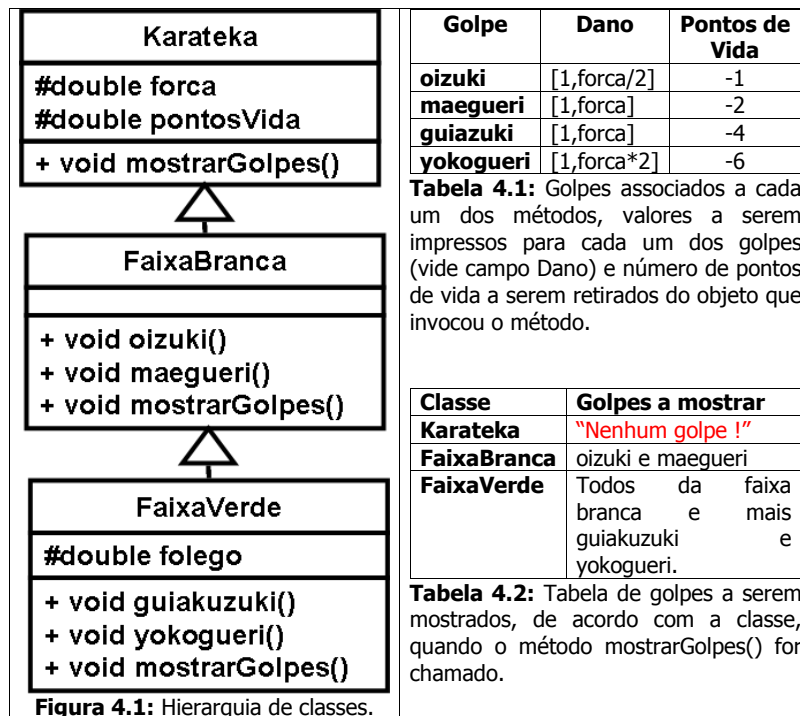


Figura 4.1: Hierarquia de classes.

Criar um programa que empregue as classes definidas na Figura 4.1 e mostre o funcionamento dos métodos dos objetos de cada uma das classes.

Exercício 5: Uma fábrica possui dois tipos de máquinas e estas possuem características comuns e específicas de acordo com o diagrama UML da Figura 5.1.

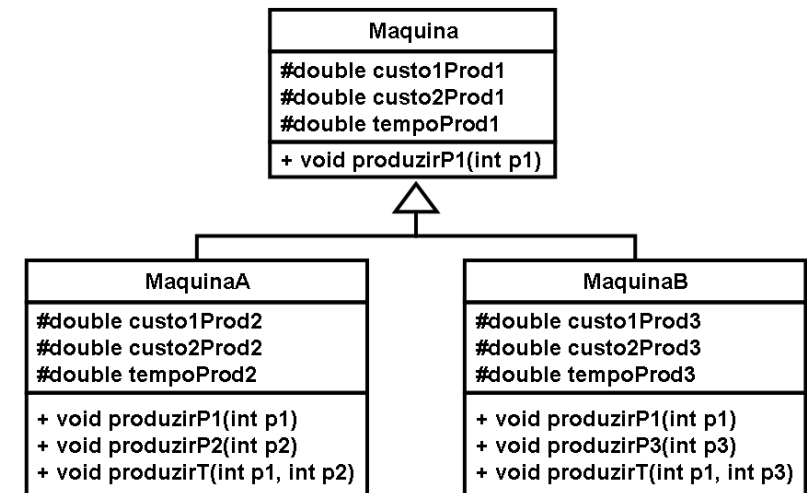


Figura 5.1: Diagrama UML com a hierarquia entre as classes.

Todas as máquinas possuem o método **void** produzirP1(int p) que calcula e exibe os custos de se produzir uma quantidade p de um produto P1. Para tanto, os campos custo1Prod1 e custo2Prod1 são empregados. De forma similar existem as funções produzirP2 (classe MaquinaA) e produzirP3 (classe MaquinaB). Ou seja, a Equação (1) é empregada para calcular os custos de fabricação do produto Pi em uma quantidade pi:

$$\text{custo} = \text{custo1Prodi} + \text{custo2Prodi} * \text{pi} \quad (1)$$

Já o campo tempoProdi fornece o tempo gasto para se produzir uma unidade do produto pi. Ou seja, o tempo total para a fabricação de um produto pi é dado pela Equação (2).

$$\text{tempo} = \text{tempoProdi} * \text{pi} \quad (2)$$

Os resultados das Equações (1) e (2) devem ser exibidos quando o método **void** produzirPi(int pi) for invocado. Finalmente o método **void** produzirT(int px, int py) contido nas classes MaquinaA e MaquinaB deve invocar os métodos **void** produzirPx(int px) e **void** produzirPy(int py).

Criar um programa que simule o funcionamento das classes MaquinaA e MaquinaB.

Exercício 6: Uma classe fábrica pode ser empregada para englobar dois tipos de fábricas: de carros e de navios tal como dado no diagrama UML da Figura 6.1.

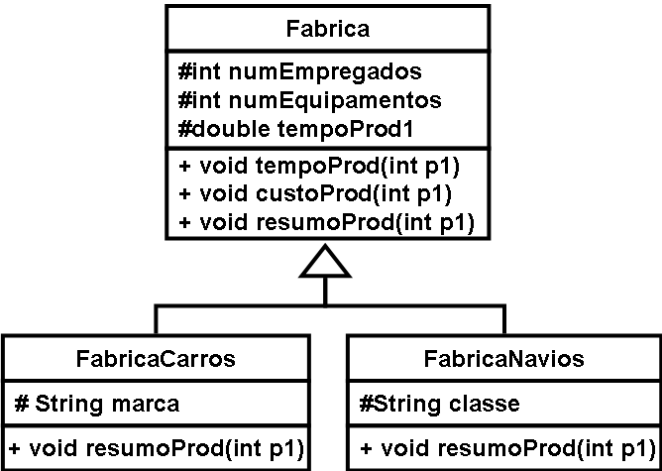


Figura 6.1: Hierarquia de classes para diferentes tipos de fábricas.

O método **void** tempoProd(**int** p1) exibe o tempo de produção necessário para se produzir uma quantidade p1 de acordo com a fórmula: $p1 / (\text{tempoProd1} * \sqrt{\text{numEquipamentos} * \text{numEmpregados}})$.

Já o método **void** custoProd(**int** p1) fornece o custo de produção dado por: $p1 * \sqrt{\text{numEquipamentos}} * (\text{numEmpregados})^2$.

Estes dois métodos são chamados pelo método void resumoProd(**int** p1) que tem o objetivo de fornecer os custos e tempo de produção de uma dada fábrica.

Além disso, o método resumoProd() presente nas classes FabricaCarros e FabricaNavios emprega o método resumoProd da classe Fabrica, mas usa, também imprime a informação contida no campo marca ou classe das classes FabricaCarros e FabricaNavios.

Criar um programa que simule o funcionamento de objetos do tipo FabricaCarros e FabricaNavios.

Exercício 7: Uma classe Navio é empregada para descrever os diversos tipos de navios, em particular ela serve de base para outras classes tal como descrito na Figura 7.1.

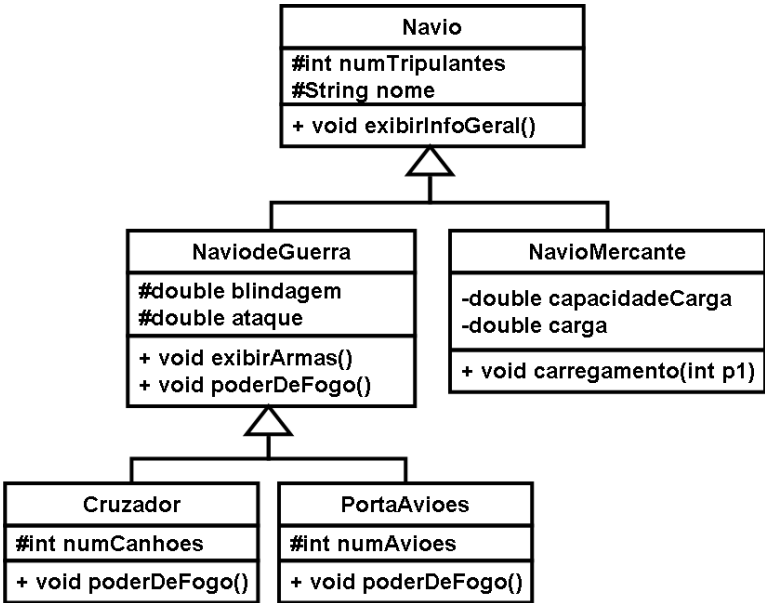


Figura 7.1: A classe Navio e suas subclasses.

A Tabela 7.1 resume a função que cada um dos métodos deve realizar.

Classe	Método	Descrição
Navio	exibirInfoGeral()	Exibe as informações contidas nos campos numTripulantes e nome.
NavioMercante	carregamento(int p1)	Exibe as informações dos campos da superclasse e também o volume do navio ocupado dado por: carga/capacidadeCarga.
NaviosdeGuerra	poderDeFogo()	Mostra o valor no campo ataque.
NaviosdeGuerra	exibirArmas()	Mostra os valores contidos nos campos da superclasse, o valor no campo blindagem e chama o método poderDeFogo().
Cruzador	poderDeFogo()	Mostra o valor calculado por $\text{ataque} * \sqrt{\text{numCanhoes}}$.
PortaAvioes	poderDeFogo()	Mostra o valor calculado por $\text{ataque} * \text{numAvioes}^2$.

Figura 7.1: Descrição dos métodos de cada classe.

Exercício 1: Criar uma classe **Particula** que herda campos e métodos de uma classe **Ponto** e que com o auxílio da classe **StdDraw** (disponível em <https://sites.google.com/site/pc22010/material-dos-topicos-de-aula/StdDraw.java?attredirects=0&d=1>) e da classe **TestaParticula** realiza a criação de animações tal como dado na Figura 1.1.

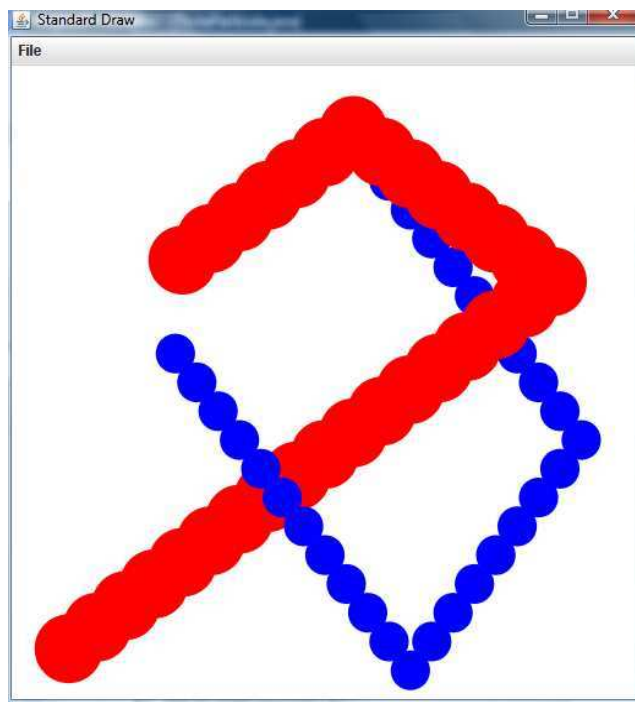


Figura 1.1: Desenho de objetos da classe **Ponto** com comandos da **StdDraw**.

Os métodos da classe **StdDraw** que são empregados nas classes **Ponto**, **Particula** e **TestaPonto** estão descritos na Tabela 1.1.

Comando	Descrição
<code>StdDraw.setXscale(0.0, 1.5);</code>	Define que a escala no eixo x da área de desenho está no intervalo [0.0, 1.0].
<code>StdDraw.setYscale(0.0, 1.5);</code>	Define que a escala no eixo y da área de desenho está no intervalo [0.0, 1.0].
<code>StdDraw.setPenColor(StdDraw.RED);</code>	Define a cor a ser empregada para o próximo comando de desenho, no caso a cor vermelha (RED). Outras cores podem ser usadas tais como GREEN, BLUE, YELLOW e MAGENTA.
<code>StdDraw.filledCircle(x, y, 0.05);</code>	Desenha um círculo preenchido nas coordenadas (x, y) e raio 0.05.

Tabela 1.1: Descrição dos métodos da classe **StdDraw**.

A seguir segue o diagrama UML da classe **Ponto** e **Particula** (Figura 1.2) e o código das classes **Ponto**, **Particula** e **TestaParticula** que constituem o programa. Não esquecer de baixar a classe **StdDraw** e incluir no projeto.

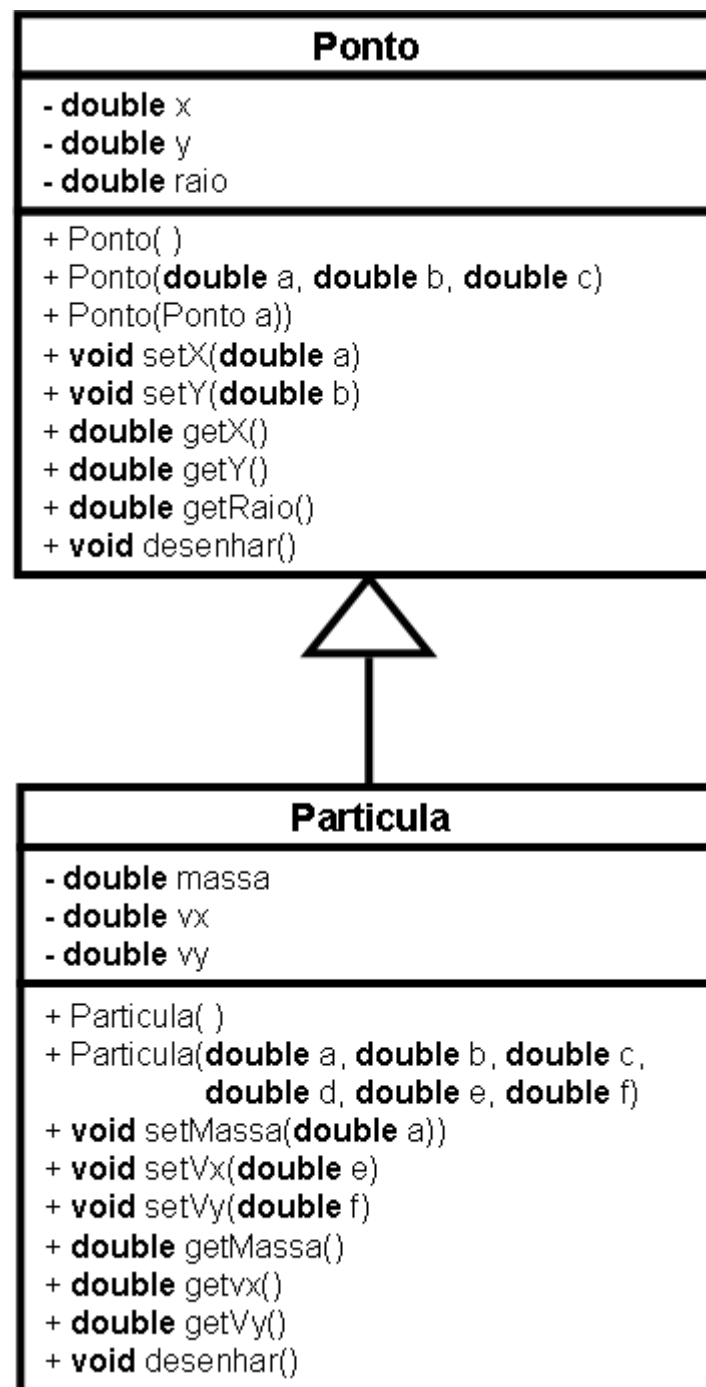


Figura 1.2: Diagrama UML das classes **Ponto** e **Particula**.

Classe Ponto

```
public class Ponto
{
    private double x;
    private double y;
    private double raio ;

    public Ponto()
    { this(0.0, 0.0, 0.01); }

    public Ponto(double a, double b, double c)
    { setX(a); setY(b); setRaio(c); }

    public void setX(double a)
    {    x = a; }

    public void setY(double b)
    {    y = b; }

    public void setRaio(double c)
    {    raio = (c > 0)?c:0.01;}

    public double getRaio()
    {return raio;}

    public double getX()
    {return x;}

    public double getY()
    {return y;}

    public void desenhar()
    { StdDraw.filledCircle(x, y, raio); }

}
```

Classe Particula

```
public class Particula extends Ponto
{
    private double massa;
    private double vx;
    private double vy;

    public Particula()
    {this(0.0,0.0,0.01,0.5,1,0.75);}
}
```

```

public Particula(double a, double b, double c, double d, double e,
double f)
{super(a,b,c); setMassa(d); setVx(e); setVy(f);}
public void setMassa(double d)
{
    massa = (d > 0)?d:0;
    setRaio(Math.sqrt(massa/Math.PI));
}

public void setVx(double e)
{vx = e;}

public void setVy(double f)
{vy = f;}

public double getMassa()
{return massa;}

public double getVx()
{return vx;}

public double getVy()
{return vy;}

public void desenhar()
{StdDraw.filledCircle(getX(),getY(),getRaio());}

public void mover()
{
    // Realiza a movimentação da partícula observando a parede !
    verificarLimite();
    setX(getX()+vx);
    setY(getY()+vy);
    desenhar();
}

public void verificarLimite()
{ // Verifica se a partícula colidiu com a parede !
    double aux1 = getX()+vx+getRaio();
    double aux2 = getY()+vy+getRaio();

    // Se houver colisão, então, mudar a direção da velocidade !
    if ((aux1 > 10.0)|| (aux1 < -10.0))
    {vx = -vx;}
    if ((aux2 > 10.0)|| (aux2 < -10.0))
    {vy = -vy;}
}

```

```

public void verificarColisao(Particula p)
{
    // Calculando a distância entre os centros de duas partículas !
    double dx = Math.pow((p.getX()-(getX())),2);
    double dy = Math.pow((p.getY()-(getY())),2);
    double d = Math.sqrt(dx + dy);
    // Calculando a distância entre os centros de 2 partículas !
    double r = p.getRaio() + getRaio();

    // Ocorreu uma colisão !!
    if (d <= r)
    {
        setVx(-(1.0*getVx()));
        setVy(-(1.0*getVy()));
        p.setVx(-(1.0*getVx()));
        p.setVy(-(1.0*getVy()));
    }
}

} // Fim Classe.

```

Classe TestaParticula

```

public class TestaParticula
{
    public static void main(String args[])
    {
        StdDraw.setXscale(-10.0,10.0);
        StdDraw.setYscale(-10.0,10.0);
        Particula p1 = new Particula(-10.0,-10.0,0.01,4.5,1,0.75);
        Particula p2 = new Particula(0.0,10.0,0.01,1.5,0.75,1);

        while(true)
        {
            // StdDraw.clear();
            StdDraw.setPenColor(StdDraw.RED);
            p1.mover();
            StdDraw.setPenColor(StdDraw.BLUE);
            p2.mover();
            p1.verificarColisao(p2);
            StdDraw.show(20);
        }
    }
}

```

Algumas perguntas importantes para entender o programa:

(i) Quais as funcionalidades e campos que a classe **Particula** adiciona em relação a classe **Ponto**?

- (ii) Como alterar a posição inicial, o tamanho, a cor e as velocidades iniciais das partículas? Descubra e realize testes para verificar o comportamento resultante.
- (iii) Como é detectada a colisão entre duas partículas?
- (iv) O comportamento duas partículas após uma colisão é determinado pelo código contido no método verificarColisão. Trocar o código existente pelos códigos 1 e 2 da Figura 1.3.

setVx(-1.1*getVx()); setVy(-1.2*getVy()); p.setVx(-0.8*getVx()); p.setVy(-0.9*getVy());	setVx(-(1.1*getVx())); setVy(-(1.2*getVy())); p.setVx(-(0.5*getVx())); p.setVy(-(0.5*getVy()));
Código 1	Código 2

Figura 1.3: Códigos a serem testados no método **verificarColisao()**.

Qual a mudança de comportamento que ocorre para cada código?

- (v) Inserir a seguinte modificação no comportamento das partículas:

- (A) Se uma partícula colidir com a parede, então, sua cor será laranja (ORANGE).
- (B) Se uma partícula colidir com outra, então, sua cor será verde (GREEN).

Exercício 2: Construir um novo programa composto das seguintes classes: Desenho, Ponto, Quadrado e TestaDesenho. O Relacionamento entre as classes Desenho, Ponto e Quadrado é dado no diagrama UML da Figura 2.1. O resultado a ser exibido na tela é dado na Figura 2.2.

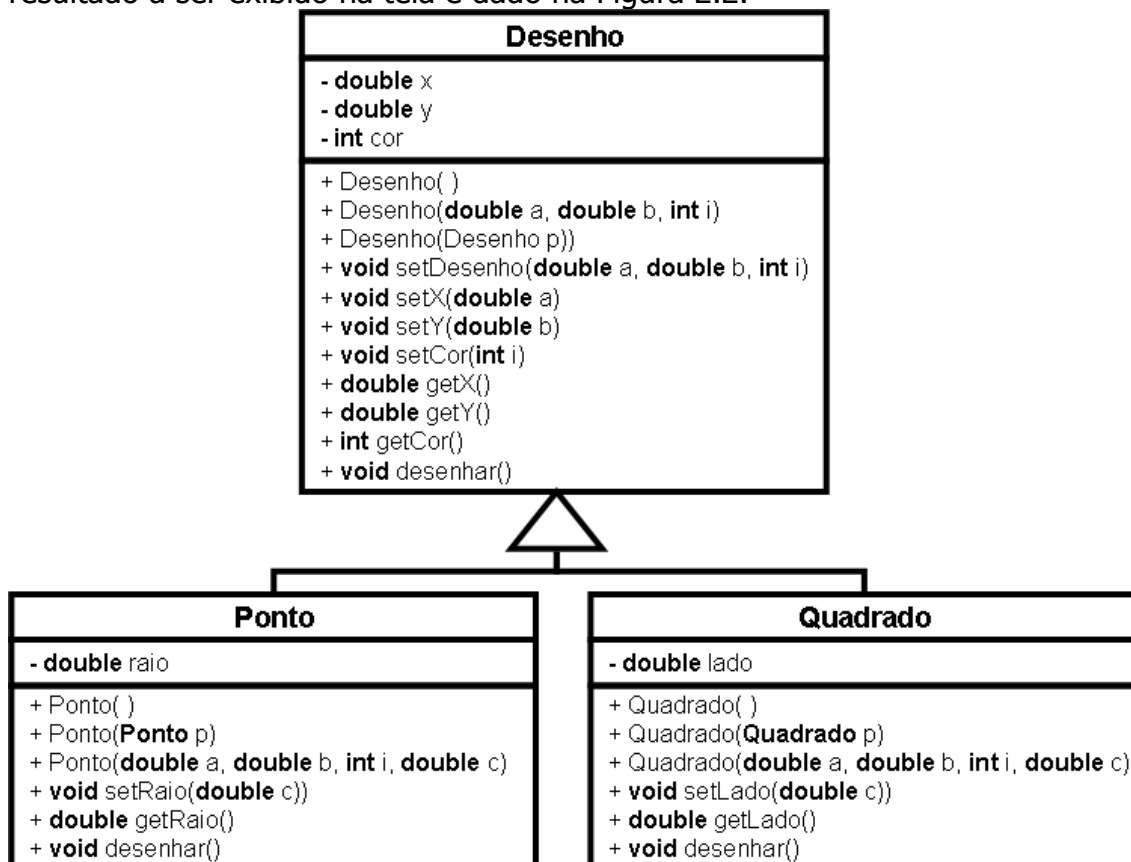


Figura 2.1: Diagrama UML das classes **Desenho**, **Ponto** e **Quadrado**.

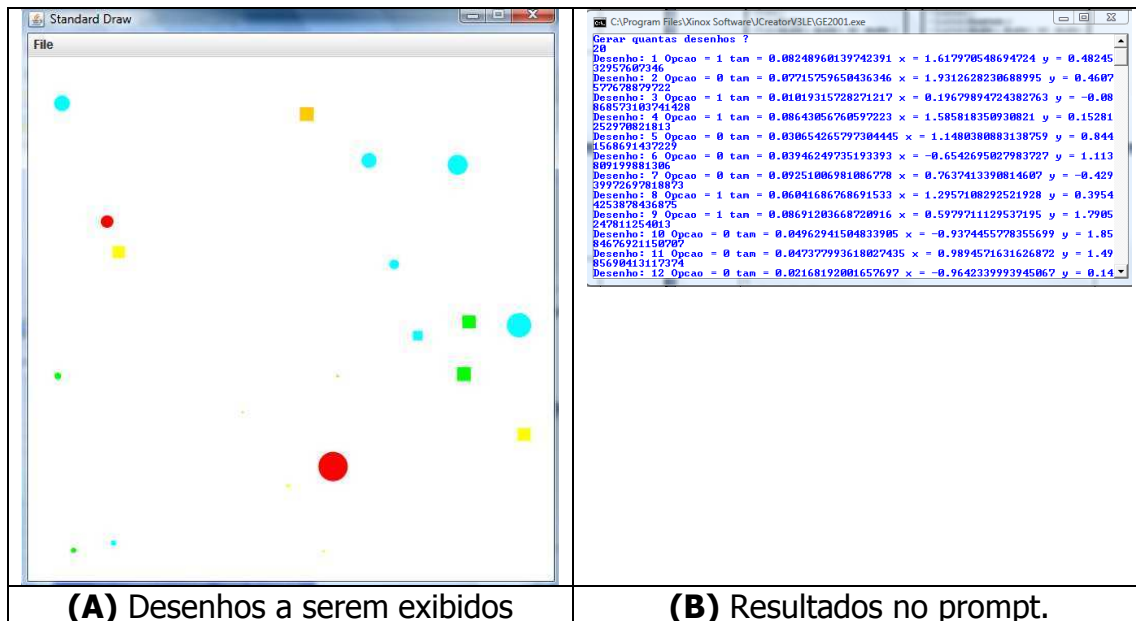


Figura 2.2: Resultados a serem apresentados pelo programa do **Exercício 2**.

```

public class Desenho
{
    private double x;
    private double y;
    private int cor;

    public Desenho()
    {
        setDesenho(0.0, 0.0, 0);
    }

    public Desenho(double a, double b, int i)
    {
        setDesenho(a, b, i);
    }

    public Desenho(Desenho p)
    {
        setDesenho(p.getX(), p.getY(), p.getCor());
    }

    public void setDesenho(double a, double b, int i)
    {
        setX(a);
        setY(b);
        setCor(i);
    }

    public void setX(double a)
    {
        x = a;
    }

    public void setY(double b)
    {
        y = b;
    }

    public void setCor(int i)
    {
        cor = (i >= 0) ? i : 0;
    }

    public double getX()
    {
        return x;
    }
}

```

```

public double getY()
{return y;}

public int getCor()
{return cor;}

public void desenhar()
{
    switch(cor)
    {
        case 0: StdDraw.setPenColor(StdDraw.RED);    break;
        case 1: StdDraw.setPenColor(StdDraw.GREEN);  break;
        case 2: StdDraw.setPenColor(StdDraw.YELLOW); break;
        case 3: StdDraw.setPenColor(StdDraw.CYAN);   break;
        case 4: StdDraw.setPenColor(StdDraw.ORANGE); break;
    }
    StdDraw.circle(x,y,0.01);
}
}

```

```

public class Ponto extends Desenho
{
    private double raio;

    public Ponto()
    {this(0.0,0.0,0,0.01);}

    public Ponto(Ponto p)
    {this(p.getX(),p.getY(),p.getCor(),p.getRaio());}

    public Ponto(double a, double b, int i, double c)
    {setPonto(a,b,i,c);}

    public void setPonto(double a, double b, int i, double c)
    {setDesenho(a,b,i);setRaio(c);}

    public void setRaio(double c)
    {raio = (c > 0)?c:0.01;}

    public double getRaio()
    {return raio;}

    public void desenhar()
    {
        switch(getCor())
        {
            case 0: StdDraw.setPenColor(StdDraw.RED);    break;
            case 1: StdDraw.setPenColor(StdDraw.GREEN);  break;
            case 2: StdDraw.setPenColor(StdDraw.YELLOW); break;

```

```

        case 3: StdDraw.setPenColor(StdDraw.CYAN);    break;
        case 4: StdDraw.setPenColor(StdDraw.ORANGE); break;
    }
    StdDraw.filledCircle(getX(),getY(),raio);
}
}

```

```

public class Quadrado extends Desenho
{
    private double lado;

    public Quadrado()
    {this(0.0,0.0,0,0.01);}

    public Quadrado(Quadrado p)
    {this(p.getX(),p.getY(), p.getCor(), p.getLado());}

    public Quadrado(double a, double b, int i, double c)
    {super(a,b,i);setLado(c);}
    public void setQuadrado(double a, double b, int i, double c)
    {setX(a);setY(b);setCor(i);setLado(c);}

    public void setLado(double c)
    {lado = (c > 0)?c:0.01;}

    public double getLado()
    {return lado;}

    public void desenhar()
    {
        switch(getCor())
        {
            case 0: StdDraw.setPenColor(StdDraw.RED);    break;
            case 1: StdDraw.setPenColor(StdDraw.GREEN);  break;
            case 2: StdDraw.setPenColor(StdDraw.YELLOW); break;
            case 3: StdDraw.setPenColor(StdDraw.CYAN);   break;
            case 4: StdDraw.setPenColor(StdDraw.ORANGE); break;
        }
        StdDraw.filledSquare(getX(),getY(),lado/2);
    }
}

```

```

import java.util.Scanner;
import java.util.Random;

```

```

public class TestaDesenho
{
    public static void main(String args[])
    {
        // Definindo a escala a ser empregada.
        double xmin = -1.0;
        double xmax = 2.0;
        double ymin = -1.0;
        double ymax = 2.0;
        double x, y;

        // Construindo os eixos da área para desenhar figuras.
        StdDraw.setXscale(xmin,xmax);
        StdDraw.setYscale(ymin,ymax);

        // Criando um vetor que pode armazenar
        // objetos da classe Desenho e das suas
        // subclasses.
        Desenho vm[];

        // Perguntar ao usuário o número de desenhos a serem geradas.
        Scanner in = new Scanner(System.in);
        System.out.println("Gerar quantos desenhos ?");
        int n = in.nextInt();

        // Alocação dinâmica de memória.
        vm = new Desenho[n];

        // Criando aleatoriamente de desenhos.
        Random r = new Random();
        int opcao, cor;
        double tam;

        // Laco para construcao aleatoria de objetos.
        for (int i=0; i < vm.length; i++)
        { // Gerando valores inteiros 0 ou 1 para
          // escolher o tipo de objeto de desenho.
          opcao = r.nextInt(2);
          // Gerando aleatoriamente o tamanho
          // do ponto: valor no intervalo [0.01,0.10].
          tam = r.nextDouble()*0.09 + 0.01;
          // Gerando aleatoriamente as coordenadas
          // x e y da imagem.
          x = r.nextDouble()*(xmax-xmin)+xmin;
          y = r.nextDouble()*(ymax-ymin)+ymin;
          // Definindo aleatoriamente a cor de cada desenho.
          cor = r.nextInt(5);

```



```

        System.out.println("Desenho: " + (i+1) + " Opcao = " + opcao + " tam
= " + tam + " x = " + x + " y = " + y);
        // Criando um objeto do tipo ponto.
        if (opcao == 0)
            vm[i] = new Ponto(x,y,cor,tam);
        // Criando um objeto do tipo quadrado.
        else
            vm[i] = new Quadrado(x,y,cor,tam);
    }
    // Laco para mostrar os objetos armazenados em vm.
    for (int i=0; i < vm.length; i++)
    {
        // Usando um metodo comum a todos os objetos,
        vm[i].desenhar();
    }
}
}

```

Perguntas importantes sobre o programa:

Item (A): Como alterar o tamanho mínimo e máximo dos desenhos?

Item (B): Como alterar o programa para fazer com que a probabilidade de ser criado um quadrado seja o dobro da probabilidade de se criar um ponto?

Item (C): Como são construídos os desenhos no Exercício 2? Qual o papel de cada classe?

Item (D): Qual a diferença entre a hierarquia de classes do Exercício 1 e do Exercício 2 em termos de campos?

Item (E): Como unificar a hierarquia de classes dos Exercícios 1 e 2 de modo a ter um programa em que seja possível ter partículas se movimentando em um ambiente com obstáculos (que são objetos Ponto e Quadrado)?

Exercício 3: Construir um novo programa que aproveita as classes do Exercício 2 e a partir da classe Ponto cria uma classe Particula. As classes Ponto, Quadrado e Particula serão empregadas em uma nova classe Jogo. Para testar a classe Jogo, é necessário construir a classe TestaJogo e o resultado deve ser semelhante ao mostrado na Figura 3.2. O relacionamento entre as classes Ponto e Particula é dado no diagrama UML da Figura 3.1. Observar que a classe Ponto é uma subclasse da classe Desenho (vide Figura 2.1).

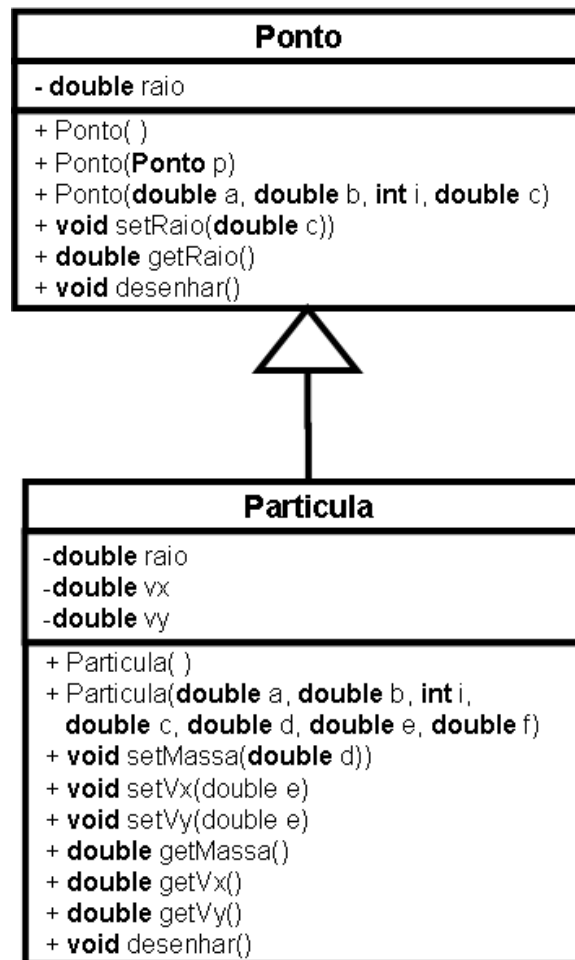


Figura 3.1: Relacionamento entre as classes **Ponto** e **Particula**.

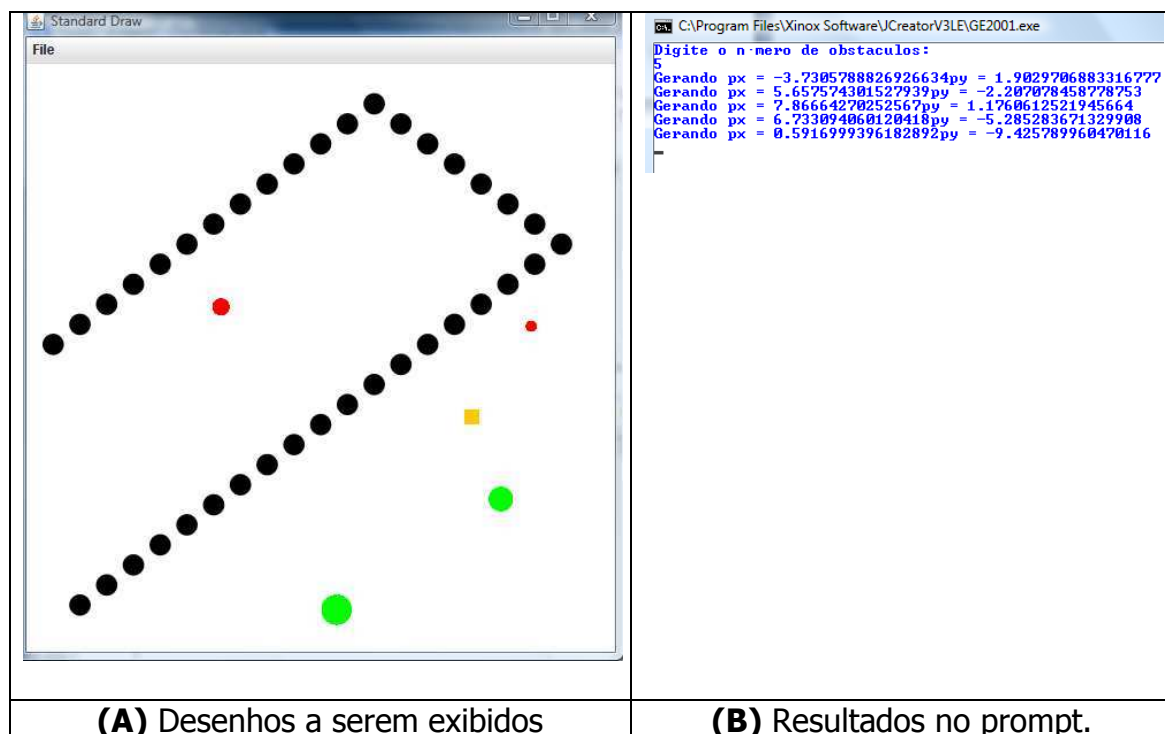


Figura 3.2: Resultados do programa com após a adição da classe **Particula**, **Jogo** e **TestaJogo**.

```

public class Particula extends Ponto
{
    private double massa;
    private double vx;
    private double vy;

    public Particula()
    {this(0.0,0.0,0,0.01,0.5,1,0.75);}

    public Particula(double a, double b, int i, double c, double d, double e,
double f)
    {super(a,b,i,c); setMassa(d); setVx(e); setVy(f);}

    public void setMassa(double d)
    {
        massa = (d > 0)?d:0;
        setRaio(Math.sqrt(massa/Math.PI));
    }

    public void setVx(double e)
    {vx = e;}

    public void setVy(double e)
    {vy = e;}

    public double getMassa()
    {return massa;}

    public double getVx()
    {return vx;}

    public double getVy()
    {return vy;}

    public void desenhar()
    {StdDraw.filledCircle(getX(),getY(),getRaio());}

    public void mover()
    {
        verificarLimite();
        setX(getX()+vx);
        setY(getY()+vy);
        desenhar();
    }

    public void verificarLimite()
    {
        double aux1 = getX()+vx+getRaio();

```

```

double aux2 = getY()+vy+getRaio();

if ((aux1 > 10.0)|| (aux1 < -10.0))
{vx = -vx;}
if ((aux2 > 10.0)|| (aux2 < -10.0))
{vy = -vy;}
}

public void verificarColisao(Particula p)
{
    double dx = Math.pow((p.getX()-(getX())),2);
    double dy = Math.pow((p.getY()-(getY())),2);
    double d = Math.sqrt(dx + dy);
    double r = p.getRaio() + getRaio();

    // Ocorreu uma colisao !!
    if (d <= r)
    {
        setVx(-getVx());
        setVy(-getVy());
        p.setVx(-getVx());
        p.setVy(-getVy());
    }
}

public static void main(String args[])
{
    StdDraw.setXscale(-10.0,10.0);
    StdDraw.setYscale(-10.0,10.0);
    Particula p1 = new Particula(-10.0,-10.0,0,0.01,4.5,1,0.75);
    Particula p2 = new Particula(0.0,10.0,0,0.01,1.5,0.75,1);
    p1.desenhar();
    while(true)
    {
        StdDraw.clear();
        StdDraw.setPenColor(StdDraw.RED);
        p1.mover();
        StdDraw.setPenColor(StdDraw.BLUE);
        p2.mover();
        p1.verificarColisao(p2);
        StdDraw.show(20);
    }
}
}

```

```

import java.util.Scanner;
import java.util.Random;

public class Jogo
{
    private Desenho vd[];
    private Particula p;
    private double xmin, xmax, ymin, ymax;

    public Jogo()
    {
        System.out.println("Digite o número de obstaculos:");
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        vd = new Desenho[n];
        xmin = -10.0;
        xmax = 10.0;
        ymin = xmin;
        ymax = xmax;
        gerarObstaculos(n);
        p = new Particula(-10.0,-10.0,0,0.01,0.5,1,0.75);
    }

    public void gerarObstaculos(int n)
    {
        Random r = new Random();
        int opcao, cor;
        double px, py, dm;

        for (int i=0; i < n; i++)
        {
            opcao = r.nextInt(2);
            px = (r.nextDouble()*(xmax-xmin))+xmin;
            py = (r.nextDouble()*(ymax-ymin))+ymin;
            dm = r.nextDouble()*(0.45)+0.10;
            cor = r.nextInt(5);
            System.out.println("Gerando px = "+px+"py = "+py);

            // Escolha aleatoria dos tipos de obstaculos.
            if (1 == opcao)
                vd[i] = new Ponto(px,py,cor,dm);
            else
                vd[i] = new Quadrado(px,py,cor,dm);
        }
    }
}

```

```

public void verificarColisao()
{
    double dx = 0.0;
    double dy = 0.0;
    double d = 0.0;
    double r = 0.0;
    Ponto aux1;
    Quadrado aux2;

    for (int i=0; i < vd.length; i++)
    {
        dx = Math.pow((vd[i].getX()-(p.getX())),2);
        dy = Math.pow((vd[i].getY()-(p.getY())),2);
        d = Math.sqrt(dx + dy);
        // Verificando se o objeto contido em v[i] é de uma dada classe.
        // Se for, então, usa-se a conversão para uma variável apropriada
        // a fim de se usar o método específico daquele objeto.
        if (vd[i] instanceof Ponto)
        {
            aux1 = (Ponto) vd[i];
            r = aux1.getRaio() + p.getRaio();
        }
        else
        {
            aux2 = (Quadrado) vd[i];
            r = aux2.getLado() + p.getRaio();
        }

        // Ocorreu uma colisao !!
        if (d <= r)
        {
            p.setVx(-p.getVx());
            p.setVy(-p.getVy());
            break;           // podemos terminar o laço de verificação da colisao.
        }
    }
}

public void verificarLimite()
{
    double aux1 = p.getX()+p.getVx()+p.getRaio();
    double aux2 = p.getY()+p.getVy()+p.getRaio();

    if ((aux1 > 10.0)|| (aux1 < -10.0))
    {p.setVx(-p.getVx());}
    if ((aux2 > 10.0)|| (aux2 < -10.0))
    {p.setVy(-p.getVy());}
}

```

```

public void mover()
{
    verificarLimite();
    verificarColisao();
    p.setX(p.getX()+p.getVx());
    p.setY(p.getY()+p.getVy());
    StdDraw.setPenColor(StdDraw.BLACK);
    p.desenhar();
    Random r = new Random();
    int cor;

    for (int i=0; i < vd.length; i++)
        vd[i].desenhar();

}

public void jogar()
{
    StdDraw.setXscale(xmin,xmax);
    StdDraw.setYscale(ymin,ymax);

    while(true)
    {
        StdDraw.clear();
        mover();
        StdDraw.show(20);
    }
}

```

```

public class TestaJogo
{
    public static void main(String args[])
    {
        Jogo j1 = new Jogo();
        j1.jogar();
    }
}

```

Perguntas Importantes sobre o programa:

Item (A): Como alterar a cor da Partícula quando a mesma colide com uma parede? E com um obstáculo?

Item (B): Como alterar a massa (conseqüentemente o tamanho) da Partícula quando a mesma colide com uma parede? E com um obstáculo?

Item (C): Quais as funcionalidades são adicionadas pela classe Particula à classe Ponto?

Item (D): Como funciona a classe Jogo?

Item (E): Como funciona a classe TestaJogo? Como são criados os obstáculos e a partícula?

Item (F): Qual o papel do método jogar da classe Jogo? E do método mover da classe Jogo?

Exercício 1: Construir um programa capaz de capturar valores de coordenadas Y e depois desenhar na Tela um gráfico, tal como dado na Figura 1.1.

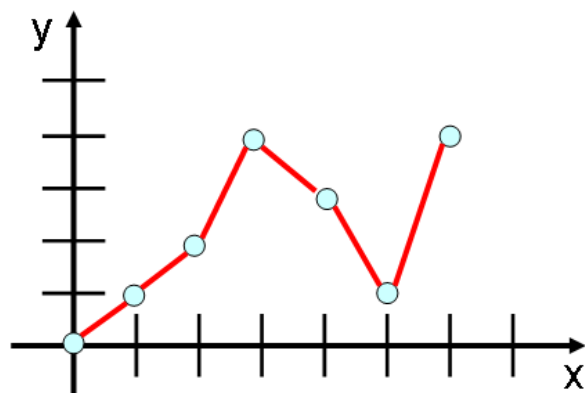


Figura 1.1: Construção de um gráfico com valores de Y por X.

Observe que neste gráfico os valores das coordenadas (x, y) são tais que os valores de X se sucedem por meio da seguinte fórmula: $x_{i+1} = x_i + 1$ e $x_0 = 0$, mas os valores de Y são fornecidos pelo usuário até que a letra s seja digitada. Para construir o gráfico você deverá utilizar a biblioteca **StdDraw**, disponível em: <http://www.cs.princeton.edu/introcs/stdlib/>.

Item (A): Construir o programa, sem considerar o uso de Exceções. O que ocorre quando o usuário digita um caractere que não é um valor inteiro?

```
import java.util.Scanner;

public class Sequencia
{
    private int elem[];

    // Construtor sem parâmetros.
    public Sequencia()
    {
        System.out.println("Digite n: ");
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        preencher(n);
    }
}
```

```
// Construtor com parâmetros.
public Sequencia(int n)
{
    preencher(n);
}

public void preencher(int n)
{
    // Criando n espaços do tipo int para elem.
    elem = new int[n];

    // Criando um objeto da classe Scanner.
    Scanner in = new Scanner(System.in);
    int a = 1;
    int b = 1;

    // Inicializando os componentes de elem.
    for (int i=0; i < n; i++)
    {
        System.out.println("Digite a:");
        a = in.nextInt();
        System.out.println("Digite b:");
        b = in.nextInt();
        elem[i] = a/b;
    }
}

// Metodo para alterar um elemento de elem.
public void setElem(int i, int aux)
{
    elem[i] = aux;
}

// Metodo para retornar um elemento de elem.
public int getElem(int i)
{
    return elem[i];
}

// Metodo para retornar o menor elemento de elem.
public int getMinElem()
{
    // Menor valor provisorio.
    int aux = elem[0];

    for(int i=0; i < elem.length; i++)
        if (elem[i] < aux)
            aux = elem[i];
    return aux;
}
```

// Metodo para retornar o maior elemento de Sequencia.

```
public int getMaxElem()
{
    // Maior valor provisório.
    int aux = elem[0];

    for(int i=0; i < elem.length; i++)
    {
        if (elem[i] > aux)
            aux = elem[i];
    }

    return aux;
}

public void graf()
{
    // Definindo a escala do grafico atraves de
    // metodos que retornam os menores e os maiores
    // valores em x e y.
    int xmin = 0;
    int xmax = elem.length;
    int ymin = getMinElem();
    int ymax = getMaxElem();
    StdDraw.setXscale(xmin, xmax);
    StdDraw.setYscale(ymin, ymax);
    StdDraw.clear();

    // Laco para desenhar
    for(int i=1; i < elem.length; i++)
    {
        StdDraw.setPenColor(StdDraw.BLUE);
        StdDraw.filledCircle(i, elem[i], 0.05);
        StdDraw.setPenColor(StdDraw.RED);
        StdDraw.line(i-1, elem[i-1], i, elem[i]);
    }
}
```

```
public class TestaSequencia
{
    public static void main(String args[])
    {
        Sequencia s1 = new Sequencia();
        s1.graf();
    }
}
```

Item (B): Construir o programa de modo a considerar o uso de Tratamento de Exceções. Para tanto, as exceções descritas na Tabela 1.1 deverão ser empregadas em conjunto com o uso dos comandos **try/catch**. Observe que para definir o tamanho do vetor o usuário poderia tentar inserir letras e valores reais ao invés de números inteiros. O tratamento de Exceções pode ser empregado para resolver este problema se for realizada a leitura de valores como String (nextLine) com posterior conversão de valores de **String** para **int** no comando "**y = Integer.parseInt(yStr);**". Depois, é necessário inserir um bloco **try-catch** tal que o bloco **catch** trate da Exceção **NumberFormatException**. O que ocorre agora quando o usuário digita uma letra? Teste as outras possibilidades de erros de modo a empregar e verificar as possibilidades fornecidas na Tabela 1.1.

Exceção (Classe)	Tratamento realizado
NumberFormatException	Previne contra erros de conversão.
NegativeArraySizeException	Previne contra erros de índice negativo de vetores.
ArithmeticException	Previne contra erros de divisão de números inteiros por zero.
ArrayIndexOutOfBoundsException	Previne contra índice do vetor fora dos limites adequados (0 até vetor.length).

Tabela 1.1: Classes para realizar o tratamento de exceção adequado para cada problema que pode ocorrer durante a execução do programa.

```
import java.util.Scanner;

public class Sequencia2
{
    private int elem[];

    // Construtor sem parâmetros.
    public Sequencia2()
    {
        System.out.println("Digite n: ");
        Scanner in = new Scanner(System.in);
        // Previne contra erros de dados de entrada.
        String aux = in.nextLine();
        int n = 3;
        // Tenta transformar em numero inteiro, se for possivel.
        try
        {
            n = Integer.parseInt(aux);
        }
        catch (NumberFormatException e)
        {
            System.out.println("Erro de leitura: " + e.getMessage());
            System.out.println("Assumindo n padrao: " + n);
        }
        preencher(n);
    }
}
```

```

// Construtor com parâmetros.
public Sequencia2(int n)
{
    preencher(n);
}

public void preencher(int n)
{
    // Criando n espacos do tipo int para elem.
    try
    {
        elem = new int[n];
    }
    catch( NegativeArraySizeException e )
    {
        n = 3;
        System.out.println("Erro de tamanho de elem: " + e.getMessage());
        System.out.println("Assumindo n padrao: " + n);
        elem = new int[n];
    }
}

// Criando um objeto da classe Scanner.
Scanner in = new Scanner(System.in);
int a = 1;
int b = 1;

// Inicializando os componentes de elem.
for (int i=0; i < n; i++)
{
    System.out.println("Digite a:");
    a = in.nextInt();
    System.out.println("Digite b:");
    b = in.nextInt();
    // Previne contra erros de divisao por zero.
    try
    {
        elem[i] = a/b;
    }
    catch (ArithmeticException e)
    {
        System.out.println("Erro aritmetico: " + e.getMessage());
        System.out.println("Assumindo elem = 0");
        elem[i] = 0;
    }
}
}

```

```

// Metodo para alterar um elemento de elem.
public void setElem(int i, int aux)
{
    // Previne contra erro de indices fora dos limites do vetor.
    try
    {
        elem[i] = aux;
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
        System.out.println("Indice fora dos limites: " + e.getMessage());
    }
}

public int getElem(int i)
{
    // Previne contra erro de indices fora dos limites do vetor.
    try
    {
        return elem[i];
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
        System.out.println("Indice fora dos limites: " + e.getMessage());
        return 0;
    }
}

// Metodo para retornar o menor elemento de Sequencia.
public int getMinElem()
{
    // Menor valor provisorio.
    int aux = elem[0];
    for(int i=0; i < elem.length; i++)
        if (elem[i] < aux)
            aux = elem[i];
    return aux;
}

// Metodo para retornar o maior elemento de Sequencia.
public int getMaxElem()
{
    // Maior valor provisorio.
    int aux = elem[0];
    for(int i=0; i < elem.length; i++)
        if (elem[i] > aux)
            aux = elem[i];
    return aux;
}

```

```

public void graf()
{
    // Definindo a escala do grafico atraves de
    // metodos que retornam os menores e os maiores
    // valores em x e y.
    int xmin = 0;
    int xmax = elem.length;
    int ymin = getMinElem();
    int ymax = getMaxElem();
    StdDraw.setXscale(xmin, xmax);
    StdDraw.setYscale(ymin, ymax);
    StdDraw.clear();

    // Laco para desenhar
    for(int i=1; i < elem.length; i++)
    {
        StdDraw.setPenColor(StdDraw.BLUE);
        StdDraw.filledCircle(i, elem[i], 0.05);
        StdDraw.setPenColor(StdDraw.RED);
        StdDraw.line(i-1, elem[i-1], i, elem[i]);
    }
}

```

Item (C): Reconstruir o programa de modo que em um primeiro momento os valores reais de X e Y sejam armazenados em dois vetores. Após armazenar os valores de X e Y em dois vetores, é necessário empregar um método em que os índices deste vetor serão utilizados para construir os gráficos. Para tanto, este método deve empregar um bloco **try-catch**, cujo **catch** captura o tratamento da Exceção **ArrayIndexOutOfBoundsException**.

```

import java.util.Scanner;

public class Sequencia3
{
    private double x[];
    private double y[];

    // Construtor sem parâmetros.
    public Sequencia3()
    {
        System.out.println("Digite n: ");
        Scanner in = new Scanner(System.in);
        // Previne contra erros de dados de entrada.
        String aux = in.nextLine();
        int n = 3;

        // Tenta transformar em numero inteiro, se for possivel.

```

```

try
{
    n = Integer.parseInt(aux);
}
catch (NumberFormatException e)
{
    System.out.println("Erro de leitura: " + e.getMessage());
    System.out.println("Assumindo n padrao: " + n);
}
preencher(n);
}

// Construtor com parâmetros.
public Sequencia3(int n)
{
    preencher(n);
}

public void preencher(int n)
{
    // Criando n espacos do tipo int para elem.
    try
    {
        x = new double[n];
        y = new double[n];
    }
    catch( NegativeArraySizeException e )
    {
        n = 3;
        System.out.println("Erro de tamanho de elem: " + e.getMessage());
        System.out.println("Assumindo n padrao: " + n);
        x = new double[n];
        y = new double[n];
    }

    // Criando um objeto da classe Scanner.
    Scanner in = new Scanner(System.in);
    String a;

    // Inicializando os componentes de elem.
    for (int i=0; i < n; i++)
    {
        System.out.println("Digite x:");
        a = in.nextLine();

        // Previne contra erros de leitura e conversao.

```

```

try
{
    x[i] = Integer.parseInt(a);
}
catch (NumberFormatException e)
{
    System.out.println("Erro de conversao: " + e.getMessage());
    System.out.println("Assumindo x = 0");
    x[i] = 0;
}

System.out.println("Digite y:");
a = in.nextLine();
// Previne contra erros de leitura e conversao.
try
{
    y[i] = Integer.parseInt(a);
}
catch (NumberFormatException e)
{
    System.out.println("Erro de conversao: " + e.getMessage());
    System.out.println("Assumindo y = 0");
    y[i] = 0;
}
}

// Metodo para alterar um elemento de Y.
public void setElem(int i, int aux)
{
    // Previne contra erro de indices fora dos limites do vetor.
    try
    {
        y[i] = aux;
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
        System.out.println("Indice fora dos limites: " + e.getMessage());
    }
}

// Metodo para retornar um elemento de Y.

```

```

public double getElem(int i)
{
    // Previne contra erro de indices fora dos limites do vetor.
    try
    {
        return y[i];
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
        System.out.println("Indice fora dos limites: " + e.getMessage());
        return 0;
    }
}

// Metodo para retornar o menor elemento de x.
public double getXMinElem()
{
    // Menor valor provisorio.
    double aux = x[0];

    for(int i=0; i < x.length; i++)
    {
        if (x[i] < aux)
            aux = x[i];
    }

    return aux;
}

// Metodo para retornar o maior elemento de x.
public double getXMaxElem()
{
    // Maior valor provisorio.
    double aux = x[0];

    for(int i=0; i < x.length; i++)
    {
        if (x[i] > aux)
            aux = x[i];
    }

    return aux;
}

// Metodo para retornar o menor elemento de y.

```

```

public double getYMinElem()
{
    // Menor valor provisorio.
    double aux = y[0];
    for(int i=0; i < y.length; i++)
        if (y[i] < aux)
            aux = y[i];
    return aux;
}

// Metodo para retornar o maior elemento de y.
public double getYMaxElem()
{
    // Maior valor provisorio.
    double aux = y[0];
    for(int i=0; i < y.length; i++)
        if (y[i] > aux)
            aux = y[i];
    return aux;
}

public void graf()
{
    // Definindo a escala do grafico atraves de metodos que retornam os
    // menores e os maiores valores em x e y.
    double xmin = getXMinElem();
    double xmax = getXMaxElem();
    double ymin = getYMinElem();
    double ymax = getYMaxElem();
    StdDraw.setXscale(xmin, xmax);
    StdDraw.setYscale(ymin, ymax);
    StdDraw.clear();

    // Laco para desenhar
    for(int i=1; i < x.length; i++)
    {
        StdDraw.setPenColor(StdDraw.BLUE);
        StdDraw.filledCircle(x[i], y[i], 0.05);
        StdDraw.setPenColor(StdDraw.RED);
        StdDraw.line(x[i-1], y[i-1], x[i], y[i]);
    }
}

```

Não esquecer de também alterar a classe TestaSequencia.

Exercício 2: Todo objeto da classe Exception possui o método `printStackTrace()`. Com este método é possível identificar a linha de código

do Programa que lançou uma dada Exceção. Modificar o tratamento de Exceção do Programa apresentado no **Exercício 1** de modo a empregar o método `printStackTrace()`. Um exemplo de utilização do método é dado na Figura 2.1.

```

System.err.println(e + "\n");
e.printStackTrace();

```

Figura 2.1: Uso do método `printStackTrace()`.

Exercício 3: O esquema do funcionamento de uma usina nuclear do tipo PWR é como dado na Figura 3.1.

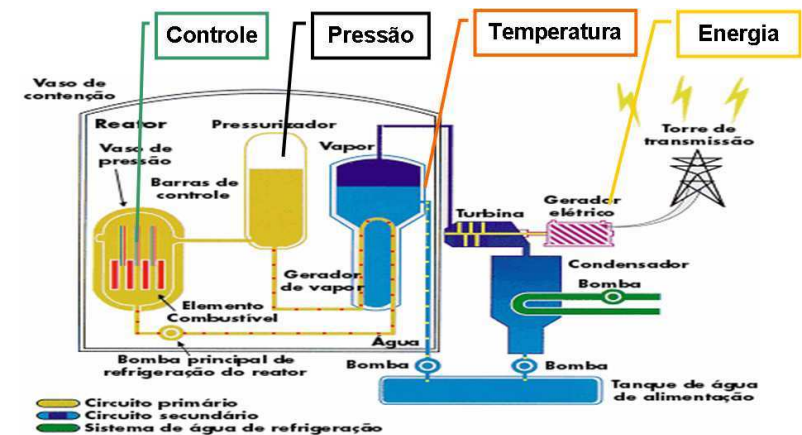


Figura 3.1: Diagrama do funcionamento de uma usina nuclear PWR.

Criar um programa que simula o funcionamento e o controle de uma usina PWR através apenas do ajuste da pressão. Supor que:

- (i) A pressão $x(t)$ é um valor inteiro que pode variar de $[0, 30]$ no tempo t .
- (ii) O tempo t é uma variável inteira e cada instante de tempo é calculada a energia gerada pela usina.
- (iii) A energia $y(t)$ gerada pela usina em cada instante de tempo depende do valor da pressão de acordo com a seguinte equação: $y(t) = -2x(t)^2 + 80x(t) + 3600$.
- (iv) O usuário pode reduzir ou aumentar a pressão de acordo com um valor inteiro contido no intervalo $[1, 10]$.

(v) Cada vez que o valor da pressão $x(t)$ é modificado existe uma componente aleatória cujo valor está contido no intervalo $[1, 5]$ que é adicionada (se o

usuário manda aumentar $x(t)$) ou retirada (se o usuário manda reduzir a pressão) de $x(t)$.

Construir um programa que simule o funcionamento da usina, não permitindo que $x(t)$ assuma valores negativos ou maiores que 30 durante a simulação (se isto ocorrer, então, será considerado que a usina gerou zero de energia). Os valores de geração de energia deverão ser armazenados em um vetor e posteriormente os mesmos deverão ser desenhados em um gráfico tal como descrito no **Exercício 1**. Não se esqueça de empregar os conceitos de Tratamento de Exceção no desenvolvimento do seu programa.

Exercício 4: Uma sequência de números inteiros é gerada aleatoriamente e armazenada em um vetor y . A sequência de números é gerada de acordo com a fórmula: $y[i] = y[i-1] + x$ e $y[0] = 2$, onde x é um número inteiro aleatório contido no intervalo $[0, 5]$. Para gerar número aleatórios empregue o código da Figura 4.1.

```
import java.util.Random;

// Cria um objeto da classe Random que funciona como gerador aleatório.
Random randomNumbers = new Random();
// Gera valores aleatorios inteiros: valores {{0},{1}}.
aleat = randomNumbers.nextInt(2);
// Valores reais contidos no intervalo [20,50].
preco = 30*randomNumbers.nextDouble() + 20;
```

Figura 4.1: Gerando números aleatórios inteiros ou reais.

O programa deve ter três etapas:

- (i) Escolha do número de valores aleatórios e geração dos números aleatórios.
- (ii) Escolha dos pontos y a serem visualizados pelo usuário.
- (iii) Construir um gráfico com os valores contidos em y .

Não esquecer de empregar um bloco **try-catch** para o passo (ii) (**ArrayIndexOutOfBoundsException**).

Exercício 5: Modificar o Exercício 4 de modo que y seja um vetor de valores do tipo **double** e calculado por: $y[i] = x + 6/x$ e $y[0] = 2$, onde x é um número inteiro aleatório contido no intervalo $[0, 6]$. Neste programa, o tratamento de Exceções deve contemplar a **ArithmeticException**.

Exercício 6: Modificar o **Exercício 5** de modo que o usuário pode inserir o número x tal que $y[i] = x + 6/x$ e $y[0] = 2$. Além disso, o usuário pode escolher o índice i para o qual deseja armazenar o valor calculado. É útil empregar **NumberFormatException** e **ArrayIndexOutOfBoundsException** e um bloco **try** com múltiplos **catch**.

Exercício 7: O comando Além das exceções já previstas na linguagem Java, pode-se gerar novas exceções sempre que necessário. Neste caso, é possível construir uma classe tratadora de exceção que deve ser uma subclasse da classe **Exception**. Assim, é possível considerar uma situação para a qual não está prevista uma exceção pela linguagem Java. Para gerar uma exceção, deve-se usar a instrução **throw** tal como dado no programa a seguir. Descubra o resultado a ser mostrado pelo programa e justifique.

Classe Misterio

```
public class Misterio
{
    public static void main(String args[])
    {
        try
        {
            facaPingPong(3);
            facaPong(2);
            facaAlgo(1);
            facaPing(0);
        }
        catch (Ping e)
        {
            System.out.println("Excecao ping: " + e + "\n");
        }
        catch (Pong e)
        {
            System.out.println("Excecao pong: " + e + "\n");
        }
        catch (PingPong e)
        {
            System.out.println("Excecao pingpong: " + e + "\n");
        }
    }

    static void facaPing(int valor) throws Ping
    {
        if (0 == valor)
            throw new Ping(valor);
        else
            System.out.println("Valor correto = "+valor+" \n");
    }
}
```

```

static void facaAlgo(int valor) throws PingPong
{
    if (0 == valor)
        throw new PingPong(valor);
    else if (1 == valor)
        System.out.println("Valor correto = "+valor+" \n");
    else if (2 == valor)
        throw new PingPong(valor);
    else
        System.out.println("Nem Ping, nem Pong, Valor = "+valor+" \n");
}

static void facaPong(int valor) throws Pong
{
    if (1 == valor)
        System.out.println("Valor correto = "+valor+" \n");
    else
        throw new Pong(valor);
}

static void facaPingPong(int valor) throws PingPong
{
    if (0 == valor)
        throw new PingPong(valor);
    else
        System.out.println("Nem Ping, nem Pong, Valor = "+valor+" \n");
}
}

```

Classe Ping

```

public class Ping extends Exception
{
    int valor;

    public Ping(int v)
    {
        valor = v;
    }

    public String toString()
    {
        return "\n Ping " + valor;
    }
}

```

Classe Pong

```

public class Pong extends Exception
{
    int valor;

    public Pong(int v)
    {
        valor = v;
    }

    public String toString()
    {
        return "\n Pong " + valor;
    }
}

```

Classe PingPong

```

public class PingPong extends Exception
{
    int valor;

    public PingPong(int v)
    {
        valor = v;
    }

    public String toString()
    {
        return "\n PingPong " + valor;
    }
}

```


Exercício 1: Dado o diagrama UML da Figura 1.1, construir um programa capaz de simular o funcionamento de folha de pagamento com quatro classes de trabalhadores: *Empregado*, *PorHora*, *PorComissao* e *PorHoeComissao*. A classe *Empregado* deve ser abstrata, pois o método *getPay()*, que retorna o quanto cada tipo de empregado deve ganhar, só poderá ser definido nas subclasses. Desse modo, a classe *Empregado* deve ser declarada abstrata. Para todas as classes cujo ganho dos trabalhadores está relacionado com a comissão relativa ao montante de vendas (*PorComissao* e *PorHoeComissao*), deve-se empregar o método *setVendas* e a informação contida no campo *COMMISSION_RATE*. Por último, a classe *FolhadePagamento* emprega objetos de todas as classes. Uma visão geral do programa é dada no diagrama UML da Figura 1.1.

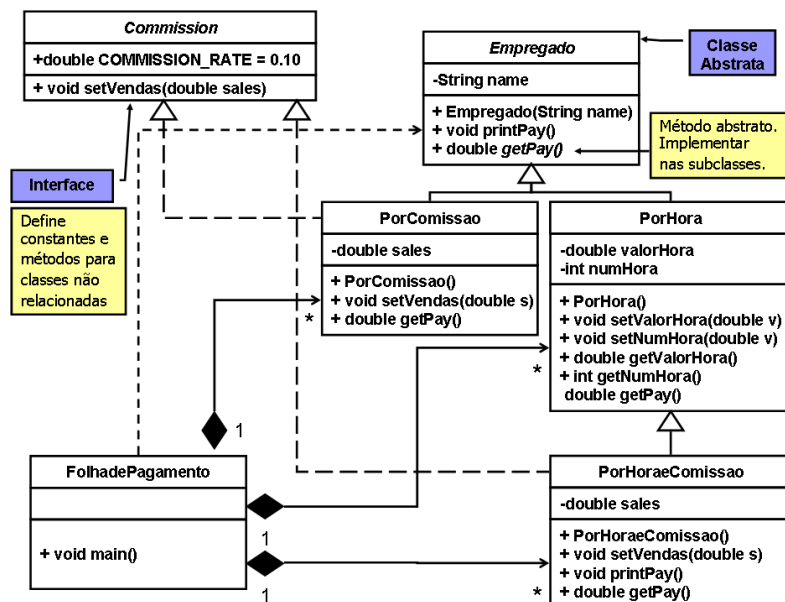


Figura 1.1: Diagrama UML das classes para construir a folha de pagamento.

O detalhamento de como implementar o diagrama UML mostrado na Figura 1.1 é como se segue.

Interface Commission

```
// Esta interface especifica um atributo
// comum e declara um comportamento comum
// a todos os trabalhadores comissionados.
public interface Commission
{
    double COMMISSION_RATE = 0.10; // Taxa Constante.
    void setVendas(double vendas);
} // Fim Interface Commission.
```

Classe Abstrata Empregado

```
// Esta classe abstrata descreve os empregados.
public abstract class Empregado
{
    private String name;

    public Empregado(String name)
    {
        this.name = name;
    }

    public void printPay()
    {
        System.out.printf("%10s: ganha %8.2f \n", name, getPay());
    }

    // Metodo abstrato: o retorno deste
    // metodo sera especificado nas subclasses.
    public abstract double getPay();
} // Fim classe abstrata Empregado.
```

Classe PorComissao

```
// Esta classe representa empregados que apenas ganham por comissao.
public class PorComissao extends Empregado implements Commission
{
    private double sales = 0.0;
    /**
     * CONSTRUTORES
     */
    public PorComissao() { this("Sem Nome", 0.0); }

    public PorComissao(String name) { this(name, 0.0); }
```

```

public PorComissao(String name, double sales)
{
    super(name);
    setVendas(sales);
} // Fim construtor com 2 parametros.

/*****/

public void setVendas(double sales)
{
    this.sales = (sales < 0)?0.0:sales;
}

// Definindo o metodo abstrato.
public double getPay()
{
    double pay = COMMISSION_RATE * sales;
    return pay;
}

} // Fim classe PorComissao.

```

Classe PorHora

```

public class PorHora extends Empregado
{
    private double valorHora = 0.0;
    private int numHora = 0;

    /*****/
    // CONSTRUTORES
    /*****/
    public PorHora() { this("Sem Nome",0.0,0); }

    public PorHora(String name, double vhora) { this(name,vhora,0); }

    public PorHora(String name, double vhora, int nhora)
    {
        super(name);
        setValorHora(vhora);
        setNumHora(nhora);
    }

    /*****/
    public void setValorHora(double vhora)
    { this.valorHora = (vhora < 0.0)?0.0:vhora; }

    public void setNumHora(int nhora)

```

```

{ this.numHora = (nhora < 0)?0:nhora; }

public double getValorHora() {return valorHora;}

public int getNumHora() {return numHora; }

// Definindo o metodo abstrato.
public double getPay()
{
    double pay = valorHora * numHora;
    return pay;
}
} // Fim classe PorHora.

```

Classe PorHraeComissao

```

public final class PorHraeComissao extends PorHora implements
Commission
{
    private double sales = 0.0;

    /*****/
    // CONSTRUTORES
    /*****/
    public PorHraeComissao() { this("Sem Nome",0.0,0,0.0); }

    public PorHraeComissao(String name, double vhora, int nhora)
    { this(name,vhora,nhora,0.0); }

    public PorHraeComissao(String name, double vhora, int nhora, double s)
    {
        super(name,vhora,nhora);
        setVendas(s);
    }

    /*****/

    public void setVendas(double s) { sales = (s < 0.0)?0.0:s; }

    // Redefinindo o metodo herdado de PorHora.
    public double getPay()
    {
        double pay = getValorHora() * getNumHora() + COMMISSION_RATE * sales;
        return pay;
    }

    // Redefinindo o metodo herdado de PorHora.

```

```

public void printPay()
{
    super.printPay();
    // Detalhando o calculo dos ganhos.
    System.out.printf("Ganho por hora      : %8.2f \n",getValorHora() *
getNumHora());
    System.out.printf("Ganho por comissao: %8.2f \n",COMMISSION_RATE
* sales);
}
} // Fim da classe PorHoraComissao.

```

Classe FolhadePagamento

```

public class FolhadePagamento
{
    public static void main(String args[])
    {
        PorHora h1;
        Empregado listapag[] = new Empregado[10];
        listapag[0] = new PorComissao("Joao");
        listapag[1] = new PorHora("Jose",30.0);
        listapag[2] = new PorHoraComissao("Maria",10.0,200);

        ((Commission) listapag[0]).setVendas(25000);
        ((Commission) listapag[2]).setVendas(10000);

        // Laco para adicionar horas, se necessario.
        // Para todos os elementos, imprimir os dados.
        for (int i=0; i < listapag.length && listapag[i] != null; i++)
        {
            if (listapag[i] instanceof PorHora)
            {
                h1 = (PorHora) listapag[i];
                h1.setNumHora(200);
            }

            listapag[i].printPay();
        } // Fim laço.
        // Fim main.
    } // Fim classe FolhadePagamento
}

```

Exercício 2: Baseado no **Exercício 1** simular a operação de um cadastro de imóveis. Para tanto, empregar como referência o diagrama UML da Figura 2.1 e as informações contidas na Tabela 2.1.

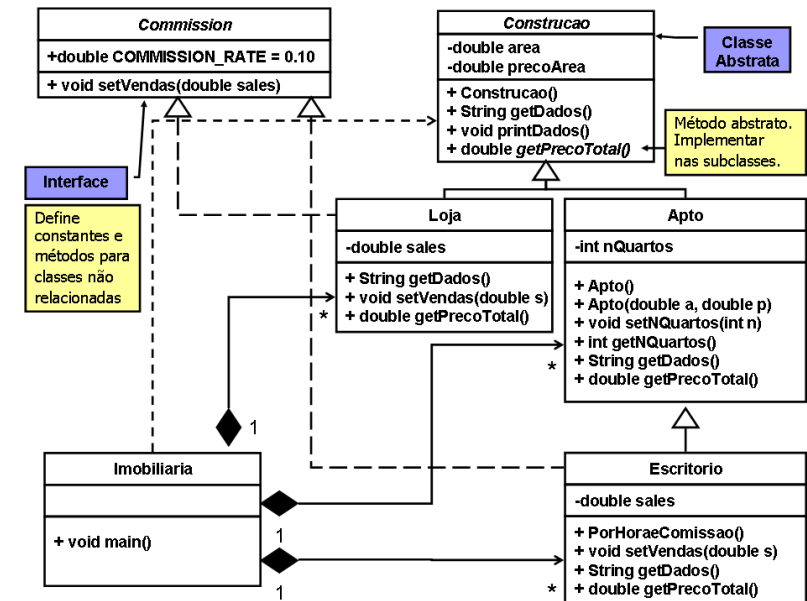


Figura 2.1: Diagrama UML das classes para construir cadastro de imóveis.

Item (A): Construir as classes e os métodos descritos na **Figura 2.1** cujo funcionamento é detalhado na **Tabela 2.1**.

	getPrecoTotal()	getDados()	printDados()
Construcão	Retornar o valor contido no campo area (área total do imóvel em m ²) multiplicado pelo valor contido no campo precoArea (preço por m ²).	Retorna uma String contendo valores dos campos da classe.	Imprimir o conteúdo dos campos das classes, empregando o método getDados() e o valor total fornecido por getPrecoTotal() .
Loja	Retornar o mesmo valor do método da superclasse acrescido da taxa contida na classe Commission sobre o valor contido no campo de cada instância.	Retorna uma String contendo valores dos campos da classe.	Emprega a implementação da superclasse.

Apto	Retornar o mesmo valor do método da superclasse multiplicado pelo número de quartos.	Retorna uma String contendo valores dos campos da classe.	Emprega a implementação da superclasse.
Escritorio	Combinar o retorno oferecido pela classe Loja que é multiplicado pelo retorno oferecido pela classe Apto.	Retorna uma String contendo valores dos campos da classe.	Mostrar os dados da classe e também a parcela de ganho advinda da classe Loja e a parcela de ganho advinda da classe Apto.

Tabela 2.1: Comportamento de cada método para cada classe.

Elaborar também os construtores **com** e **sem** parâmetros de todas as classes.

Item (B): Gerar 6 objetos da classe **Loja** ou **Apto** ou **Escritorio**. A escolha de geração dos objetos **Loja** ou **Apto** ou **Escritorio** é realizada de modo aleatório, bem como a atribuição de valores das variáveis de instância dos objetos **Loja** ou **Apto** ou **Escritorio** é realizada aleatoriamente. Para tanto, será necessário empregar o pacote **import java.util.Random**, bem como os comandos dados na **Figura 2.2**. Após gerar um objeto para o vetor, se o objeto for da classe **Loja** ou **Escritorio**, então, o método **setVendas** deverá ser chamado (pode ser usado o comando **instanceof** para definir se um objeto contido no vetor é de uma dada classe). Se for da classe **Apto**, então, chamar o método **setNQuartos**. Imprimir todos os objetos gerados aleatoriamente contidos no vetor.

```
// Cria um objeto da classe Random que funciona como gerador aleatório.
Random randomNumbers = new Random();
// Gera valores aleatorios inteiros: valores {{0},{1}}.
aleat = randomNumbers.nextInt(2);
// Valores reais contidos no intervalo [20,50].
valor = 30*randomNumbers.nextDouble() + 20;
```

Figura 2: Gerando números aleatórios inteiros ou reais.

Exercício 3: Dado o diagrama UML da Figura 3.1, construir um programa capaz de simular o funcionamento de um jogo com um personagem que sabe golpes de Karatê. Para tanto, existem 2 interfaces: **Imagem** e **Som**. A Interface **Imagem** define parâmetros e um método para desenhar na tela figuras armazenadas em arquivos com extensão **.gif**. A Interface **Som** é responsável por definir o método a ser empregado para se tocar o som contido em arquivos com extensão **.wav**. Para poder mostrar imagens e tocar sons será necessário incorporar ao seu projeto as classes **StdDraw** e **StdAudio**, respectivamente (Estas classes se encontram nos endereços: <http://www.cs.princeton.edu/introcs/15inout/StdDraw.java.html> e <http://www.cs.princeton.edu/introcs/15inout/StdAudio.java.html>). Além disso, você vai precisar baixar do site do curso os arquivos: "Karate-01.gif", "Karate-04.gif", "Castelo1.gif" e "oizuki.wav". Todos estes arquivos devem estar no mesmo diretório onde estão os arquivos do projeto. Uma visão geral do programa é dada no diagrama UML da Figura 3.1.

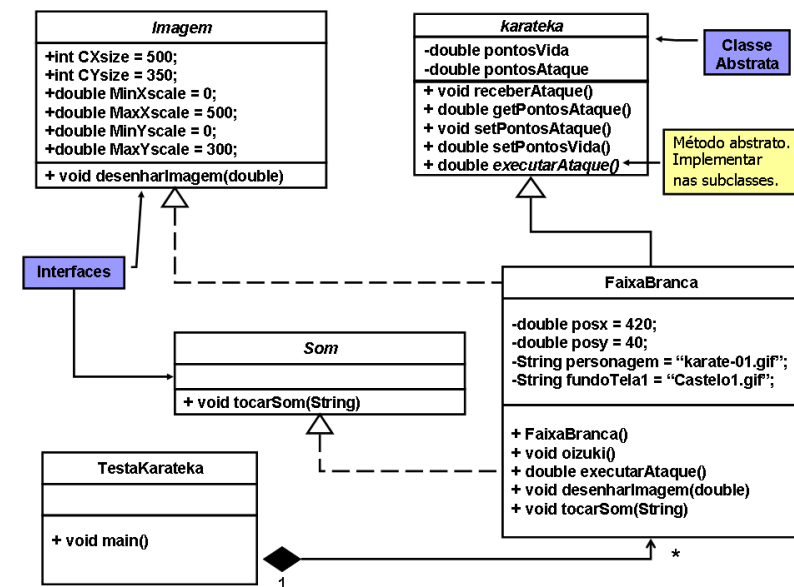


Figura 3.1: Diagrama UML das classes para construir o jogo do Karateka.

O detalhamento de como implementar o diagrama UML mostrado na Figura 3.1 é como se segue.

Interface Imagem

// Esta interface especifica um atributo
// comum e declara um comportamento comum
// a todos as classes que empregam imagens.

```
public interface Imagem
{
    int CXsize = 500;
    int CYsize = 350;
    double MinXscale = 0;
    double MaxXscale = 500;
    double MinYscale = 0;
    double MaxYscale = 300;

    public void desenharImagem(double posx, double posy, String
imagem);
} // Fim Interface Commission.
```

Interface Som

// Esta interface especifica um comportamento comum
// a todos as classes que tocam sons.

```
public interface Som
{
    public void tocarSom(String name);
} // Fim classe abstrata Empregado.
```

Classe Abstrata Karateka

// Esta classe especifica um comportamento padrao
// de um Karateka.

```
public abstract class Karateka
{
    private double pontosVida;
    private double pontosAtaque;

    public abstract double executarAtaque();

    public void receberAtaque(double ataque)
    { setPontosVida(pontosVida - ataque); }

    public void setPontosVida(double v)
    {pontosVida = (v < 0.0)?0.0:v;}

    public double getPontosVida()
    {return pontosVida;}
}
```

```
public void setPontosAtaque(double v)
{pontosAtaque= (v < 0.0)?0.0:v;}

public double getPontosAtaque()
{return pontosAtaque;}

} // Fim classe abstrata Karateka.
```

Classe FaixaBranca

// Esta interface especifica um comportamento comum
// a todos as classes que tocam sons.

```
public class FaixaBranca extends Karateka implements Imagem, Som
{
    double posx = 420;
    double posy = 40;
    String personagem = "karate-01.gif";
    String fundoTela1 = "Castelo1.gif";

    public FaixaBranca()
    {
        StdDraw.setCanvasSize(CXsize, CYsize);
        StdDraw.setXscale(MinXscale, MaxXscale);
        StdDraw.setYscale(MinYscale, MaxYscale);
        StdDraw.clear(StdDraw.GREEN);
        // Desenhar o Castelo.
        desenharImagem(250.0,165.0,fundoTela1);
        // Desenhar o Karateka - Posicao inicial.
        desenharImagem(posx,posy,personagem);
        // Caracteristicas atuais do personagem.
        setPontosVida(10.0);
    }

    // Metodo que aplica um oizuki.
    public void oizuki()
    { // Nova imagem do personagem.
        personagem = "karate-04.gif";
        // Redesenhar o Karateka - Nova Posicao.
        desenharImagem(posx,posy,personagem);
        // Som a ser produzido pelo ataque.
        tocarSom("oizuki.wav");
        // Define o dano a ser provocado.
        setPontosAtaque(2.0);
        // Voltando a imagem inicial do personagem.
        personagem = "karate-01.gif";
        desenharImagem(posx,posy,personagem);
    }
} // Faixa branca so pode aplicar um tipo de ataque !
```

```

public double executarAtaque()
{
    oizuki();
    return getPontosAtaque();
}

// Implementacao do metodo definido na interface Imagem.
public void desenharImagem(double posx, double posy, String s)
{StdDraw.picture(posx, posy, s);}

// Implementacao do metodo definido na interface Som.
public void tocarSom(String s)
{StdAudio.play(s);}

} // Fim classe FaixaBranca.

```

Classe TestaKarateka

```

public class TestaKarateka
{
    // Testa Classe FaixaBranca.
    public static void main(String[] args)
    {

        // Cria e desenha o ambiente com o Karateka.
        FaixaBranca f1 = new FaixaBranca();

        // Se o usuario apertar o mouse em cima da area
        // onde esta o karateka, entao, um novo desenho
        // sera realizado.
        while (true)
        {
            // Ao clique do mouse aplica um oizuki.
            if (StdDraw.mousePressed()) f1.executarAtaque();
        }

    }
} // Fim classe abstrata Empregado.

```

Exercício 4: Baseado no **Exercício 1** acrescentar um novo golpe para a classe FaixaBranca. Além disso, uma nova classe FaixaVerde, que incorpora novos golpes em relação a classe FaixaBranca, deverá ser criado. Detalhes das modificações a serem implementadas são dadas na Figura 4.1, nas Tabelas 4.1 e 4.2.

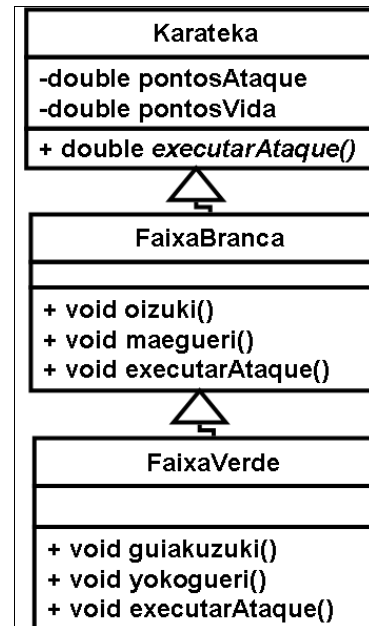


Figura 4.1: Hierarquia de classes.

Golpe	Dano
oizuki	[2.0]
maegueri	[3.0]
guiazuki	[2.75]
yokogueri	[3.5]

Tabela 4.1: Golpes associados a cada um dos métodos, valores a serem produzidos para cada um dos golpes (vide campo pontosAtaque) e número de pontos de vida a serem retirados do objeto que invocou o método.

Classe	Golpes a mostrar
Karateka	"Nenhum golpe !"
FaixaBranca	Sorteio aleatório entre oizuki e maegueri.
FaixaVerde	Sorteio aleatório entre todos da faixa branca e mais guiakuzuki e yokogueri.

Tabela 4.2: Tabela de golpes a serem mostrados, de acordo com a classe, quando o método executarAtaque() for chamado.

Além das mudanças listadas na Figura 4.1 e nas Tabelas 4.1 e 4.2, devem ser incorporadas ao sistema novas imagens e sons correspondentes aos novos golpes.

Criar um programa que empregue as classes definidas na Figura 4.1 e mostre o funcionamento dos métodos dos objetos de cada uma das classes.

Exercício 5: O **Exercício 1 da Lista 6** consistia em simular o cadastro de uma loja que vende CD e DVDS e que para tanto, utilizava o diagrama de classes dado na Figura 5.1.

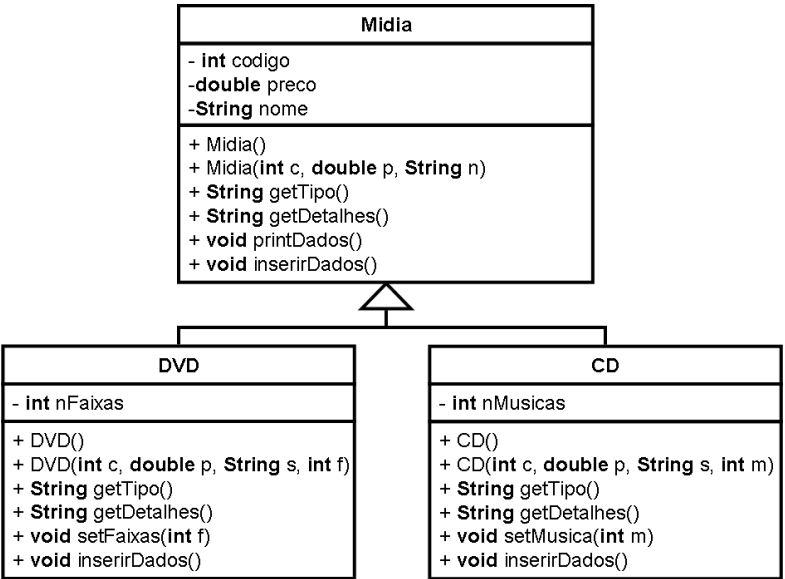


Figura 5.1: Hierarquia de classes para construir um cadastro de mídias.

A Tabela 5.1 fornece uma descrição dos métodos que deveriam ser elaborados para cada uma das classes.

Método	Descrição
getTipo()	Retorna uma String com o nome da classe.
getDetalhes()	Retorna uma String com as informações contidas nos campos.
printDados()	Imprime as informações contidas nos campos da classe. Para tanto, usa dois métodos para recuperar estas informações: getTipo() e getDetalhes(). Estas funções por sua vez são polimórficas, ou seja, seu tipo retorno varia de acordo com a classe escolhida, tal que este método é sobreposto nas subclasses.
inserirDados()	Insere os dados necessários para se preencher os campos de um objeto de uma dada classe. Seu comportamento é polimórfico.

Tabela 5.1: Descrição dos métodos a serem implementados.

Além dos métodos descritos na Tabela 5.1, deveriam ser criados os métodos **get** e **set** correspondentes para retorna e modificar o conteúdo dos campos,

respectivamente, bem como os construtores com e sem parâmetros de cada classe. Criar um programa que simule o uso de um cadastro de CD e DVDS.

Baseado no que foi apresentado nos **Exercícios 3 e 4** desta lista, deseja-se inserir uma nova funcionalidade na qual ao se chamar o método getDetalhes() das classes da Figura 5.1. Esta nova funcionalidade consiste em se tocar uma música ao se chamar o método getDetalhes para um objeto da classe CD (por exemplo a primeira música do CD) e mostrar uma imagem ao se chamar getDetalhes para um objeto da classe DVD (por exemplo a capa do DVD). Para tanto, as interfaces Imagem e Som deverão ser empregadas em conjunto com as bibliotecas StdDraw e StdAudio tal como ilustrado no **Exercício 3** desta lista. Inserir as alterações necessárias e testar.

Exercício 1: Criar um programa que utiliza que desenha uma interface gráfica tal como descrito na Figura 1.1.

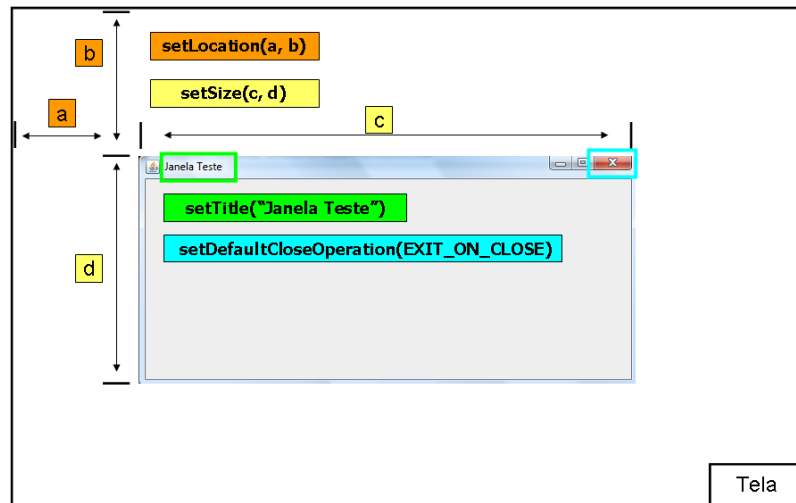


Figura 1.1: Primeira interface gráfica e seus elementos.

A resolução deste problema é como dado na Figura 1.2.

Classe Janela1

```
import java.awt.*;
import javax.swing.*;

public class Janela1 extends JFrame
{
    // Construtor e responsavel por inicializar propriedades da interface tais como:
    // Titulo, dimensoes e localizacao inicial.
    public Janela1()
    {
        // Titulo da interface.
        setTitle("Janela Teste");
        // Dimensões da interface.
        setSize(560,260);
        // Localização inicial da interface.
        setLocation(200,200);
    }
}
```

```
// Se a interface for fechada, então, terminar o programa.
setDefaultCloseOperation(EXIT_ON_CLOSE);
// Exibir a interface.
setVisible(true);
}

// Funcao para criar e testar a interface criada.
public static void main(String args[])
{
    Janela1 j1 = new Janela1();
}
}
```

Figura 1.2: Detalhamento da classe Janela1.

Exercício 2: Aperfeiçoar o programa do **Exercício 1** e inserir um **JButton** tal como descrito na Figura 2.1 e detalhado no código da Figura 2.2.



Figura 2.1: Interface gráfica com JButton.

A resolução deste problema é como dado na Figura 2.2.

Classe Janela2

```
import java.awt.*;
import javax.swing.*;

public class Janela2 extends JFrame
{
    JButton b1;

    public Janela2()
    {
        setTitle("Janela Teste 2");
        setSize(560,260);
        setLocation(100,100);
    }
}
```



```

setDefaultCloseOperation(EXIT_ON_CLOSE);
//-----//
// Criando e adicionando o botão.
//-----//
b1 = new JButton("Botão 1");
// Adicionando botão a interface (Janela2).
add(b1);
//-----//
// Exibir a interface com seus elementos.
setVisible(true);
}

public static void main(String args[])
{
    Janela2 j2 = new Janela2();
}
}

```

Figura 2.2: Detalhamento da classe Janela2.

Exercício 3: Aperfeiçoar o programa do **Exercício 2** e inserir 2 botões na interface gráfica tal como dado na Figura 3.1.

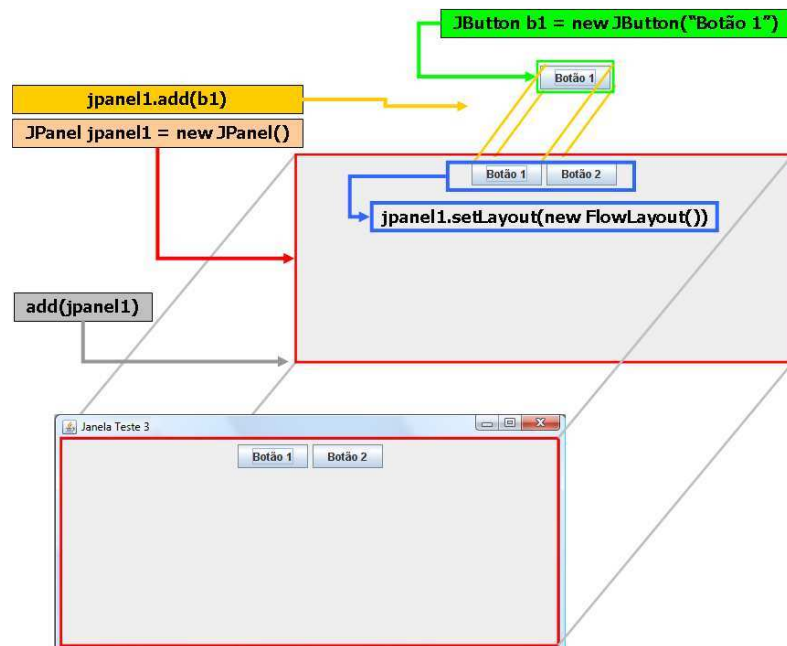


Figura 3.1: Elementos da interface gráfica com dois botões.

A resolução deste problema é como dado na Figura 3.2.

Classe Janela3

```

import java.awt.*;
import javax.swing.*;

public class Janela3 extends JFrame
{
    JButton b1, b2;
    JPanel jpanel1;

    public Janela3()
    {
        setTitle("Janela Teste 3");
        setSize(560,260);
        setLocation(200,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        // Criando efetivamente um jpanel.
        jpanel1 = new JPanel();
        // Definindo a arrumacao dos elementos
        // a serem inseridos no jpanel1.
        jpanel1.setLayout(new FlowLayout());
        // Criando botões.
        b1 = new JButton("Botão 1");
        b2 = new JButton("Botão 2");
        // Adicionando os botoes ao jpanel1.
        jpanel1.add(b1);
        jpanel1.add(b2);
        // Adicionando jpanel1 a interface.
        add(jpanel1);
        setVisible(true);
    }

    public static void main(String args[])
    {
        Janela3 j3 = new Janela3();
    }
}

```

Figura 3.2: Código para implementar a interface da Figura 3.1.

Exercício 4: Criar uma interface gráfica com **JButtons**, **JLabels** e **JTextFields**, bem como 3 **Jpanels** e diferentes formas de organização dos elementos gráficos (FlowLayout, GridLayout e BorderLayout) tal como ilustrado na Figura 4.1.

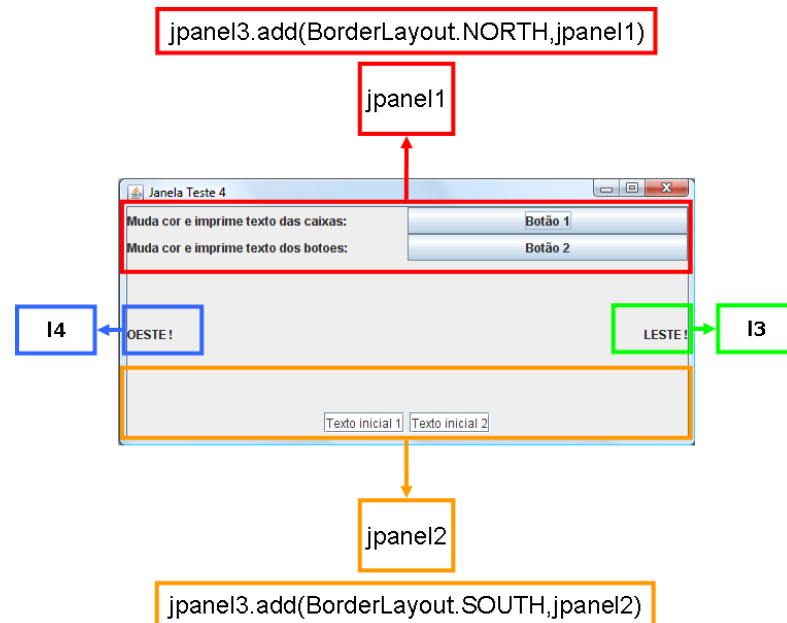


Figura 4.1: Interface gráfica com **JButtons**, **JLabels** e **JTextFields**.

A resolução deste problema é como dado na Figura 4.2.

Classe Janela4

```
import java.awt.*;
import javax.swing.*;

public class Janela4 extends JFrame
{
    JButton b1, b2;
    JLabel I1, I2, I3, I4;
    JTextField tf1, tf2;
    JPanel jpanel1, jpanel2, jpanel3;

    public Janela4()
    {
        setTitle("Janela Teste 4");
        setSize(560,260);
    }
}
```

```
setLocation(200,200);
setDefaultCloseOperation(EXIT_ON_CLOSE);
// Criando efetivamente jpanels.
jpanel1 = new JPanel();
jpanel2 = new JPanel();

//-----//
// CRIANDO JPANEL1 E SEUS ELEMENTOS. //
//-----//
// Definindo a arrumacao dos elementos no jpanel1.
jpanel1.setLayout(new GridLayout(2,2));
// Criando labels e botoes.
I1 = new JLabel("Muda cor e imprime texto das caixas:");
I2 = new JLabel("Muda cor e imprime texto dos botoes:");
b1 = new JButton("Botão 1");
b2 = new JButton("Botão 2");
// Adicionando os botoes ao jpanel1.
jpanel1.add(I1);
jpanel1.add(b1);
jpanel1.add(I2);
jpanel1.add(b2);
//-----//

//-----//
// CRIANDO JPANEL2 E SEUS ELEMENTOS. //
//-----//
// Definindo a arrumacao dos elementos no jpanel2.
jpanel2.setLayout(new FlowLayout());
// Criando botoes.
tf1 = new JTextField("Texto inicial 1");
tf2 = new JTextField("Texto inicial 2");
// Adicionando os botoes ao jpanel2.
jpanel2.add(tf1);
jpanel2.add(tf2);
//-----//

//-----//
// CRIANDO JPANEL3: JPANEL1 + JPANEL2. //
//-----//
// Criando efetivamente o jpanel3.
jpanel3 = new JPanel();
// Definindo a arrumacao dos elementos no jpanel3.
jpanel3.setLayout(new BorderLayout());
// Adicionando os jpanel 1 e 2 ao jpanel3.
jpanel3.add(BorderLayout.NORTH,jpanel1);
jpanel3.add(BorderLayout.SOUTH,jpanel2);
// Criando labels a serem adicionados ao jpanel3.
I3 = new JLabel("LESTE !");
```

```

I4 = new JLabel("OESTE !");
// Adicionando novos elementos a jpanel3.
jpanel3.add(BorderLayout.EAST,I3);
jpanel3.add(BorderLayout.WEST,I4);
//-----//

// Adicionando jpanel3 a interface.
add(jpanel3);

setVisible(true);
}

public static void main(String args[])
{
    Janela4 j4 = new Janela4();
}
}

```

Exercício 5: Modificar o **Exercício 4** de modo que a ação de clicar nos dois botões da interface gráfica produza modificações nas propriedades de elementos da interface gráfica tal como ilustrado na Figura 5.1. Além disso, uma caixa de mensagem (**JOptionPane**) é exibida toda vez que um dos botões é apertado. Para tanto, a interface gráfica deverá implementar (**implements**) o método **actionPerformed** da classe **ActionListener**.

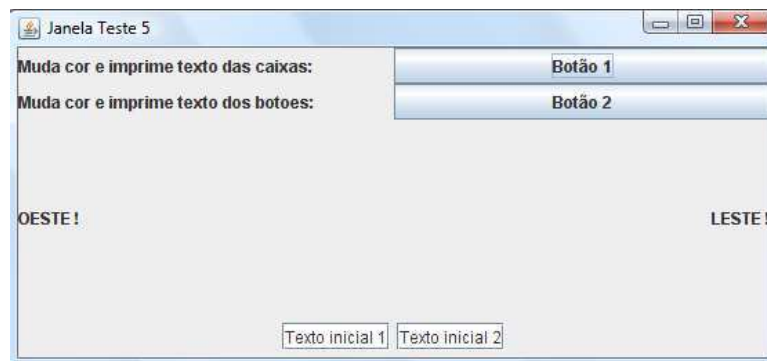


Figura 5.1(A): Estado inicial da interface gráfica.

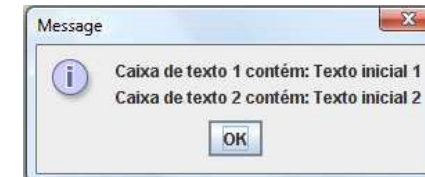


Figura 5.2(B): Caixa de diálogo após pressionar o Botão 1.

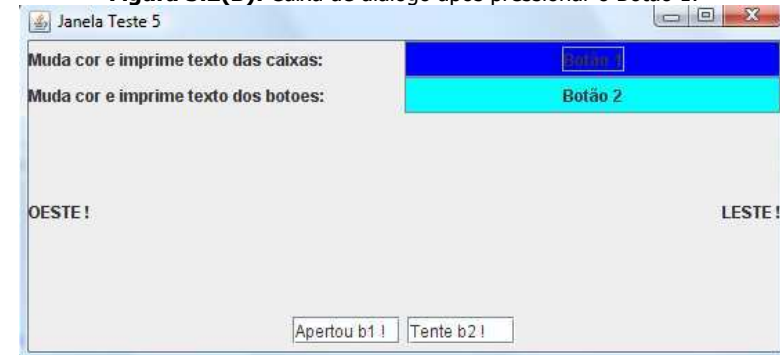


Figura 5.2(C): Estado da interface gráfica após pressionar o Botão 1.

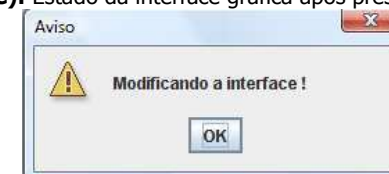


Figura 5.3(D): Caixa de diálogo após pressionar o Botão 2.

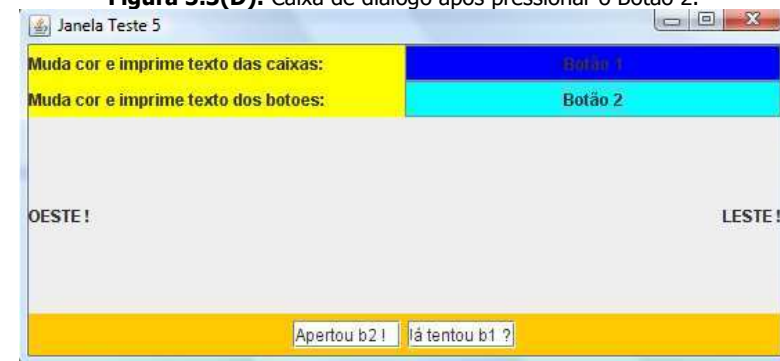


Figura 5.4(E): Estado da interface gráfica após pressionar o Botão 2.

```

import java.awt.event.*;
import java.awt.*;
import javax.swing.*;

public class Janela5 extends JFrame implements ActionListener
{
    JButton b1, b2;
    JLabel l1, l2, l3, l4;
    JTextField tf1, tf2;
    JPanel jpanel1, jpanel2, jpanel3;

    public Janela5()
    {
        setTitle("Janela Teste 5");
        setSize(560,260);
        setLocation(200,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        jpanel1 = new JPanel();
        jpanel2 = new JPanel();

        //-----//
        // CRIANDO JPANEL1 E SEUS ELEMENTOS.    //
        //-----//
        jpanel1.setLayout(new GridLayout(2,2));
        l1 = new JLabel("Muda cor e imprime texto das caixas:");
        l2 = new JLabel("Muda cor e imprime texto dos botoes:");
        b1 = new JButton("Botão 1");
        b2 = new JButton("Botão 2");

        //-----//
        // ADICIONANDO "LISTENER" AOS BOTOES DE           //
        // MODO QUE O TRATAMENTO SERA REALIZADO           //
        // NA PROPRIA CLASSE NO METODO ACTIONPERFORMED. //
        //-----//
        b1.addActionListener(this);
        b2.addActionListener(this);
        //-----//
        jpanel1.add(l1);
        jpanel1.add(b1);
        jpanel1.add(l2);
        jpanel1.add(b2);
        //-----//

        //-----//
        // CRIANDO JPANEL2 E SEUS ELEMENTOS.    //
        //-----//
        jpanel2.setLayout(new FlowLayout());
        tf1 = new JTextField("Texto inicial 1");

```

```

        tf2 = new JTextField("Texto inicial 2");
        jpanel2.add(tf1);
        jpanel2.add(tf2);
        //-----//

        //-----//
        // CRIANDO JPANEL3: JPANEL1 + JPANEL2.    //
        //-----//
        jpanel3 = new JPanel();
        jpanel3.setLayout(new BorderLayout());
        jpanel3.add(BorderLayout.NORTH,jpanel1);
        jpanel3.add(BorderLayout.SOUTH,jpanel2);
        l3 = new JLabel("LESTE !");
        l4 = new JLabel("OESTE !");
        jpanel3.add(BorderLayout.EAST,l3);
        jpanel3.add(BorderLayout.WEST,l4);
        //-----//

        // Adicionando jpanel3 a interface.
        add(jpanel3);

        setVisible(true);
    }

    // Adicionando acao ao botoes.
    public void actionPerformed(ActionEvent e)
    {
        // Verifica se o botao 1 foi apertado e reage realizando modificacoes nos
        // elementos da interface grafica e mostrando uma caixa de dialogo.
        if (e.getSource() == b1)
        {JOptionPane.showMessageDialog(null,
            "Caixa de texto 1 contém: " + tf1.getText() + "\n" +
            "Caixa de texto 2 contém: " + tf2.getText());

            tf1.setText("Apertou b1 !");
            b1.setBackground(Color.BLUE);
            tf2.setText("Tente b2 !");
            b2.setBackground(Color.CYAN);
        }

        // Verifica se o botao 2 foi apertado e reage realizando modificacoes nos
        // elementos da interface grafica e mostrando uma caixa de dialogo.
        if (e.getSource() == b2)
        {
            // Pode ser: JOptionPane.ERROR_MESSAGE,
            // JOptionPane.WARNING_MESSAGE ou
            // JOptionPane.PLAIN_MESSAGE.
            JOptionPane.showMessageDialog(null,

```

```

"Modificando a interface
!", "Aviso", JOptionPane.WARNING_MESSAGE);

tf1.setText("Apertou b2 !");
jpanel1.setBackground(Color.YELLOW);
tf2.setText("Já tentou b1 ?");
jpanel2.setBackground(Color.ORANGE);

}

}

public static void main(String args[])
{
    Janela5 j5 = new Janela5();
}

}

```

Exercício 6: Realizar uma dinâmica de grupo de modo que as seguintes etapas deverão ser realizada:

Passo 1: A turma deverá se organizar em grupos de modo que para cada grupo será atribuída uma das 3 interfaces dadas na Tabela 5.1.

Passo 2: Utilizando o material dos exercícios anteriores o grupo deverá fornecer o código para construir a sua respectiva interface. Todos os membros do grupo deverão ter uma cópia do código em papel ou em arquivo.

Passo 3: Novos grupos deverão ser formados de modo que cada membro do grupo possa explicar uma das 3 interfaces para os demais membros.

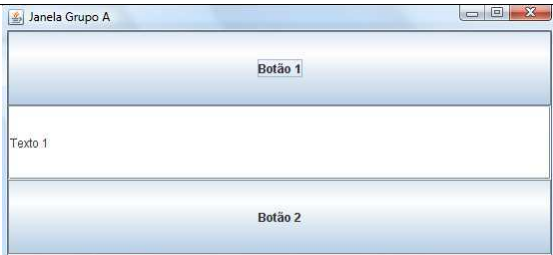
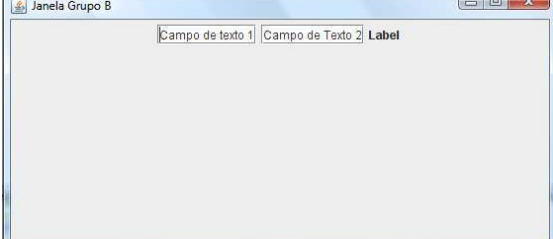
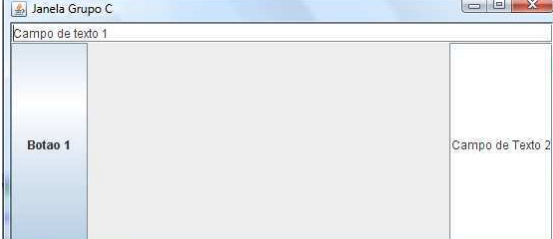
Grupo	Interface Gráfica
A	
B	
C	

Tabela 5.1: Interfaces a serem construídas por cada grupo.

Exercício 7: Construir uma interface gráfica que contenha **3 jpanels** nos moldes do que foi apresentado no **Exercício 4**. Para tanto, o seguinte algoritmo deverá ser seguido:

Passo 1: Some todos os números da sua matrícula e armazene em uma variável x. Se houver letras, use-as empregando a seguinte correspondência: a -> 1, b-> 2, c->3, e assim por diante.

Passo 2: Calcule o valor da variável m de acordo com a seguinte fórmula:
 $m = (x \% 30) + 1$.

Passo 3: Compare o valor de m com o valor contido na primeira coluna da **Tabela 7.1**. A este valor corresponderá uma dada interface gráfica que deverá ter 3 **JPanels** cujos **Layouts** e **elementos gráficos**. O **jpanel3**, em particular, deverá conter os **jpanels 1 e 2** de acordo com o Layout determinado na última coluna da Tabela 7.1. Tanto os **jpanels**, como os **elementos gráficos** podem ser determinados por meio das Tabelas 7.2 e 7.3.

Exemplo 1:

Seja a seguinte matrícula: 08001

Passo 1: 08001 -> $x = 0 + 8 + 0 + 0 + 1 = 9$

Passo 2: $m = (9 \% 30) + 1 = 9 + 1 = 10$

Passo 3: Fazer a interface gráfica com **jpanel1** que usa o Layout 2 (FlowLayout) e possui os componentes B e B (JTextField), com **jpanel2** que o usa o Layout 1 (GridLayout) e possui os componentes A e B (JButton e JTextField), e finalmente o **jpanel3** contém os **jpanel1** e 2 organizados de acordo com o Layout 3 (BorderLayout).

Exemplo 2:

Seja a seguinte matrícula: 08125

Passo 1: 08125 -> $x = 0 + 8 + 1 + 2 + 5 = 16$

Passo 2: $m = (16 \% 30) + 1 = 16 + 1 = 17$

Passo 3: Fazer a interface gráfica com **jpanel1** que usa o Layout 2 (FlowLayout) e possui os componentes A e B (JButton e JTextField), com **jpanel2** que o usa o Layout 3 (BorderLayout) e possui os componentes C e A (JLabel e JButton), e finalmente o **jpanel3** contém os **jpanel1** e 2 organizados de acordo com o Layout 1 (GridLayout).

Interface (m)	jpanel1		jpanel2		jpanel3
	Layout	Elementos	Layout	Elementos	Layout
1	1	A e B	2	A e C	3
2	2	B e C	3	A e A	1
3	3	C e A	1	B e B	2
4	2	A e C	3	C e C	2
5	3	A e A	1	A e B	1
6	1	B e B	2	B e C	3
7	3	C e C	2	C e A	1
8	1	A e C	1	B e B	3
9	2	A e A	3	C e C	2
10	2	B e B	1	A e B	3
11	1	C e C	3	B e C	2
12	3	A e B	2	C e A	1
13	1	B e C	3	A e C	2
14	3	C e A	2	A e A	3
15	2	B e B	1	A e B	1
16	3	C e C	2	B e C	3
17	2	A e B	3	C e A	1
18	1	B e C	1	A e C	2
19	2	C e A	3	A e A	2
20	3	A e C	1	B e B	1
21	1	A e A	2	C e C	3
22	3	A e B	2	A e C	1
23	1	B e C	1	A e A	3
24	2	C e A	3	B e B	2
25	2	A e C	1	C e C	1
26	1	A e A	3	A e B	2
27	3	B e B	2	B e C	3
28	1	C e C	1	C e A	2
29	3	A e C	2	B e B	3
30	2	A e A	3	C e C	1

Tabela 7.1: Correspondência entre m e a interface gráfica a ser construída.

Número	1	2	3
Layout	GridLayout	FlowLayout	BorderLayout

Tabela 7.2: Correspondência entre o número e o Layout a ser aplicado no **jpanel**.

Letra	A	B	C
Elemento	JButton	JTextField	JLabel

Tabela 7.3: Correspondência entre a letra e o elemento que deverá estar presente em cada **jpanel**.

Instruções Gerais para a elaboração de GUIs e tratamento de eventos:

Nesta lista de exercício, todos os problemas resolvidos seguem um padrão com o intuito de tornar mais didático o aprendizado acerca do tratamento de eventos para interfaces gráficas. Pode-se dizer que todas as interfaces gráficas desta lista seguem o seguinte esquema geral:

(1) Todos os elementos gráficos aos quais será adicionado algum tratamento de evento são declarados nos campos da classe da interface gráfica. Dessa forma, tanto os métodos responsáveis pelo posicionamento dos elementos gráficos na GUI, como os métodos tratadores de evento podem alterar o estado destes elementos.

(2) O construtor da classe define as propriedades da janela e chama um método para incorporar os demais elementos gráficos à janela (método createContents()). Este método, por sua vez, chama métodos auxiliares que são responsáveis pela organização da janela para áreas específicas.

(3) O tratamento de eventos, com os seus respectivos métodos, é realizado na própria classe onde está definida a interface.

Exercício 1: Criar uma interface gráfica tal como dado na Figura 1.1.

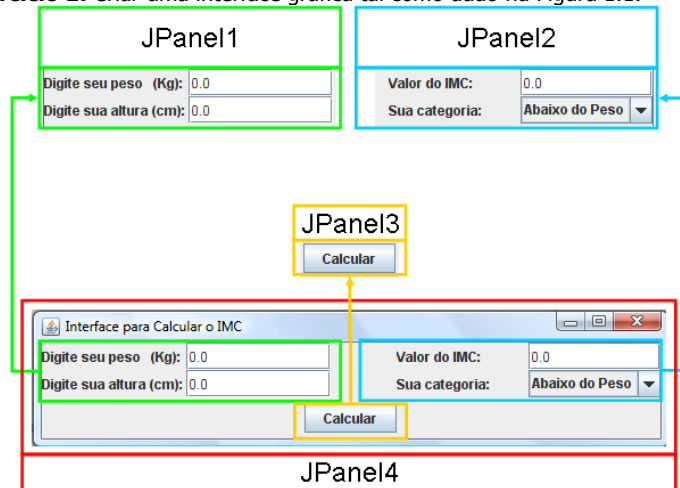


Figura 1.1: Interface para cálculo do IMC.

O código para a interface da Figura 1.1 é como dado na Figura 1.2.

Classe J1

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// Criando interface grafica que contem todos
// os elementos (detalhados em metodos) e que
// implementa a acao na propria classe.
public class J1 extends JFrame
{
    // Definicao dos elementos da GUI.
    private JPanel p1, p2, p3, p4;
    private JLabel l1, l2, l3, l4;
    private JTextField fieldNum1, fieldNum2, fieldNum3;
    private JComboBox box1;
    // Strings para cada opcao da caixa de escolha (JComboBox).
    private String categorias[] = {"Abaixo do Peso", "Peso Normal", "Acima do
    Peso", "Obesidade I", "Obesidade II", "Obesidade III"};
    private JButton btnSimular;

    // Definicao dos campos da classe de modo que todos os elementos da
    // interface sao declarados aqui.
    public J1()
    {
        setTitle("Interface para Calcular o IMC");
        setLocation(200,200);
        setSize(560,120);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        // Criando elementos da GUI.
        createContents();
        setVisible(true);
    }

    // Funcao principal que coordena a criacao de todos os JPanels.
    private void createContents()
    {
        // Criando o JPanel 1: Campos para inserir dados.
        p1 = createJPanel1();
        // Criando o JPanel 2: Campos para mostrar o resultado.
        p2 = createJPanel2();
        // Criando o JPanel 3: Botao de execucao do programa.
        p3 = createJPanel3();
        // Criando o JPanel 4: Engloba todos os JPanels.
        p4 = createJPanel4();
        // Adicionando o JPanel4 a Janela.
        add(p4);
    }
}
```

```
//-----//
// Criacao do JPanel p1 para inserir dados.
//-----//
private JPanel createJPanel1()
{
    p1 = new JPanel();
    l1 = new JLabel("Digite seu peso (Kg): ");
    l2 = new JLabel("Digite sua altura (cm): ");
    fieldNum1 = new JTextField("0.0");
    fieldNum2 = new JTextField("0.0");

    // Definindo o Layout para o JPanel1.
    p1.setLayout(new GridLayout(2,2));

    // Adicionando todos os componentes a JPanel p1.
    p1.add(l1);
    p1.add(fieldNum1);
    p1.add(l2);
    p1.add(fieldNum2);

    // Retornando JPanel p1 com elementos.
    return p1;
}
//-----//

//-----//
// Criacao do JPanel p2 mostrar resultados.
//-----//
private JPanel createJPanel2()
{
    p2 = new JPanel();
    p2.setLayout(new GridLayout(2,2));
    l3 = new JLabel("Valor do IMC:");
    l4 = new JLabel("Sua categoria:");
    fieldNum3 = new JTextField("0.0");
    box1 = new JComboBox(categorias);

    // Adicionando todos os componentes a JPanel p2.
    p2.add(l3);
    p2.add(fieldNum3);
    p2.add(l4);
    p2.add(box1);

    // Retornando JPanel p2 com elementos.
    return p2;
}
//-----//
```

```
//-----//
// Criacao do JPanel p3 com o botao de acao.
//-----//
private JPanel createJPanel3()
{
    p3 = new JPanel();
    p3.setLayout(new FlowLayout());
    btnSimular = new JButton("Calcular");

    // Adicionando todos os componentes a JPanel p3.
    p3.add(btnSimular);

    // Retornando JPanel p3 com elementos.
    return p3;
}
//-----//

//-----//
// Criacao do JPanel p4 para inserir JPanels.
//-----//
private JPanel createJPanel4()
{
    p4 = new JPanel();

    // Define o Layout antes de criar elementos na GUI.
    p4.setLayout(new BorderLayout());

    // Adicionando os 3 JPanels ao JPanel4.
    p4.add(p1,BorderLayout.WEST);
    p4.add(p2,BorderLayout.EAST);
    p4.add(p3,BorderLayout.SOUTH);

    return p4;
}
//-----//

public static void main(String args[])
{
    J1 janela1 = new J1();
}
}
```


Exercício 2: Inserir funcionalidade na interface gráfica dada no Exercício 1 de forma que ao se apertar o botão Calcular o IMC será calculado e a interface será modificada tal como dado na Figura 2.1.

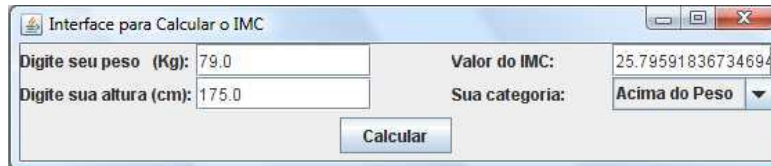


Figura 2.1: Interface para cálculo do IMC com tratamento de evento.

O código para a interface da Figura 1.1 é como dado na Figura 1.2.

Classe J2

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// Criando interface grafica que contem todos os elementos (detalhados em
// metodos) e que implementa a acao na propria classe.
public class J2 extends JFrame implements ActionListener
{
    // Definicao dos elementos da GUI.
    private JPanel p1, p2, p3, p4;
    private JLabel l1, l2, l3, l4;
    private JTextField fieldNum1, fieldNum2, fieldNum3;
    private JComboBox box1;
    // Strings para cada opcao da caixa de escolha (JComboBox).
    private String categorias[] = {"Abaixo do Peso", "Peso Normal", "Acima do
    Peso", "Obesidade I", "Obesidade II", "Obesidade III"};
    private JButton btnSimular;

    // Definicao dos campos da classe de modo que
    // todos os elementos da interface sao declarados aqui.
    public J2()
    {
        setTitle("Interface para Calcular o IMC");
        setLocation(200,200);
        setSize(560,120);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        // Criando elementos da GUI.
        createContents();
        setVisible(true);
    }
}
```

```
// Funcao principal que coordena a criacao de todos
// os elementos de todos os JPanels.
private void createContents()
{

    // Criando o JPanel 1: Campos para inserir dados.
    p1 = createJPanel1();

    // Criando o JPanel 2: Campos para mostrar o resultado.
    p2 = createJPanel2();

    // Criando o JPanel 3: Botao de execucao do programa.
    p3 = createJPanel3();

    // Criando o JPanel 4: Engloba todos os JPanels.
    p4 = createJPanel4();

    // Adicionando o JPanel4 a Janela.
    add(p4);

}

//-----//
// Criacao do JPanel p1 para inserir dados.
//-----//
private JPanel createJPanel1()
{
    p1 = new JPanel();
    l1 = new JLabel("Digite seu peso (Kg): ");
    l2 = new JLabel("Digite sua altura (cm): ");
    fieldNum1 = new JTextField("0.0");
    fieldNum2 = new JTextField("0.0");

    // Definindo o Layout para o JPanel1.
    p1.setLayout(new GridLayout(2,2));

    // Adicionando todos os componentes a JPanel p1.
    p1.add(l1);
    p1.add(fieldNum1);
    p1.add(l2);
    p1.add(fieldNum2);

    // Retornando JPanel p1 com elementos.
    return p1;
}
//-----//
```

```
//-----//
// Criacao do JPanel p2 mostrar resultados.
//-----//
private JPanel createJPanel2()
{
    p2 = new JPanel();
    p2.setLayout(new GridLayout(2,2));
    l3 = new JLabel("Valor do IMC:");
    l4 = new JLabel("Sua categoria:");
    fieldNum3 = new JTextField("0.0");
    box1 = new JComboBox(categorias);

    // Adicionando todos os componentes a JPanel p2.
    p2.add(l3);
    p2.add(fieldNum3);
    p2.add(l4);
    p2.add(box1);

    // Retornando JPanel p2 com elementos.
    return p2;
}
//-----//

//-----//
// Criacao do JPanel p3 com o botao de acao.
//-----//
private JPanel createJPanel3()
{
    p3 = new JPanel();
    p3.setLayout(new FlowLayout());
    btnSimular = new JButton("Calcular");

    // Adicionando "listener" ao botao.
    btnSimular.addActionListener(this);

    // Adicionando todos os componentes a JPanel p3.
    p3.add(btnSimular);

    // Retornando JPanel p3 com elementos.
    return p3;
}
//-----//
```

```
//-----//
// Criacao do JPanel p4 para inserir JPanels.
//-----//
private JPanel createJPanel4()
{
    p4 = new JPanel();

    // Define o Layout antes de criar elementos na GUI.
    p4.setLayout(new BorderLayout());

    // Adicionando os 3 JPanels ao JPanel4.
    p4.add(p1,BorderLayout.WEST);
    p4.add(p2,BorderLayout.EAST);
    p4.add(p3,BorderLayout.SOUTH);

    return p4;
}
//-----//

public void actionPerformed(ActionEvent e)
{
    // Verificando se o botao Simular foi acionado.
    if (e.getSource() == btnSimular)
    {
        // Obter dados dos campos. Cuidar para converter
        // corretamente de string para double. Senao,
        // serao utilizados valores padrao.
        double peso = 0.0;
        try
        {
            peso = Double.parseDouble(fieldNum1.getText());
        }
        catch( NumberFormatException ex)
        {
            // Caixa de texto apropriada.
            JOptionPane.showMessageDialog(null,
                "Inserir um numero com .",
                "Aviso",JOptionPane.WARNING_MESSAGE);
        }

        double altura = 0.0;
        try
        {
            altura = Double.parseDouble(fieldNum2.getText());
        }
        catch(NumberFormatException ex)
        {
            // Caixa de texto apropriada.
        }
    }
}
```

```

JOptionPane.showMessageDialog(null,
    "Inserir um numero com .
    !", "Aviso", JOptionPane.WARNING_MESSAGE);
}

// Realizando o calculo do IMC com os dados obtidos
// da interface: peso/((altura/100)^2).
double imc = peso / Math.pow(altura/100.0,2);

// Exibir o valor do imc na interface grafica.
// Nao esquecer de converter imc para String.
//fieldNum3.setText(Double.toString(imc));
fieldNum3.setText(imc+"");

// Modificar a opcao selecionada no JComboBox para a faixa
// adequada ao valor do imc.
int opt = 0;
if (imc < 18.5)
    opt = 0;
else if (imc < 25.0)
    opt = 1;
else if (imc < 30.0)
    opt = 2;
else if (imc < 35.0)
    opt = 3;
else if (imc < 40.0)
    opt = 4;
else
    opt = 5;

// Modificando o JComboBox para a mensagem adequada !
box1.setSelectedIndex(opt);
}
}

public static void main(String args[])
{ J2 janela1 = new J2(); }
}

```

Figura 2.1: Código Java do Exercício 2.

Item (A): Melhorar a interface da Figura 2.1, de modo que o usuário ao inserir um valor no campo correspondente ao valor do peso ou ainda no campo correspondente ao valor da altura, após inserir o valor e pressionar enter será disparada uma ação de modo que o IMC é calculado e no JComboBox é mostrada a classificação correspondente. Para tanto, as seguintes alterações de código deverão ser realizadas:

Alteração 1: No método createJPanel1

Trocar:

```

// Adicionando todos os componentes a JPanel p1.
p1.add(l1);
p1.add(fieldNum1);
p1.add(l2);
p1.add(fieldNum2);

```

Por:

```

// Adicionando todos os componentes a JPanel p1.
p1.add(l1);
// Adiciona tratamento de evento no JTextField fieldNum1: campo do peso.
fieldNum1.addActionListener(this);
p1.add(fieldNum1);
p1.add(l2);
// Adiciona tratamento de evento no JTextField fieldNum2: campo da altura.
fieldNum2.addActionListener(this);
p1.add(fieldNum2);

```

Alteração 2: No método actionPerformed

Trocar:

```

// Verificando se o botao Simular foi acionado.
if (e.getSource() == btnSimular)

```

Por:

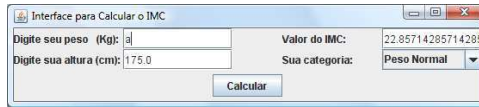
```

// Verificando se o botao Simular ou os campos fieldNum1 ou fieldNum2 foram
// acionados.
if ( (e.getSource() == btnSimular) || (e.getSource() == fieldNum1) ||
    (e.getSource() == fieldNum2) )

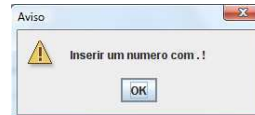
```

A alteração 1 permite que sejam inseridos listeners nos campos de modo que quando o usuário aperte enter seja disparados um tratamento de evento. O tratamento de evento será definido no método actionPerformed de modo que é necessário alterar este método também para permitir que evento originários de outras fontes além do JButton (cujo nome é btnSimular) possam ser também tratados.

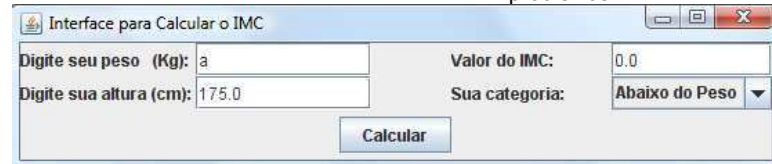
Item (B): Modificar a interface de modo que o comportamento 1 descrito na Figura 2.3 seja alterado para o comportamento 2 descrito na Figura 2.4.



(A) Antes de digitar "a" no peso.

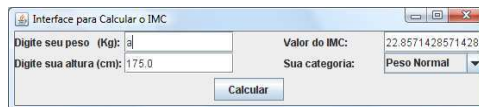


(B) Caixa de aviso de problemas.

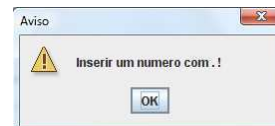


(C) Após mostrar caixa de aviso, o estado do campo peso continua com valor inválido.

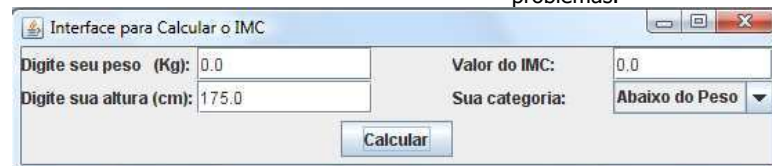
Figura 2.3: Comportamento 1 da interface gráfica do Exercício 2.



(A) Antes de digitar "a" no peso.



(B) Caixa de aviso de problemas.



(C) Após mostrar caixa de aviso, o estado do campo peso volta assumir o valor válido padrão, ou seja, o valor 0.0.

Figura 2.4: Comportamento 2 da interface gráfica do Exercício 2.

Item (C) Como modificar o programa para que a informação sobre o sexo seja considerada na interface através de um JComboBox? Se for sexo Masculino, então, usar a seguinte fórmula: $(\text{peso} - 2.0) / \text{altura}^2$. Para obter qual opção foi selecionada de um JComboBox, empregar o método **getSelectedIndex**.

Exercício 3: Criar um programa que realiza a simulação de números da Megasena de acordo com a interface fornecida na Figura 3.1.

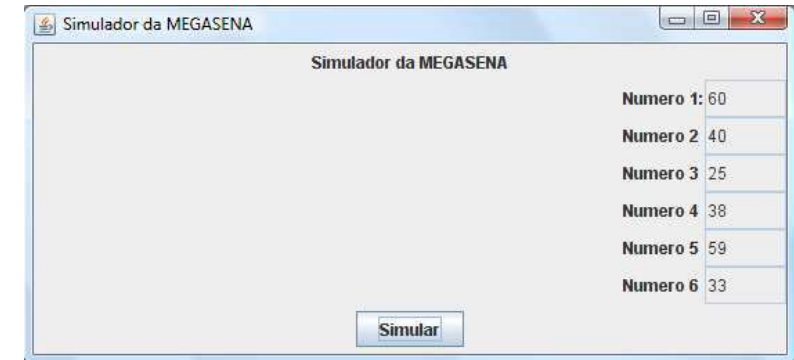


Figura 3.1: Interface do simulador de números da Megasena.

A resolução deste problema é como dado na Figura 3.2.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.Random;

public class J3 extends JFrame implements ActionListener
{
    // Definição dos elementos da GUI relacionadas com tratamento de evento.
    private JPanel p1, p2, p3, p4;
    private JTextField fieldNum1, fieldNum2, fieldNum3, fieldNum4, fieldNum5,
    fieldNum6;
    private JButton btnSimular;

    public J3()
    {
        setTitle("Simulador da MEGASENA");
        setSize(560, 260);
        setLayout(new BorderLayout());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        createContents();
        setVisible(true);
    }

    private void createContents()
    {
        // Criando o JPanel 1: Campo para mostrar uma mensagem.
        p1 = createJPanel1();
    }
}
```

```

// Criando o JPanel 2: Campos para o sorteio.
p2 = createJPanel2();

// Criando o JPanel 3: Botao do sorteio.
p3 = createJPanel3();

// Criando o JPanel 4: Organiza todos os JPanels.
p4 = createJPanel4();

// Adiciona todos os JPanels na Janela.
add(p4);
}

private JPanel createJPanel1()
{
    JPanel p1 = new JPanel();
    JLabel labelMensagem = new JLabel("Simulador da MEGASENA");
    // Adicionando todos os componentes a JPanel p1.
    p1.add(labelMensagem);
    return p1;
}

private JPanel createJPanel2()
{
    JPanel p2 = new JPanel();

    // Texto e campo para numero 1.
    JLabel labelNum1 = new JLabel("Numero 1:");
    JTextField fieldNum1 = new JTextField(5);
    fieldNum1.setText("");
    fieldNum1.setEditable(false);

    // Texto e campo para numero 2.
    JLabel labelNum2 = new JLabel("Numero 2");
    JTextField fieldNum2 = new JTextField(5);
    fieldNum2.setText("");
    fieldNum2.setEditable(false);

    // Texto e campo para numero 3.
    JLabel labelNum3 = new JLabel("Numero 3");
    JTextField fieldNum3 = new JTextField(5);
    fieldNum3.setText("");
    fieldNum3.setEditable(false);

    // Texto e campo para numero 4.
    JLabel labelNum4 = new JLabel("Numero 4");
    JTextField fieldNum4 = new JTextField(5);
    fieldNum4.setText("");
    fieldNum4.setEditable(false);
}

```

```

// Texto e campo para numero 5.
JLabel labelNum5 = new JLabel("Numero 5");
JTextField fieldNum5 = new JTextField(5);
fieldNum5.setText("");
fieldNum5.setEditable(false);

// Texto e campo para numero 6.
JLabel labelNum6 = new JLabel("Numero 6");
JTextField fieldNum6 = new JTextField(5);
fieldNum6.setText("");
fieldNum6.setEditable(false);

// Adicionando todos os componentes a JPanel2.
p2.setLayout(new GridLayout(6,1));
p2.add(labelNum1);
p2.add(fieldNum1);
p2.add(labelNum2);
p2.add(fieldNum2);
p2.add(labelNum3);
p2.add(fieldNum3);
p2.add(labelNum4);
p2.add(fieldNum4);
p2.add(labelNum5);
p2.add(fieldNum5);
p2.add(labelNum6);
p2.add(fieldNum6);
return p2;
}

private JPanel createJPanel3()
{
    JPanel p3 = new JPanel();
    // Adicionando botoes na interface.
    JButton btnSimular = new JButton("Simular");
    // Adicionando eventos ao botao, apenas.
    btnSimular.addActionListener(this);
    // Adicionando botao para realizar simulacao.
    p3.add(btnSimular);
    return p3;
}

public JPanel createJPanel4()
{
    JPanel p4 = new JPanel();
    p4.setLayout(new BorderLayout());
    // Adicionando os 3 JPanels a Janela.
    add(p1,BorderLayout.NORTH);
    add(p2,BorderLayout.EAST);
    add(p3,BorderLayout.SOUTH);
    return p4; }

```

```

public void actionPerformed(ActionEvent e)
{
    // Se o botao gerar foi pressionado, entao, gerar
    // 6 valores aleatorios.
    int d1, d2, d3, d4, d5, d6;
    if (e.getSource() == btnSimular)
    { // Gerar 6 valores aleatorios.
        d1 = geraAleat();
        d2 = geraAleat();
        d3 = geraAleat();
        d4 = geraAleat();
        d5 = geraAleat();
        d6 = geraAleat();

        // Interacoes com a interface.
        // Colocando o numero gerado no campo correspondente.
        fieldNum1.setText(Integer.toString(d1));
        fieldNum2.setText(Integer.toString(d2));
        fieldNum3.setText(Integer.toString(d3));
        fieldNum4.setText(Integer.toString(d4));
        fieldNum5.setText(Integer.toString(d5));
        fieldNum6.setText(Integer.toString(d6));

    } // Fim do if de apertar o botao.
}

private int geraAleat()
{
    Random r = new Random();
    // Para gerar numeros inteiros aleatórios em [1, 60].
    int a = 1;
    int b = 60;
    int numAleat = r.nextInt(b-a+1) + a; // Gera valores em [a,b].
    return numAleat; }

public static void main( String args[] )
{ J3 exe = new J3(); }

} // Fim Classe.

```

Figura 3.2: Detalhamento da classe J3.

Item (A): Qual o erro de lógica existente no programa da Figura 3.2? Descubra e modifique o programa de modo a corrigir o erro.

Item (B): Como modificar a interface para permitir a geração de 8 valores aleatórios? Descubra e modifique a interface.

Item (C): Como modificar o programa para gerar valores aleatórios no intervalo [1, 8]? Descubra e modifique. Observe ainda que com isso é possível verificar se o algoritmo elaborado para o **Item (B)** está correto.

Exercício 4: O Jogo de Craps consiste em lançar dois dados de seis faces (valores inteiros de 1 até 6) e depois obter a soma das faces viradas para cima. Se a soma for 7 ou 11 no primeiro lance, você ganha. Se a soma for 2, 3 ou 12 no primeiro lance (chamado Craps), você perde (isto é, a casa ganha). Se a soma for 4, 5, 6, 8, 9 ou 10 no primeiro lance, essa soma torna-se sua 'pontuação'. Para ganhar, você deve continuar a rolar os dados até 'fazer a sua pontuação' (isto é, obter um valor igual à sua pontuação). Você perde se obter um 7 antes de fazer a pontuação. Construir uma interface que simule o jogo de Craps tal como dado na Figura 4.1

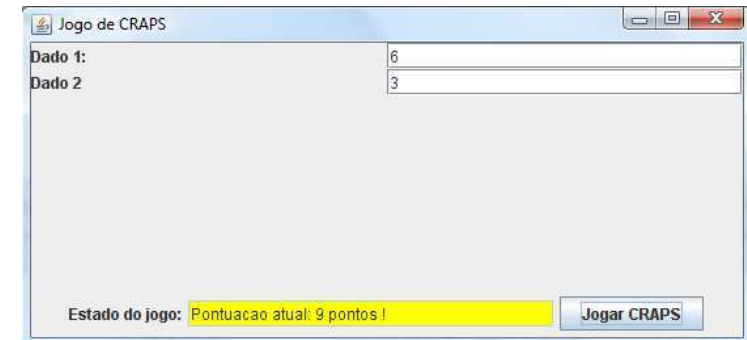


Figura 4.1: Sugestão de interface gráfica para o jogo Craps.

A resolução deste problema é como dado na Figura 4.2.

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.Random;

public class J4 extends JFrame implements ActionListener
{
    // Definicao dos elementos da GUI que precisam de tratamento de evento.
    private JPanel p1, p2, p3;
    private JTextField fieldDado1, fieldDado2, fieldMensagem;
    private JButton btnJogar;
    private int numDado1;
    private int numDado2;
    private int gameStatus; // 0 - Inicio, 1 - Continuar, 2 - Parar.
    private int totalPontos = 0;

    public J4()
    {
        setTitle("Jogo de CRAPS");
        setSize(560,260);
        setLayout(new BorderLayout());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}

```

```

createContents();
setVisible(true);
}

private void createContents()
{
    // Criando o JPanel 1: Campos para o sorteio.
    p1 = createJPanel1();

    // Criando o JPanel 2: Campo para mostrar uma mensagem.
    p2 = createJPanel2();

    // Criando o JPanel 3: Reúne todos os JPanels.
    p3 = createJPanel3();

    // Coloca na janela o JPanel3.
    add(p3);
}

private JPanel createJPanel3()
{
    p3 = new JPanel();
    p3.setLayout(new BorderLayout());

    // Adicionando os 2 JPanels a Janela.
    p3.add(p1, BorderLayout.NORTH);
    p3.add(p2, BorderLayout.SOUTH);

    return p3;
}

private JPanel createJPanel2()
{
    // Campo para impressao de mensagem sobre o estado do jogo.
    JPanel p2 = new JPanel();
    JLabel labelMensagem = new JLabel("Estado do jogo:");
    JTextField fieldMensagem = new JTextField(25);
    fieldMensagem.setText("");
    fieldMensagem.setEditable(false);
    fieldMensagem.setBackground(Color.yellow);

    // Adicionando botao de jogar CRAPS na interface.
    btnJogar = new JButton("Jogar CRAPS");

    // Adicionando eventos ao botao, apenas.
    btnJogar.addActionListener(this);
}

```

```

// Adicionando todos os componentes a JPanel3.
p2.setLayout(new FlowLayout());
p2.add(labelMensagem);
p2.add(fieldMensagem);
p2.add(btnJogar);

return p2;
}

private JPanel createJPanel1()
{
    JPanel p1 = new JPanel();

    // Texto e campo para informacao de limite inferior.
    JLabel labelDado1 = new JLabel("Dado 1:");
    JTextField fieldDado1 = new JTextField(5);
    fieldDado1.setText("0");
    // Texto e campo para informacao de limite superior.
    JLabel labelDado2 = new JLabel("Dado 2:");
    JTextField fieldDado2 = new JTextField(5);
    fieldDado2.setText("0");

    // Adicionando todos os componentes a JPanel1.
    p1.setLayout(new GridLayout(2,2));
    p1.add(labelDado1);
    p1.add(fieldDado1);
    p1.add(labelDado2);
    p1.add(fieldDado2);

    return p1;
}

public void actionPerformed(ActionEvent e)
{
    // Se o botao gerar foi pressionado, entao, gerar
    // dois valores aleatorios.
    if (e.getSource() == btnJogar)
    {
        int d1 = geraAleat();
        int d2 = geraAleat();
        int aux = d1 + d2;

        // Interacoes com a interface.
        // Colocando o numero gerado no campo correspondente.
        fieldDado1.setText(Integer.toString(d1));
        fieldDado2.setText(Integer.toString(d2));
    }
}

```



```

// Verificando qual o estado atual do jogo.
switch (gameStatus)
{

    case 0:// Começou o jogo agora !
        if ((aux == 7)|| (aux == 11)) // Ganhou na primeira rodada.
        {
            gameStatus = 0; // permitir reinício do jogo.
        }
        else if ((aux == 2)|| (aux == 3)|| (aux == 12)) // perdeu na 1a.
        {
            aux = 0;
            gameStatus = 0; // Permitir nova tentativa !
        }
        else // O jogo deve continuar.
        {
            gameStatus = 1;
        }
        // Atribuição de pontos de acordo com a situação.
        totalPontos = aux;
        break;

    default:// Continuando o jogo com a pontuação anterior !
        if (aux == totalPontos) // Ganha o jogo !
        {
            gameStatus = 0; // Começar do zero o jogo.
        }
        if (aux == 7)
        {
            gameStatus = 0; // Começar do zero o jogo.
            totalPontos = 0; // zerar a pontuação !
        }
        break;
} // Fim do switch.

// Realizando alterações na interface de acordo com o
// resultado obtido.
if (gameStatus != 1)
if (totalPontos > 0)
{
    fieldMensagem.setText("Ganhou o jogo: " + totalPontos + " pontos !");
    totalPontos = 0;
}
else
    fieldMensagem.setText("Perdeu o jogo: " + totalPontos + " pontos !");
else
    fieldMensagem.setText("Pontuação atual: " + totalPontos + " pontos !");

```

```

} // Fim do if de apertar o botão.

} // Fim do método actionPerformed.

private int geraAleat()
{
    Random r = new Random();
    // Para gerar números aleatórios usa os valores contidos
    // nos campos da interface.
    int a = 1;
    int b = 6;
    int numAleat = r.nextInt(b-a+1) + a; // Gera valores em [a,b].
    return numAleat;
}

public static void main( String args[] )
{
    J4 exe = new J4();
}

} // Fim classe J4.

```

Figura 4.2: Detalhamento da classe J4.

Exercício 5: Construir uma das 6 interfaces gráficas dadas a seguir. Para tanto, o seguinte algoritmo deverá ser seguido:

Passo 1: Some todos os números da sua matrícula e armazene em uma variável x. Se houver letras, use-as empregando a seguinte correspondência: a -> 1, b -> 2, c -> 3, e assim por diante.

Passo 2: Calcule o valor da variável m de acordo com a seguinte fórmula:

$$m = (x \% 6) + 1.$$

Passo 3: Compare o valor de m com o valor contido na primeira coluna da Tabela 5.1. A este valor corresponderá uma dada interface gráfica completa detalhada na Tabela 5.1.

Exemplo 1:

Seja a seguinte matrícula: 08001

Passo 1: 08001 -> $x = 0 + 8 + 0 + 0 + 1 = 9$

Passo 2: $m = (9 \% 6) + 1 = 3 + 1 = 4$

Passo 3: Fazer a interface gráfica 4 de acordo com a Tabela 5.1, ou seja, a interface gráfica correspondente a Prova P3A4.

Interface (m)	Prova
1	P3A1
2	P3A2
3	P3A3
4	P3A4
5	P3A5
6	P3A6

Tabela 5.1: Correspondência entre m e a interface gráfica a ser construída.

Nome:
Matricula:

Questão 1: (10,0):
Deseja-se construir uma interface gráfica que seja capaz de realizar simulações de condições de financiamento de imóveis. Uma sugestão de interface gráfica é dada na Figura 1.

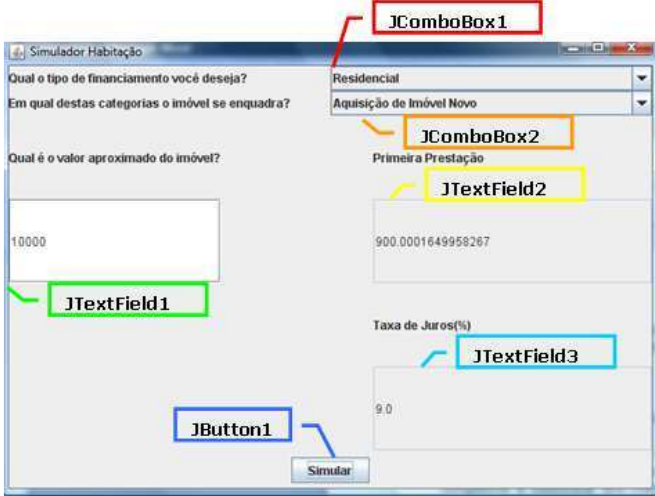


Figura 1: Sugestão de interface gráfica de simulação de financiamento de imóveis.

Observe que a Figura 1 fornece uma sugestão de organização dos elementos gráficos e que não necessariamente a sua interface deve ter a mesma organização. Porém, é obrigatória a existência dos mesmos elementos gráficos (**JButton**, **JTextField**, etc) que são apresentados na Figura 1. Uma especificação detalhada destes elementos é dada na Tabela 1.

Item (A) (5,0): Construir os elementos gráficos descritos na **Figura 1**. Alguns elementos têm seu funcionamento detalhado na **Tabela 1**.

Elemento Gráfico	Descrição Visual
JComboBox1	Oferecer duas opções de financiamento: <ul style="list-style-type: none"> • Comercial. • Residencial.
JComboBox2	Oferecer duas opções de categorias de imóvel: <ul style="list-style-type: none"> • Aquisição de Imóvel Novo. • Aquisição de Imóvel Usado.
JTextField1	Permitir a inserção do valor do imóvel.
JTextField2	Mostrar o resultado da parcela mensal do financiamento após se pressionar o JButton1 .
JTextField3	Mostrar a taxa de juros do financiamento após pressionar o JButton1 .
JButton1	Calcular 1ª parcela do financiamento e juros ao ser pressionado.

Tabela 1: Elementos gráficos e sua descrição visual.
Elaborar também os **JLabels** associados aos elementos gráficos.

Item (B) (5,0): Inserir funcionalidade no botão **JButton1** de modo que ao se pressionar o mesmo a primeira parcela do financiamento seja calculada de acordo com a Equação (1) (Sistema Price).

$$p = v \frac{i(1+i)^n}{(1+i)^n - 1} \quad (1)$$

Onde: p – valor da parcela, i é a taxa de juro utilizada (lembre-se que uma taxa de juros de 10% equivale à i=0.1), n é o número de meses de financiamento (neste caso considerado fixo em 180 meses) e v é o valor a ser financiado.

Para calcular p deve ser observado, ainda, que a taxa de juros depende da opção de financiamento (residencial ou comercial) e a categoria (novo ou usado) de acordo com os dados da Tabela 2.

	Comercial	Residencial
Imóvel Novo	11%	9%
Imóvel Usado	12%	10%

Tabela 2: Taxa de juros a ser empregada para cada opção de financiamento e categoria de imóvel.

Ao final do cálculo deverá ser mostrado o valor da parcela (no campo **JTextField2**) e o valor da taxa de juros empregada (no campo **JTextField3**).

Nome: **Matricula:**

Questão 1: (10,0):

Deseja-se construir uma interface gráfica que seja capaz de realizar simulações de condições de financiamento de imóveis. Uma sugestão de interface gráfica é dada na Figura 1.

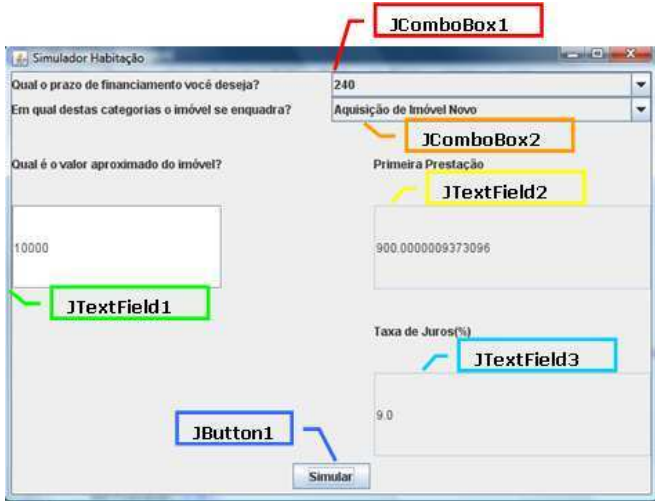


Figura 1: Sugestão de interface gráfica de simulação de financiamento de imóveis.

Observe que a Figura 1 fornece uma sugestão de organização dos elementos gráficos e que não necessariamente a sua interface deve ter a mesma organização. Porém, é obrigatória a existência dos mesmos elementos gráficos (**JButton**, **JTextField**, etc) que são apresentados na Figura 1. Uma especificação detalhada destes elementos é dada na Tabela 1.

Item (A) (5,0): Construir os elementos gráficos descritos na **Figura 1**. Alguns elementos têm seu funcionamento detalhado na **Tabela 1**.

Elemento Gráfico	Descrição Visual
JComboBox1	Oferecer quatro opções de prazo de financiamento (em meses): • 60, • 120, • 180, • 240.
JComboBox2	Oferecer duas opções de categorias de imóvel: • Aquisição de Imóvel Novo. • Aquisição de Imóvel Usado.
JTextField1	Permitir a inserção do valor do imóvel.
JTextField2	Mostrar o resultado da parcela mensal do financiamento após se pressionar o JButton1 .
JTextField3	Mostrar a taxa de juros do financiamento após pressionar o JButton1 .
JButton1	Calcular 1ª parcela do financiamento e juros ao ser pressionado.

Tabela 1: Elementos gráficos e sua descrição visual.

Elaborar também os **JLabels** associados aos elementos gráficos.

Item (B) (5,0): Inserir funcionalidade no botão **JButton1** de modo que ao se pressionar o mesmo a primeira parcela do financiamento seja calculada de acordo com a Equação (1) (Sistema Price).

$$p = v \frac{i(1+i)^n}{(1+i)^n - 1} \quad (1)$$

Onde: p – valor da parcela, i é a taxa de juro utilizada (lembre-se que uma taxa de juros de 10% equivale à i=0.1), n é o número de meses do financiamento (depende do valor selecionado em **JComboBox1**) e v é o valor a ser financiado.

Para calcular p deve ser observado, ainda, que a taxa de juros depende da opção de categoria (novo ou usado) de acordo com os dados da Tabela 2.

Imóvel Novo	Imóvel Usado
7%	8%

Tabela 2: Taxa de juros a ser empregada para cada opção de categoria de imóvel.

Ao final do cálculo deverá ser mostrado o valor da parcela (no campo **JTextField2**) e o valor da taxa de juros empregada (no campo **JTextField3**).

Nome: **Matricula:**
Questão 1: (10,0):

Deseja-se construir uma interface gráfica que seja capaz de realizar simulações de condições de financiamento de imóveis. Uma sugestão de interface gráfica é dada na Figura 1.

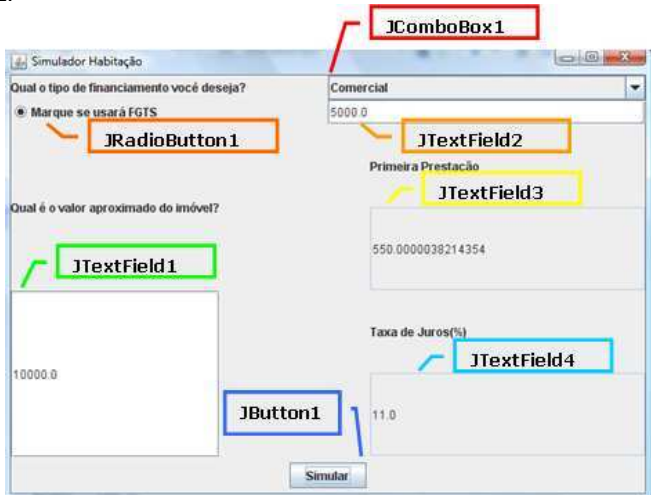


Figura 1: Sugestão de interface gráfica de simulação de financiamento de imóveis. Observe que a Figura 1 fornece uma sugestão de organização dos elementos gráficos e que não necessariamente a sua interface deve ter a mesma organização. Porém, é obrigatória a existência dos mesmos elementos gráficos (**JButton**, **JTextField**, etc) que são apresentados na Figura 1. Uma especificação detalhada destes elementos é dada na Tabela 1.

Item (A) (5,0): Construir os elementos gráficos descritos na **Figura 1**. Alguns elementos têm seu funcionamento detalhado na **Tabela 1**.

Elemento Gráfico	Descrição Visual
JComboBox1	Oferecer duas opções de financiamento: • Imóvel Residencial, • Imóvel Comercial.
JRadioButton1	Indicar se o valor contido no JTextField2 será usado no abatimento das parcelas obtidas no cálculo do financiamento.
JComboBox2	Oferecer duas opções de categorias de imóvel: • Aquisição de Imóvel Novo, • Aquisição de Imóvel Usado.
JTextField1	Permitir a inserção do valor do imóvel.
JTextField2	Permitir a inserção do valor do FGTS a ser abatido de JTextField1 .
JTextField3	Mostrar o resultado da parcela mensal do financiamento após se pressionar o JButton1 .
JTextField4	Mostrar a taxa de juros do financiamento após pressionar o JButton1 .
JButton1	Calcular 1ª parcela do financiamento e juros ao ser pressionado.

Tabela 1: Elementos gráficos e sua descrição visual. Elaborar também os **JLabels** associados aos elementos gráficos.

Item (B) (5,0): Inserir funcionalidade no botão **JButton1** de modo que ao se pressionar o mesmo a primeira parcela do financiamento seja calculada de acordo com a Equação (1) (Sistema Price).

$$p = v \frac{i(1+i)^n}{(1+i)^n - 1} \quad (1)$$

Onde: p – valor da parcela, i é a taxa de juro utilizada (lembre-se que uma taxa de juros de 10% equivale à i=0.1), n é o número de meses do financiamento (considerado fixo e igual a 240) e v é o valor a ser financiado (**JTextField1**)reduzido do valor do FGTS se o **JRadioButton1** estiver selecionado.

Para calcular p deve ser observado, ainda, que a taxa de juros depende da opção de financiamento (residencial ou comercial) de acordo com os dados da Tabela 2.

Residencial	Comercial
8.7%	10.1%

Tabela 2: Taxa de juros a ser empregada para cada opção de financiamento.

Ao final do cálculo deverá ser mostrado o valor da parcela (no campo **JTextField3**) e o valor da taxa de juros empregada (no campo **JTextField4**).

PROVA P3A4

Nome: **Matricula:**

Questão 1: (10,0):
Deseja-se construir uma interface gráfica que seja capaz de realizar simulações dos custos de aquisição de um carro. Uma sugestão de interface gráfica é dada na Figura 1.

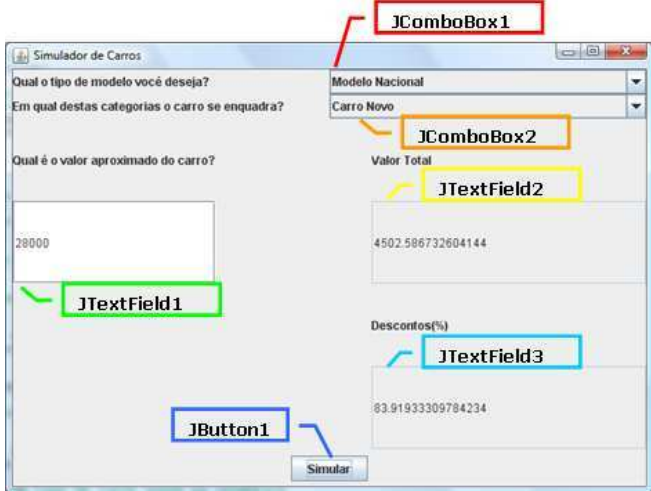


Figura 1: Sugestão de interface gráfica de simulação de custos de carros.

Observe que a Figura 1 fornece uma sugestão de organização dos elementos gráficos e que não necessariamente a sua interface deve ter a mesma organização. Porém, é obrigatória a existência dos mesmos elementos gráficos (**JButton**, **JTextField**, etc) que são apresentados na Figura 1. Uma especificação detalhada destes elementos é dada na Tabela 1.

Item (A) (5,0): Construir os elementos gráficos descritos na **Figura 1**. Alguns elementos têm seu funcionamento detalhado na **Tabela 1**.

Elemento Gráfico	Descrição Visual
JComboBox1	Oferecer duas opções de modelo: <ul style="list-style-type: none">• Modelo Nacional.• Modelo Importado.
JComboBox2	Oferecer duas opções de categorias de carro <ul style="list-style-type: none">• Carro Novo.• Carro Usado.
JTextField1	Permitir a inserção do valor do carro novo sem descontos.
JTextField2	Mostrar o resultado do valor total do carro após se pressionar o JButton1 .
JTextField3	Mostrar o total de descontos após pressionar o JButton1 .
JButton1	Calcular valor total do carro e total de descontos ao ser pressionado.

Tabela 1: Elementos gráficos e sua descrição visual.

Elaborar também os **JLabels** associados aos elementos gráficos.

Item (B) (5,0): Inserir funcionalidade no botão **JButton1** de modo que ao se pressionar o mesmo o valor total seja calculado de acordo com a Equação (1).

$$p = v(1 - i)^n \quad (1)$$

Onde: p – valor da parcela, i é o desconto por ano (lembre-se que um desconto de 10% equivale à i=0.1), n é o número de meses de idade do carro (neste caso considerado fixo em 60 meses) e v é o valor do carro sem descontos.

Para calcular p deve ser observado, ainda, que o desconto depende da opção de modelo (nacional ou importado) e a categoria (novo ou usado) de acordo com os dados da Tabela 2.

	Nacional	Importado
Carro Novo	3%	1%
Carro Usado	10%	8%

Tabela 2: Desconto a ser empregado para cada opção de modelo e categoria de carro.

Ao final do cálculo deverá ser mostrado o valor total (no campo **JTextField2**) e o valor total de desconto empregado (no campo **JTextField3**).

Nome: **Matricula:**
Questão 1: (10,0):

Deseja-se construir uma interface gráfica que seja capaz de realizar simulações de dos custos de aquisição de um carro. Uma sugestão de interface gráfica é dada na Figura 1.

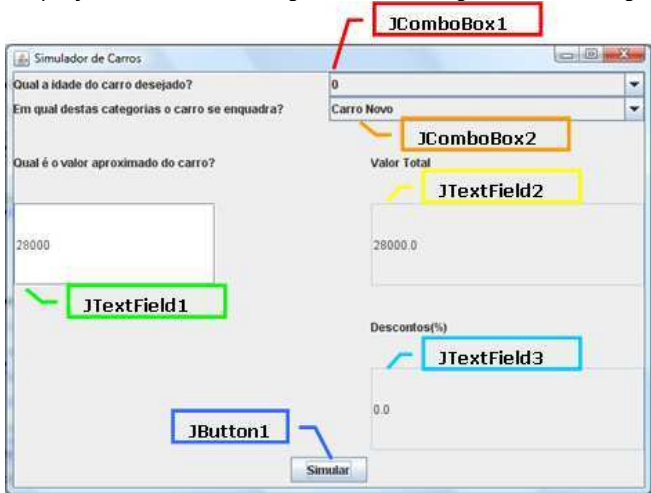


Figura 1: Sugestão de interface gráfica de simulação de financiamento de imóveis.

Observe que a Figura 1 fornece uma sugestão de organização dos elementos gráficos e que não necessariamente a sua interface deve ter a mesma organização. Porém, é obrigatória a existência dos mesmos elementos gráficos (**JButton**, **JTextField**, etc) que são apresentados na Figura 1. Uma especificação detalhada destes elementos é dada na Tabela 1.

Item (A) (5,0): Construir os elementos gráficos descritos na **Figura 1**. Alguns elementos têm seu funcionamento detalhado na **Tabela 1**.

Elemento Gráfico	Descrição Visual
JComboBox1	Oferecer quatro opções de idade de um carro (em meses): •0, •12, •36, •60.
JComboBox2	Oferecer duas opções de categorias de carro: • Carro Novo. • Carro Usado.
JTextField1	Permitir a inserção do valor do carro novo sem descontos.
JTextField2	Mostrar o resultado do valor total do carro após se pressionar o JButton1 .
JTextField3	Mostrar o total de descontos após pressionar o JButton1 .
JButton1	Calcular o total do carro e total de descontos ao ser pressionado.

Tabela 1: Elementos gráficos e sua descrição visual.

Elaborar também os **JLabels** associados aos elementos gráficos.

Item (B) (5,0): Inserir funcionalidade no botão **JButton1** de modo que ao se pressionar o mesmo o valor total do carro seja calculado de acordo com a Equação (1).

$$p = v(1 - i)^n \quad (1)$$

Onde: p – valor da parcela, i é o desconto empregado (lembre-se que um desconto de 10% equivale à i=0.1), n é o número de meses da idade do carro (depende do valor selecionado em **JComboBox1**) e v é o valor do carro sem descontos.

Para calcular p deve ser observado, ainda, que a taxa de juros depende da opção da categoria do carro (novo ou usado) de acordo com os dados da Tabela 2.

Carro Novo	Carro Usado
7%	8%

Tabela 2: Taxa de juros a ser empregada para cada opção de categoria de carro.

Ao final do cálculo deverá ser mostrado o valor total do carro (no campo **JTextField2**) e o valor total dos descontos empregados (no campo **JTextField3**).

PROVA P3A6

Nome:

Matricula:

Questão 1: (10,0):

Deseja-se construir uma interface gráfica que seja capaz de realizar simulações de dos custos de aquisição de um carro. Uma sugestão de interface gráfica é dada na Figura 1.

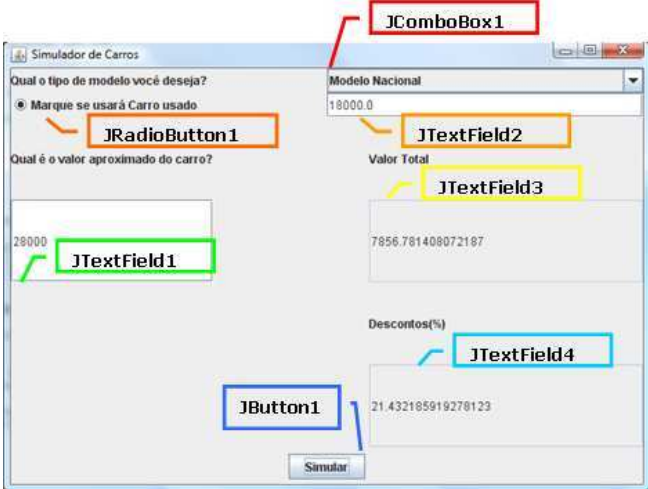


Figura 1: Sugestão de interface gráfica de simulação de financiamento de imóveis.

Observe que a Figura 1 fornece uma sugestão de organização dos elementos gráficos e que não necessariamente a sua interface deve ter a mesma organização. Porém, é obrigatória a existência dos mesmos elementos gráficos (**JButton**, **JTextField**, etc) que são apresentados na Figura 1. Uma especificação detalhada destes elementos é dada na Tabela 1.

Item (A) (5,0): Construir os elementos gráficos descritos na **Figura 1**. Alguns elementos têm seu funcionamento detalhado na **Tabela 1**.

Elemento Gráfico	Descrição Visual
JComboBox1	Oferecer duas opções de financiamento: • Modelo Nacional, • Modelo Importado.
JRadioButton1	Indicar se o valor contido no JTextField2 será usado no abatimento do valor total do carro.
JTextField1	Permitir a inserção do valor do carro.
JTextField2	Permitir a inserção do valor do carro a ser abatido de JTextField1 .
JTextField3	Mostrar o resultado do valor total do carro após se pressionar o JButton1 .
JTextField4	Mostrar os descontos totais do valor do carro após pressionar o JButton1 .
JButton1	Calcular valor total do carro e total dos descontos.

Tabela 1: Elementos gráficos e sua descrição visual.

Elaborar também os **JLabels** associados aos elementos gráficos.

Item (B) (5,0): Inserir funcionalidade no botão **JButton1** de modo que ao se pressionar o mesmo o valor total do carro seja calculado de acordo com a Equação (1).

$$p = v(1 - i)^n \quad (1)$$

Onde: p – valor da parcela, i é o desconto empregado (lembre-se que um desconto de 10% equivale à i=0.1), n é o número de meses da idade do carro (considerado fixo e igual a 24) e v é o valor total sem descontos do carro (**JTextField1**) reduzido do valor do carro usado (**JTextField2**) se o **JRadioButton1** estiver selecionado.

Para calcular p deve ser observado, ainda, que a taxa de juros depende da opção de modelo (nacional ou importado) de acordo com os dados da Tabela 2.

Nacional	Importado
1%	3%

Tabela 2: Desconto a ser empregado para cada opção de modelo.

Ao final do cálculo deverá ser mostrado o valor total do carro (no campo **JTextField3**) e o valor total de descontos (no campo **JTextField4**).

Exercício 1: Criar uma interface gráfica tal como dado na Figura 1.1.

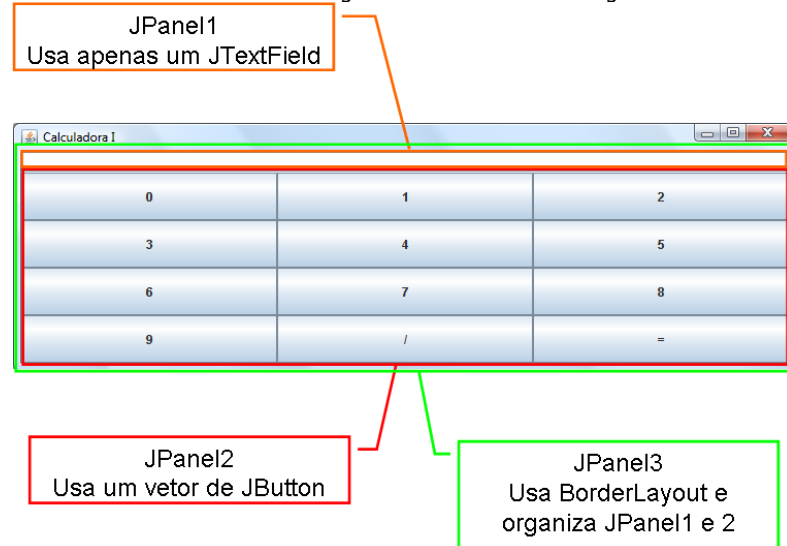


Figura 1.1: Interface Gráfica que funciona como Calculadora.

O código para a interface da Figura 1.1 é como dado na Figura 1.2.

Classe E11J1a

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class J11E1a extends JFrame implements ActionListener
{
    private JPanel p1, p2, p3;
    private JTextField txt1;
    private JButton vjbt[] = new JButton[12];
    private double number1, number2;
    private int opr = 1;

    // Construtor para inicializar a janela.
    public J11E1a()
    {
```

```
        setTitle("Calculadora I");
        setLocation(200,200);
        setSize(760,240);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        // Criando elementos da GUI.
        createContents();
        setVisible(true);
    }

    // Funcao principal que coordena a criacao de todos
    // os elementos de todos os JPanels.
    private void createContents()
    {

        // Criando o JPanel 1: Campo que mostra dados inseridos.
        p1 = createJPanel1();

        // Criando o JPanel 2: Campos para realizar operacoes.
        p2 = createJPanel2();

        // Criando o JPanel 3: Engloba todos os JPanels.
        p3 = createJPanel3();

        // Adicionando o JPanel3 a Janela.
        add(p3);

    }

    //-----//
    // Criando o JPanel 1: Campo que mostra dados inseridos.
    //-----//
    private JPanel createJPanel1()
    {
        p1 = new JPanel();
        txt1 = new JTextField("");

        // Definindo o Layout para o JPanel1.
        p1.setLayout(new GridLayout(1,1));

        // Adicionando todos os componentes a JPanel p1.
        p1.add(txt1);

        // Retornando JPanel p1 com elementos.
        return p1;
    }
    //-----//
```



```

//-----//
// Criando o JPanel 2: Campos para realizar operacoes.
//-----//
private JPanel createJPanel2()
{
    p2 = new JPanel();
    p2.setLayout(new GridLayout(4,3));

    // Insercao dos JButtons relativos aos digitos e operacoes.
    for (int i=0, j = 0; i < 12; i++)
    {
        // Verifica se deve inserir operacao.
        if ((i == 10)|| (i == 11))
        {
            // Verificando qual operacao sera inserida.
            switch(i)
            {
                // Operacao /.
                case 10: vjbt[i] = new JButton("/"); break;
                // Operacao =.
                case 11: vjbt[i] = new JButton("="); break;
            }
        }
        // Inserindo digitos.
        else
        {
            // Criando o botao correspondente ao digito j.
            vjbt[i] = new JButton(j+"");
            // Atualizando o contador de digitos.
            j++;
        }
        // Adicionando tratamento de evento a todos os botoes.
        vjbt[i].addActionListener(this);
        // Adicionando todos os componentes a JPanel p2.
        p2.add(vjbt[i]);
    }

    // Retornando JPanel p2 com elementos.
    return p2;
}
//-----//

//-----//
// Criacao do JPanel p3 para inserir JPanels.
//-----//
private JPanel createJPanel3()
{
    p3 = new JPanel();

```

```

// Define o Layout antes de criar elementos na GUI.
p3.setLayout(new BorderLayout());

// Adicionando os 2 JPanels ao JPanel3.
p3.add(p1,BorderLayout.NORTH);
p3.add(p2,BorderLayout.CENTER);

return p3;
}
//-----//

//-----//
//-----//
// TRATAMENTO DE Acao DOS BOTOES. //
//-----//
//-----//
//-----//
public void actionPerformed(ActionEvent event)
{
    //-----//
    // Verificando se uma operacao foi pressionada: /.
    //-----//
    if (event.getSource() == vjbt[10])
    {
        // Armazenar o numero que esta no JTextField.
        try
        {
            number1 = Double.parseDouble(txt1.getText());
        }
        // Se nao for possivel, tratar erro !
        catch(NumberFormatException e)
        {
            number1 = 0.0;
            txt1.setText(number1+"");
        }

        // Armazenar a operacao pressionada para realizar a conta
        // quando a operacao = for pressionada.
        if (event.getSource() == vjbt[11])    opr = 1;

        // Limpar numero contido no JTextField.
        txt1.setText("");
    }
    //-----//
}

```

```
//-----//
// Verificando se a operacao = foi pressionada.
//-----//
else if (event.getSource() == vjbt[11])
{
    // Usar o numero que esta no JTextField.
    try
    {
        number2 = Double.parseDouble(txt1.getText());
    }
    // Se nao for possivel, tratar erro !
    catch(NumberFormatException e)
    {
        number2 = 0.0;
        txt1.setText(number2+"");
    }

    // Calcular o resultado, usando a operacao anteriormente
    // digitada, bem como o numero armazenado.
    double number3 = 0.0;
    switch (opr)
    {
        case 1: number3 = number1/number2; break;
    }

    // Colocar o resultado no JTextField.
    txt1.setText(number3+"");

}
//-----//

//-----//
// Verificando se algum digito foi pressionado e atualizar
// a interface.
//-----//
else
{
    // Dígito.
    String dig = "";
    // Verificar a qual digito corresponde um dado botão que
    // foi pressionado.
    if (event.getSource() == vjbt[0])    dig = "0";
    else if (event.getSource() == vjbt[1]) dig = "1";
    else if (event.getSource() == vjbt[2]) dig = "2";
    else if (event.getSource() == vjbt[3]) dig = "3";
    else if (event.getSource() == vjbt[4]) dig = "4";
    else if (event.getSource() == vjbt[5]) dig = "5";
```

```
else if (event.getSource() == vjbt[6]) dig = "6";
else if (event.getSource() == vjbt[7]) dig = "7";
else if (event.getSource() == vjbt[8]) dig = "8";
else if (event.getSource() == vjbt[9]) dig = "9";

    // Atualizar o valor na interface no campo txt1.
    txt1.setText(txt1.getText()+dig);

}
//-----//

}
//-----//

public static void main(String args[])
{
    J11E1a janela1 = new J11E1a();
}

}
```

Figura 1.2 : Código Java para a interface gráfica da Figura 1.1.

Item (A): Modificar o programa de modo que se o usuário digitar enter no JTextField txt1, então, esta ação dispara um tratamento de evento igual ao tratamento de evento realizado qual o botão associado a operação "=" é pressionado. Para realizar tal alteração no comportamento do programa é necessário modificar o código:

Modificação 1: No método createJPanel1

Trocar

```
// Adicionando todos os componentes a JPanel p1.
p1.add(txt1);
```

Por

```
// Adicionando todos os componentes a JPanel p1.
// Adiciona tratamento de evento ao JTextField txt1.
txt1.addActionListener(this);
p1.add(txt1);
```

Modificação 2: No método actionPerformed

Trocar

```
// Verificando se a operacao = foi pressionada.
else if (event.getSource() == vjbt[11])
```

Por

```
// Verificando se o botão da operacao = foi pressionada ou enter em txt1.
else if ((event.getSource() == vjbt[11]) || (event.getSource() == txt1))
```

Item (B): Modificar o programa de modo que se o usuário digitar enter ou clicar o mouse em cima do JTextField txt1, então, esta ação dispara um tratamento de evento igual ao tratamento de evento realizado qual o botão associado a operação "=" é pressionado. Para realizar tal alteração no comportamento do programa é necessário modificar o código:

Modificação 1: No início da declaração da classe

Trocar

```
// Implementa apenas ação sobre um elemento da GUI.
public class J11E1a extends JFrame implements ActionListener
```

Por

```
// Implementa ação sobre um elemento da GUI e evento com o mouse.
public class J11E1a extends JFrame implements ActionListener,
MouseListener
```

Modificação 2: No método createJPanel1

Trocar

```
// Adicionando os componentes a JPanel p1 e tratamento de eventos a txt1.
txt1.addActionListener(this);
p1.add(txt1);
```

Por

```
// Adicionando os componentes a JPanel p1 e tratamento de eventos a txt1.
txt1.addActionListener(this);
txt1.addMouseListener(this); // Tratamento de eventos com o mouse.
p1.add(txt1);
```

Inserção 1: Inserir ao final da classe novos métodos de MouseListener

```
//-----//
// Metodos para tratar o comportamento do mouse. //
//-----//
// Métodos da interface MouseListener
public void mousePressed(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}

//-----//
// Metodo que trata o evento de clique do mouse. //
//-----//
public void mouseClicked(MouseEvent event)
{
```

```
// Usar o numero que esta no JTextField.
try
{
    number2 = Double.parseDouble(txt1.getText());
}
// Se nao for possivel, tratar erro !
catch(NumberFormatException e)
{
    number2 = 0.0;
    txt1.setText(number2+"");
}
// Calcular o resultado, usando a operacao anteriormente
// digitada, bem como o numero armazenado.
double number3 = 0.0;
switch (opr)
{
    case 1: number3 = number1/number2; break;
}
// Colocar o resultado no JTextField.
txt1.setText(number3+"");
}

//-----//
```

Item (C): Modificar o programa de modo que o comportamento 1 descrito na Figura 1.3 seja alterado para o comportamento 2 descrito na Figura 1.4.



Figura 1.3: Comportamento 1 da interface gráfica definida no Exercício 1.

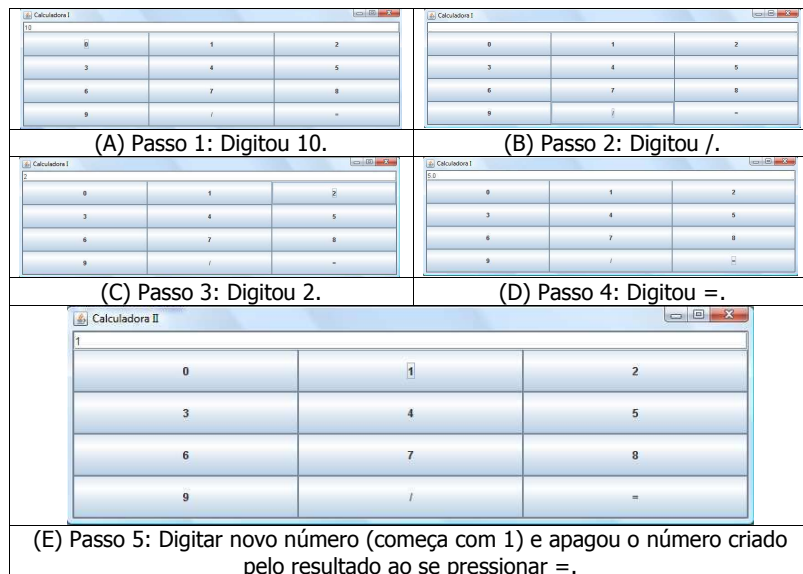


Figura 1.4: Comportamento 2 da interface gráfica definida no Exercício 1.

Para realizar tais alterações no comportamento do programa é necessário realizar as seguintes modificações no código:

Inserção 1: Inserir nos campos da classe um novo campo que controla quando apagar ou manter um valor no visor da calculadora.

// Nova variável capaz de controlar quando apagar o conteúdo do visor
// da calculadora

private boolean apagar = **true**;

Modificação 1: Modificar o método **actionPerformed** de modo que toda vez que o botão **=** for pressionado, então, o conteúdo do visor será substituído por um novo número a ser digitado.

```
//-----//
// TRATAMENTO DE AÇÃO DOS BOTOES. //
//-----//
public void actionPerformed(ActionEvent event)
{
    //-----//
    // Verificando se uma operacao foi pressionada: /.
    //-----//
    if (event.getSource() == vjbt[10])
```

```
{
    // Armazenar o numero que esta no JTextField.
    try
    {
        number1 = Double.parseDouble(txt1.getText());
    }
    // Se nao for possivel, tratar erro !
    catch(NumberFormatException e)
    {
        number1 = 0.0;
        txt1.setText(number1+""");
    }

    // Armazenar a operacao pressionada para realizar a conta
    // quando a operacao = for pressionada.
    if (event.getSource() == vjbt[10])    opr = 1;

    // Impedir que o numero contido no visor seja eliminado
    // da "memoria" da calculadora caso antes da operacao o
    // "=" tenha sido pressionado.
    if (apagar)
        apagar = false;

    // Limpar numero contido no JTextField.
    txt1.setText("");

}
//-----//

//-----//
// Verificando se a operacao = foi pressionada.
//-----//
else if (event.getSource() == vjbt[11])
{
    // Usar o numero que esta no JTextField.
    try
    {
        number2 = Double.parseDouble(txt1.getText());
    }
    // Se nao for possivel, tratar erro !
    catch(NumberFormatException e)
    {
        number2 = 0.0;
        txt1.setText(number2+""");
    }

    // Calcular o resultado, usando a operacao anteriormente
    // digitada, bem como o numero armazenado.
```

```

double number3 = 0.0;
switch (opr)
{
    case 1: number3 = number1/number2; break;
}

// Colocar o resultado no JTextField.
txt1.setText(number3+"");

// Se o igual foi pressionado, entao, da proxima vez
// que se pressionar um numero, apagar o resultado !!
apagar = true;

}
//-----//

//-----//
// Verificando se algum digito foi pressionado e atualizar
// a interface.
//-----//
else
{
    // Verificar se deve colocar novo valor na interface, pois
    // a operacao igual acabou de ser digitada !!
    if (apagar)
    {
        txt1.setText("");
        number1 = 0.0;
        number2 = 0.0;
        apagar = false;
    }

    // Inicializando o digito como vazio.
    String dig = "";
    // Armazenar a operacao pressionada para realizar a conta
    // quando a operacao = for pressionada.
    if (event.getSource() == vjbt[0]) dig = "0";
    else if (event.getSource() == vjbt[1]) dig = "1";
    else if (event.getSource() == vjbt[2]) dig = "2";
    else if (event.getSource() == vjbt[3]) dig = "3";
    else if (event.getSource() == vjbt[4]) dig = "4";
    else if (event.getSource() == vjbt[5]) dig = "5";
    else if (event.getSource() == vjbt[6]) dig = "6";
    else if (event.getSource() == vjbt[7]) dig = "7";
    else if (event.getSource() == vjbt[8]) dig = "8";
    else if (event.getSource() == vjbt[9]) dig = "9";

```

```

// Manter o valor na interface ate que o igual ou operacao
// seja digitada !
txt1.setText(txt1.getText()+dig);

}
//-----//

} // Fim do tratamento de eventos para os JButtons.

//-----//

```

Item (D): Modificar o programa de modo a inserir novos operadores na Calculadora tais como *, + e -. Modifique o programa para que seja possível também números decimais (com o uso do botão ".") tal como dado na Figura 1.5. Para tanto o código dado na Figura 1.6 deverá ser empregado.

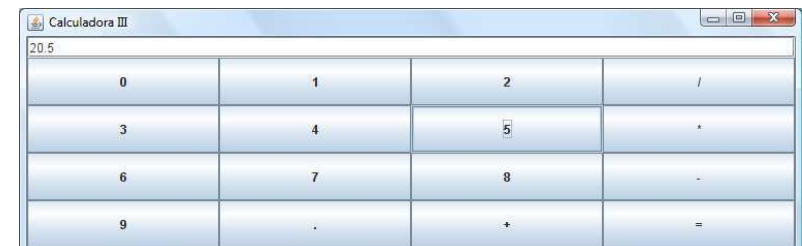


Figura 1.5: Nova interface gráfica para a Calculadora.

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class J11E1c extends JFrame implements ActionListener
{
    private JPanel p1, p2, p3;
    private JTextField txt1;
    private JButton vjbt[] = new JButton[16];
    private double number1, number2;
    private int opr = 1;
    private boolean opt = false;

    // Construtor para inicializar a janela.
    public J11E1c()
    {
        setTitle("Calculadora III");
    }

```

```

setLocation(200,200);
setSize(760,240);
setDefaultCloseOperation(EXIT_ON_CLOSE);
// Criando elementos da GUI.
createContents();
setVisible(true);
}

// Funcao principal que coordena a criacao de todos
// os elementos de todos os JPanels.
private void createContents()
{

    // Criando o JPanel 1: Campo que mostra dados inseridos.
    p1 = createJPanel1();

    // Criando o JPanel 2: Campos para realizar operacoes.
    p2 = createJPanel2();

    // Criando o JPanel 3: Engloba todos os JPanels.
    p3 = createJPanel3();

    // Adicionando o JPanel3 a Janela.
    add(p3);

}

//-----//
// Criando o JPanel 1: Campo que mostra dados inseridos.
//-----//
private JPanel createJPanel1()
{
    p1 = new JPanel();
    txt1 = new JTextField("");

    // Definindo o Layout para o JPanel1.
    p1.setLayout(new GridLayout(1,1));

    // Adicionando todos os componentes a JPanel p1.
    p1.add(txt1);

    // Retornando JPanel p1 com elementos.
    return p1;
}
//-----//

```

```

//-----//
// Criando o JPanel 2: Campos para realizar operacoes.
//-----//
private JPanel createJPanel2()
{
    p2 = new JPanel();
    p2.setLayout(new GridLayout(4,4));

    // Insercao dos JButtons relativos aos digitos e operacoes.
    for (int i=0, j = 0; i < 16; i++)
    {
        // Verifica se deve inserir operacao.
        if ((i == 3)|| (i == 7)|| (i == 11)|| (i == 13)|| (i == 14)|| (i == 15))
        {
            // Verificando qual operacao sera empregada.
            switch(i)
            {
                // Operacao /.
                case 3: vjbt[i] = new JButton("/"); break;
                // Operacao *.
                case 7: vjbt[i] = new JButton("*"); break;
                // Operacao -.
                case 11: vjbt[i] = new JButton("-"); break;
                // Operacao ..
                case 13: vjbt[i] = new JButton("."); break;
                // Operacao +.
                case 14: vjbt[i] = new JButton("+"); break;
                // Operacao =.
                case 15: vjbt[i] = new JButton("="); break;
            }
        }
        // Inserindo digitos.
        else
        {
            // Criando o botao correspondente ao digito j.
            vjbt[i] = new JButton(j+"");
            // Atualizando o contador de digitos.
            j++;
        }
        // Adicionando tratamento de evento a todos os botoes.
        vjbt[i].addActionListener(this);
        // Adicionando todos os componentes a JPanel p2.
        p2.add(vjbt[i]);
    }
    // Retornando JPanel p2 com elementos.
    return p2;
}
//-----//

```

```

//-----//
// Criacao do JPanel p3 para inserir JPanels.
//-----//
private JPanel createJPanel3()
{
    p3 = new JPanel();

    // Define o Layout antes de criar elementos na GUI.
    p3.setLayout(new BorderLayout());

    // Adicionando os 2 JPanels ao JPanel3.
    p3.add(p1,BorderLayout.NORTH);
    p3.add(p2,BorderLayout.CENTER);

    return p3;
}
//-----//

//-----//
// TRATAMENTO DE Acao DOS BOTOES. //
//-----//
public void actionPerformed(ActionEvent event)
{
    //-----//
    // Verificando se uma operacao foi pressionada: /, *, - e +.
    //-----//
    if (event.getSource() == vjbt[3] || event.getSource() == vjbt[7]
        || event.getSource() == vjbt[11] || event.getSource() == vjbt[14])
    {
        // Armazenar o numero que esta no JTextField.
        try
        {
            number1 = Double.parseDouble(txt1.getText());
        }
        // Se nao for possivel, tratar erro !
        catch(NumberFormatException e)
        {
            number1 = 0.0;
            txt1.setText(number1+"");
        }

        // Armazenar a operacao pressionada para realizar a conta
        // quando a operacao = for pressionada.
        if (event.getSource() == vjbt[3])      opr = 1;
        else if (event.getSource() == vjbt[7]) opr = 2;
        else if (event.getSource() == vjbt[11]) opr = 3;
        else                                   opr = 4;
    }
}

```

```

// Limpar numero contido no JTextField.
txt1.setText("");

// Pode digitar um ponto.
opt = false;
}
//-----//

//-----//
// Verificando se a operacao = foi pressionada.
//-----//
else if (event.getSource() == vjbt[15])
{
    // Usar o numero que esta no JTextField.
    try
    {
        number2 = Double.parseDouble(txt1.getText());
    }
    // Se nao for possivel, tratar erro !
    catch(NumberFormatException e)
    {
        number2 = 0.0;
        txt1.setText(number2+"");
    }

    // Calcular o resultado, usando a operacao anteriormente
    // digitada, bem como o numero armazenado.
    double number3 = 0.0;
    switch (opr)
    {
        case 1: number3 = number1/number2; break;
        case 2: number3 = number1*number2; break;
        case 3: number3 = number1-number2; break;
        case 4: number3 = number1+number2; break;
    }

    // Colocar o resultado no JTextField.
    txt1.setText(number3+"");

    // Pode digitar um ponto.
    opt = false;
}
//-----//

```

```

//-----//
// Verificando se um ponto foi pressionado.
//-----//
else if (event.getSource() == vjbt[13])
{
    // Atualizar a interface apenas se uma operacao acabou de ser digitada.
    if (!opt)
    {
        // Atualizar a interface.
        txt1.setText(txt1.getText()+".");
        opt = true;
    }
}
//-----//

//-----//
// Verificando se algum digito foi pressionado e atualizar
// a interface.
//-----//
else
{
    // Inicializando digito digitado como vazio.
    String dig = "";
    // Armazenar a operacao pressionada para realizar a conta
    // quando a operacao = for pressionada.
    if (event.getSource() == vjbt[0])    dig = "0";
    else if (event.getSource() == vjbt[1]) dig = "1";
    else if (event.getSource() == vjbt[2]) dig = "2";
    else if (event.getSource() == vjbt[4]) dig = "3";
    else if (event.getSource() == vjbt[5]) dig = "4";
    else if (event.getSource() == vjbt[6]) dig = "5";
    else if (event.getSource() == vjbt[8]) dig = "6";
    else if (event.getSource() == vjbt[9]) dig = "7";
    else if (event.getSource() == vjbt[10]) dig = "8";
    else if (event.getSource() == vjbt[12]) dig = "9";

    txt1.setText(txt1.getText()+dig);
}
//-----//
}
//-----//

public static void main(String args[])
{
    J11E1c janela1 = new J11E1c();
}

} // Final da classe.

```

Item (E): Cabe observar que a nova interface gráfica para a Calculadora dada na Figura 1.5 possui os mesmos problemas de memória da Calculadora, mais simples, dada no **Item (A)** e que foram resolvidos no **Item (C)**. Com base na experiência do **Item (C)**, resolver os problemas da interface gráfica, mais elaborada, cujo código foi dado na Figura 1.6.

Exercício 2: Analisar a interface gráfica dada na Figura 2.1 cujo código é dado na Figura 2.2.

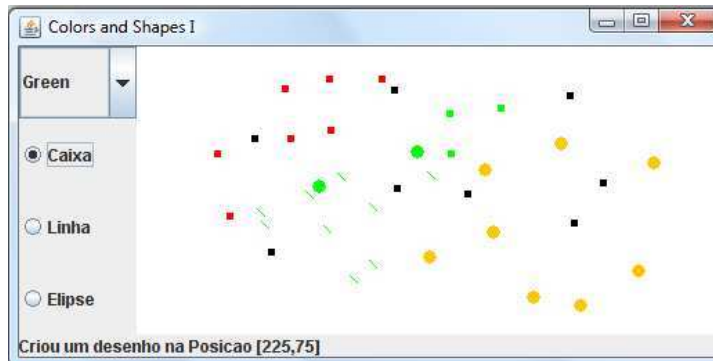


Figura 2.1: Interface gráfica cujo código está na Figura 2.2.

A sua análise deverá ser capaz de responder as seguintes perguntas:

- (i) Quais são os componentes da interface gráfica?
- (ii) Em quantos JPanels a interface gráfica está dividida e qual o papel de cada um deles?
- (iii) Quais são os eventos tratados?
- (iv) Cite os passos para se criar uma forma geométrica simples (caixa, linha ou elipse) na interface gráfica detalhando o trecho de código empregado em cada passo.
- (v) Como apagar os desenhos realizados na interface gráfica?
- (vi) O que ocorre para cada um dos botões do mouse quando o mesmo é clicado sobre a área de desenho?

Além disso, realizar as seguintes tarefas:

Item (A): Como permitir que mais de um item de forma geométrica (caixa, linha ou elipse) possa ser selecionado e depois desenhado na interface?

Item (B): Troque o tratamento de evento de modo que quando o botão direito for clicado é que uma forma geométrica (caixa, linha ou elipse) será desenhada e que quando o botão esquerdo for clicado os desenhos serão apagados.

Item (C): O texto na parte inferior da interface gráfica deverá informar não só a localização do desenho, mas também a cor empregada, bem como a forma escolhida.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class J11E2 extends JFrame implements MouseListener
{
    private JPanel p1, p2, p3;
    private String details; // armazena mensagens.
    private JLabel statusBar; // JLabel que aparece na parte inferior da janela.
    // Definindo um ComboBox.
    private JComboBox box1;
    // Definicao dos nomes a serem empregados no comboBox de cores.
    private final String nomes[] = { "Black", "Blue", "Cyan",
    "Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
    "Orange", "Pink", "Red", "White", "Yellow" };
    // Definicao das cores a serem empregadas no desenho das formas.
    private final Color colors[] = { Color.BLACK, Color.BLUE, Color.CYAN,
    Color.DARK_GRAY, Color.GRAY, Color.GREEN, Color.LIGHT_GRAY,
    Color.MAGENTA, Color.ORANGE, Color.PINK, Color.RED, Color.WHITE,
    Color.YELLOW };
    private JRadioButton radioBtn1, radioBtn2, radioBtn3;
    // Para agrupar os radiosButtons !
    private ButtonGroup bg = new ButtonGroup();
    private int WIDTH = 520;
    private int HEIGHT = 260;

    public J11E2()
    {
        setTitle("Colors and Shapes I");
        setSize(WIDTH, HEIGHT);
        setLocation(250, 250);
        setLayout(new BorderLayout());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        createContents(); // Metodo que adiciona componentes.
        setVisible(true);
    }

    // Método que coordena a criação dos jpanels: controle e desenho.
    public void createContents()
    {
        p1 = createJPanel1();
        p2 = createJPanel2();
        p3 = createJPanel3();
        add(p3);
    }
}
```

```

// Método que constrói o JPanel de controle.
public JPanel createJPanel1()
{
    // Criando o JPanel p1.
    p1 = new JPanel();

    // Definindo um Layout antes de inserir elementos.
    p1.setLayout(new GridLayout(4,1));

    // Criação dos elementos a serem inseridos na interface.
    // Menu de definição de cores.
    box1 = new JComboBox(nomes);
    // Elementos da GUI relativos ao tipo de forma a ser desenhada.
    radioBtn1 = new JRadioButton("Caixa",true);
    radioBtn2 = new JRadioButton("Linha",false);
    radioBtn3 = new JRadioButton("Elipse",false);

    // Agrupando os radiosButtons de modo que ao se selecionar um
    // outro será "deselecionado".
    bg.add(radioBtn1);
    bg.add(radioBtn2);
    bg.add(radioBtn3);

    // Adicionando elementos na interface de controle de
    // criação de Desenhos.
    p1.add(box1);
    p1.add(radioBtn1);
    p1.add(radioBtn2);
    p1.add(radioBtn3);

    return p1;
}

// Método que constrói o JPanel de desenho.
public JPanel createJPanel2()
{
    p2 = new JPanel();
    // Iniciando o background de p2 com WHITE !
    p2.setBackground(colors[11]);

    //-----//
    // Adicionando MouseListener no JPanel2 !! //
    //-----//
    p2.addMouseListener(this);
    //-----//

    return p2;
}

```

```

// Método que reúne os demais JPanels.
public JPanel createJPanel3()
{
    p3 = new JPanel();
    p3.setLayout(new BorderLayout());

    // Definindo o JLabel que exibe mensagens !
    statusBar = new JLabel("Pressione o mouse na area branca !");

    // Adicionando os demais JPanels a p3.
    p3.add(p1,BorderLayout.WEST);
    p3.add(p2,BorderLayout.CENTER);
    p3.add(statusBar,BorderLayout.SOUTH);

    return p3;
}

//-----//
// Metodos para tratar o comportamento do mouse. //
//-----//
// Métodos da interface MouseListener
public void mousePressed(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}

//-----//
// Metodo que trata o evento de clique do mouse. //
//-----//
public void mouseClicked(MouseEvent event)
{
    // Obtendo coordenadas da posicao do mouse na janela.
    int xPos = event.getX();
    int yPos = event.getY();

    // Clicou com o botão direito do mouse: limpar tela.
    if (event.isMetaDown())
    {
        details = String.format("Limpou a tela após clicar na posição [%d,%d]",xPos,yPos);
        // Obtendo o contexto grafico do JPanel p2.
        Graphics g = p2.getGraphics();
        g.setColor(Color.WHITE);
        int largura = p2.getWidth();
        int altura = p2.getHeight();
        // Preenche a area de desenho com um retangulo branco
        // e portanto, "limpa" a area de desenho !
        g.fillRect(0, 0, largura, altura);
    }
}

```

```

// Clicou com o botão do meio: fornece a coordenada do mouse no JPanel.
else if (event.isAltDown())
{details = String.format("Clique com o botao do meio. Posicao
[%d,%d]",xPos,yPos); }

// Clicou com o botão esquerdo do mouse: cria um desenho com a cor
// escolhida.
else
{details = String.format("Criou um desenho na Posicao
[%d,%d]",xPos,yPos);

// Determinando qual a cor do desenho.
int ci = box1.getSelectedIndex();

// Obtendo o contexto gráfico do JPainel p2.
Graphics g = p2.getGraphics();
// Empregando a cor selecionado no JComboBox.
g.setColor(colors[ci]);

// Usando a mesma largura e altura para todos os desenhos.
int largura = 5;
int altura = 5;

// Determinando qual desenho foi escolhido.
// Verificando se o radio button da Caixa foi escolhida.
if (radioBtn1.isSelected())
// Desenho de um retangulo.
g.fillRect(xPos, yPos, largura, altura);

// Verificando se o radio button da Linha foi escolhida.
else if (radioBtn2.isSelected())
g.drawLine(xPos, yPos, largura+xPos, altura+yPos);

// Verificando se o radio button da Elipse foi escolhida.
else if (radioBtn3.isSelected())
// Desenho de uma elipse.
g.fillOval(xPos, yPos, 2*largura, 2*altura);
}
// Atualizando o JLabel que mostra mensagens de ação !
statusBar.setText(details);
}
//-----//
public static void main(String args[])
{
J11E2 janela1 = new J11E2();
}
} // Final da classe.

```

Figura 2.2: Código Java da interface gráfica 2.1.

Exercício 3: Analisar a interface gráfica dada na Figura 3.1 cujo código é dado na Figura 3.2.

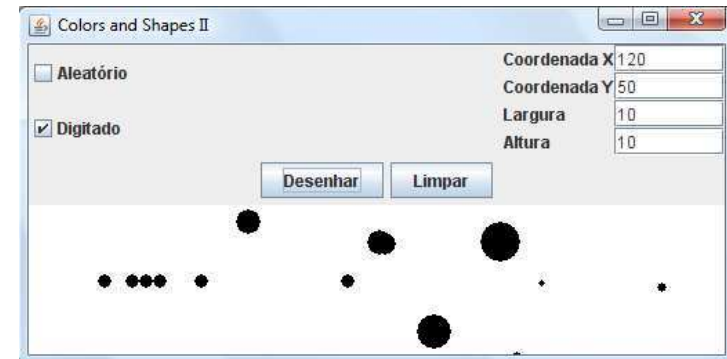


Figura 3.1: Interface gráfica cujo código está na Figura 3.2.

A sua análise deverá ser capaz de responder as seguintes perguntas:

- (i) Quais são os componentes da interface gráfica?
- (ii) Em quantos JPanels a interface gráfica está dividida e qual o papel de cada um deles?
- (iii) Quais são os eventos tratados?
- (iv) Cite os passos para se criar um desenho na interface gráfica detalhando o trecho de código empregado em cada passo.
- (v) Como apagar os desenhos realizados na interface gráfica?
- (vi) O que ocorre quando a caixa de texto selecionada é aleatório? E digitado?

Além disso, realizar as seguintes tarefas:

Item (A): Como permitir que mais de um item de uma forma de desenho (Aleatório ou Digitado) possa ser selecionado e depois desenhado na interface?

Item (B): Insira um novo tratamento de evento de modo que quando o usuário clicar enter em qualquer JTextField automaticamente a forma selecionada será desenhada.

Item (C): Inserir um terceiro botão que desenha retângulos. Para tanto, empregue o seguinte código:

```

// Desenho de um retangulo.
g.fillRect(xPos, yPos, largura, altura);

```

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.Random;

public class J11E3 extends JFrame implements ActionListener
{
    private JPanel p1, p2, p3;
    private JLabel label1, label2, label3, label4;
    private JTextField field1, field2, field3, field4;
    private JButton btn1, btn2; // botões: desenhar ou apagar desenhos.
    private JLabel statusBar; // JLabel que aparece na parte inferior da janela.
    private JCheckBox checkBtn1, checkBtn2; // Definir coordenadas ou geração
    aleatória.
    private ButtonGroup bg = new ButtonGroup(); // Para agrupar os
    checkButtons !
    private int WIDTH = 520;
    private int HEIGHT = 260;

    public J11E3()
    {
        setTitle("Colors and Shapes II");
        setSize(WIDTH,HEIGHT);
        setLocation(250,250);
        setLayout(new BorderLayout());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        createContents(); // Metodo que adiciona componentes.
        setVisible(true);
    }

    // Método que coordena a criação dos jpanels: controle e desenho.
    public void createContents()
    {
        p1 = createJPanel1();
        p2 = createJPanel2();
        p3 = createJPanel3();
        add(p3);
    }

    // Método que constrói o JPanel de controle.
    public JPanel createJPanel1()
    {
        // Criando o JPanel p1.
        p1 = new JPanel();

        // Definindo um Layout antes de inserir elementos.
        p1.setLayout(new BorderLayout());

```

```

        // Criando um novo JPanel onde colocar escolha
        // sobre uso aleatorio ou digitado das coordenadas.
        // Criando espaco para escolher tipo coordenadas.
        checkBtn1 = new JCheckBox("Aleatório",true);
        checkBtn2 = new JCheckBox("Digitado",false);

        // Adicionando "Listener" ao CheckBoxes para permitir
        // que os campos de dados com as coordenadas do desenho
        // se tornem "editáveis" ou não.
        checkBtn1.addActionListener(this);
        checkBtn2.addActionListener(this);

        // Agrupando os radiosButtons de modo que ao se selecionar um
        // outro será "deselecionado".
        bg.add(checkBtn1);
        bg.add(checkBtn2);

        // Criando um novo JPanel dentro de p1.
        JPanel p1a = new JPanel();
        p1a.setLayout(new GridLayout(2,1));
        p1a.add(checkBtn1);
        p1a.add(checkBtn2);

        // Criando demais elementos de p1.
        label1 = new JLabel("Coordenada X");
        label2 = new JLabel("Coordenada Y");
        label3 = new JLabel("Largura");
        label4 = new JLabel("Altura ");
        field1 = new JTextField("0");
        field2 = new JTextField("0");
        field3 = new JTextField("10");
        field4 = new JTextField("10");

        // Definindo inicialmente editable false
        // para os campos onde as coordenadas x
        // e y podem ser inseridas.
        field1.setEditable(false);
        field2.setEditable(false);
        field3.setEditable(false);
        field4.setEditable(false);

        // Outro JPanel dentro de p1.
        JPanel p1b = new JPanel();
        p1b.setLayout(new GridLayout(4,4));
        p1b.add(label1);
        p1b.add(field1);
        p1b.add(label2);

```

```

p1b.add(field2);
p1b.add(label3);
p1b.add(field3);
p1b.add(label4);
p1b.add(field4);

// JPanel com botoes dentro de p1.
btn1 = new JButton("Desenhar");
btn2 = new JButton("Limpar");

//-----//
// Adicionando "Listeners" ao botões. //
//-----//
btn1.addActionListener(this);
btn2.addActionListener(this);
//-----//

// Criando o terceiro JPanel dentro de p1.
JPanel p1c = new JPanel();
p1c.setLayout(new FlowLayout());
p1c.add(btn1);
p1c.add(btn2);

// Adicionando elementos na interface de controle de
// criação de Desenhos.
p1.add(p1a, BorderLayout.WEST);
p1.add(p1b, BorderLayout.EAST);
p1.add(p1c, BorderLayout.SOUTH);

return p1;
}

// Método que constrói o JPanel de desenho.
public JPanel createJPanel2()
{
    p2 = new JPanel();
    // Iniciando o background de p2 com WHITE !
    p2.setBackground(Color.WHITE);

    return p2;
}

// Método que reúne os demais JPanels.
public JPanel createJPanel3()
{
    p3 = new JPanel();
    p3.setLayout(new BorderLayout());

```

```

// Adicionando os demais JPanels a p3.
p3.add(p1, BorderLayout.NORTH);
p3.add(p2, BorderLayout.CENTER);
// p3.add(statusBar, BorderLayout.SOUTH);

return p3;
}

//-----//
//-----//
//-----//
// Metodos para tratar o comportamento dos botoes. //
//-----//
//-----//
//-----//
public void actionPerformed(ActionEvent e)
{
    //-----//
    // O botão desenhar foi apertado. //
    //-----//
    if (e.getSource() == btn1)
    {
        // Criando as variaveis necessarias para se fazer
        // o desenho.
        int x = 0;
        int y = 0;
        int largura = 0;
        int altura = 0;

        // Determinando se usara coordenada aleatorio ou
        // dada pelo usuario.
        // Verificando se o checkbox de coordenadas definidas
        // foi escolhido.
        if (checkBtn2.isSelected())
        {
            x = Integer.parseInt(field1.getText());
            y = Integer.parseInt(field2.getText());
            try
            {
                largura = Integer.parseInt(field3.getText());
                altura = Integer.parseInt(field4.getText());
            }
            catch (NumberFormatException ex)
            {
                largura = 10;
                altura = 10;
            }
        }
    }
}

```

```

        // Caixa de texto apropriada.
        JOptionPane.showMessageDialog(null,
        "Dados inválidos. Usando largura = " + largura + " e altura = " + altura,
        "Aviso",JOptionPane.WARNING_MESSAGE);
    }
}
// Gerar aleatoriamente os dados.
else
{
    Random r = new Random();
    // Para gerar numeros inteiros aleatórios em [0, WIDTH].
    int a = 0;
    int b = p2.getWidth();
    x = r.nextInt(b-a+1) + a; // Gera valores em [a,b].
    b = p2.getHeight();
    y = r.nextInt(b-a+1) + a;
    // Gera valores em [0,30].
    b = 30;
    largura = r.nextInt(b-a+1) + a;
    altura = largura;
}

// Construindo o desenho com os dados fornecidos.
// Obtendo o contexto gráfico do JPainel p2.
Graphics g = p2.getGraphics();
// Desenho de uma elipse.
g.fillOval(x, y, largura, altura);

}

//-----//

//-----//
// O botão limpar foi apertado. //
//-----//
if (e.getSource() == btn2)
{
    // Obtendo o contexto grafico do JPanel p2.
    Graphics g = p2.getGraphics();
    g.setColor(Color.WHITE);
    int largura = p2.getWidth();
    int altura = p2.getHeight();
    // Preenche a area de desenho com um retangulo branco
    // e portanto, "limpa" a area de desenho !
    g.fillRect(0, 0, largura, altura);
}
//-----//

```

```

//-----//
// O CheckBox de aleatorio foi apertado. //
//-----//
if (e.getSource() == checkBtn1)
{
    // Definindo inicialmente editable false
    // para os campos onde as coordenadas x
    // e y podem ser inseridas.
    field1.setEditable(false);
    field2.setEditable(false);
    field3.setEditable(false);
    field4.setEditable(false);
}
//-----//

//-----//
// O CheckBox de dados digitados foi apertado. //
//-----//
if (e.getSource() == checkBtn2)
{
    // Definindo inicialmente editable false
    // para os campos onde as coordenadas x
    // e y podem ser inseridas.
    field1.setEditable(true);
    field2.setEditable(true);
    field3.setEditable(true);
    field4.setEditable(true);
}
//-----//

}

//-----//

//-----//

public static void main(String args[])
{
    J11E3 janela1 = new J11E3();
}
}

```

Figura 3.2: Código Java da interface gráfica 3.1.

Exercício 4: Analisar a interface gráfica dada na Figura 4.1 cujo código é dado na Figura 4.2.

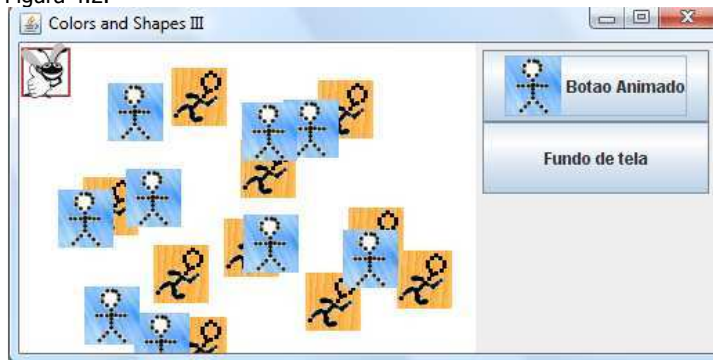


Figura 4.1: Interface gráfica cujo código está na Figura 4.2.

A sua análise deverá ser capaz de responder as seguintes perguntas:

- (i) Quais são os componentes da interface gráfica?
- (ii) Em quantos JPanels a interface gráfica está dividida e qual o papel de cada um deles?
- (iii) Quais são os eventos tratados?
- (iv) Cite os passos para se criar um desenho na interface gráfica detalhando o trecho de código empregado em cada passo.
- (v) Como mudar o fundo dos desenhos realizados na interface gráfica?
- (vi) o mouse é passado sobre o botão animado? E quando o botão animado é apertado?

Além disso, realizar as seguintes tarefas:

Item (A): Como permitir que mais de um item de uma forma de desenho (Aleatório ou Digitado) possa ser selecionado e depois desenhado na interface?

Item (B): Insira um novo tratamento de evento de modo que quando o usuário posicionar o mouse sobre o botão "Botão Animado" automaticamente irá aparecer a figura "bug1.gif" no botão "Fundo de Tela". Caso o mouse seja posicionado sobre o botão Fundo de Tela irá aparecer a figura "bug2.gif" no botão "Fundo de Tela". Os arquivos de figura são dados na Tabela 4.1.



(A) Arquivo "people1.gif".



(B) Arquivo "people2.gif".



(C) Arquivo "bug1.gif".



(D) Arquivo "bug2.gif".

Tabela 4.1: Arquivos de figuras a serem empregadas na interface.

```
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.MouseListener;
import java.awt.event.ActionEvent;
import java.awt.event.MouseEvent;
import javax.swing.*;
import java.io.*;
import java.util.Scanner;
import java.awt.image.*;
import javax.imageio.*;
import java.net.URL;
import javax.swing.ImageIcon;

//import java.io.*;
//import java.util.Scanner;
//import java.awt.image.*;
//import javax.imageio.*;
```

```
public class J11E4 extends JFrame implements ActionListener, MouseListener
{
```

```
    private JPanel p1, p2, p3;
    private JButton btn1, btn2;    // botões: desenhar ou apagar desenhos.
    private boolean pressed = false; // Variavel que controla se o botao
                                    // de desenho especial foi apertado !
```

```
    // Imagem de fundo.
    private BufferedImage img = null;
```

```
    private int WIDTH  = 520;
    private int HEIGHT = 260;
```

```
    public J11E4()
    {
        setTitle("Colors and Shapes III");
        setSize(WIDTH,HEIGHT);
        setLocation(250,250);
        setLayout(new BorderLayout());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        createContents(); // Metodo que adiciona componentes.
        setVisible(true);
    }
```

```
    // Método que coordena a criação dos jpanels: controle e desenho.
```

```
    public void createContents()
    {
        p1 = createJPanel1();
        p2 = createJPanel2();
```

```

        p3 = createJPanel3();
        add(p3);
    }

    // Método que constrói o JPanel de controle.
    public JPanel createJPanel1()
    {
        // Criando o JPanel p1.
        p1 = new JPanel();

        // Definindo um Layout antes de inserir elementos.
        p1.setLayout(new FlowLayout());

        // Criando um novo JPanel dentro de p1.
        JPanel p1a = new JPanel();
        p1a.setLayout(new GridLayout(2,1));

        // Criando um botao que determina se um desenho
        // esta ligado ou desligado.
        btn1 = new JButton("");
        Icon bug1 = new ImageIcon( getClass().getResource( "people1.gif" ) );
        Icon bug2 = new ImageIcon( getClass().getResource( "people2.gif" ) );
        btn1 = new JButton("Botao Animado",bug1); // Configura imagem.
        btn1.setRolloverIcon( bug2 ); // Configura imagem de rollover.

        // Criando um botao que recupera imagens do computador
        // e carrega como fundo de tela.
        btn2 = new JButton("Fundo de tela");

        p1a.add(btn1);      // Adiciona o botao no JPanel.
        p1a.add(btn2);      // Adiciona o botao no JPanel.

        // Cria novo Listener para tratamento de evento de botao.
        btn1.addActionListener( this );
        btn2.addActionListener( this );

        // Adicionando elementos na interface de controle de
        // criação de Desenhos.
        p1.add(p1a, BorderLayout.WEST);
        //p1.add(p1b, BorderLayout.EAST);
        //p1.add(p1c, BorderLayout.SOUTH);

        return p1;
    }

    // Método que constrói o JPanel de desenho.

```

```

public JPanel createJPanel2()
{
    p2 = new JPanel();
    // Iniciando o background de p2 com WHITE !
    p2.setBackground(Color.WHITE);

    //-----//
    // Adicionando MouseListener no JPanel2 !! //
    //-----//
    p2.addMouseListener(this);
    //-----//

    return p2;
}

// Método que reúne os demais JPannels.
public JPanel createJPanel3()
{
    p3 = new JPanel();
    p3.setLayout(new BorderLayout());

    // Adicionando os demais JPannels a p3.
    p3.add(p1, BorderLayout.EAST);
    p3.add(p2, BorderLayout.CENTER);
    // p3.add(statusBar, BorderLayout.SOUTH);

    return p3;
}

//-----//
//-----//
//-----//
// Metodos para tratar o comportamento dos botoes. //
//-----//
//-----//
//-----//

// Trata de evento de botao.
public void actionPerformed( ActionEvent event )
{
    // Verificando se o botao 1 foi pressionado
    if (event.getSource() == btn1)
    {
        String mensagem = "GIMP:www.baixaki.com.br/download/the-gimp.htm";
        JOptionPane.showMessageDialog(this, String.format("Apertou: %s. \n %s",
        event.getActionCommand(),mensagem));
    }
}

```



```

// Variavel que indica que o botao foi apertado e que portanto
// havera uma troca de qual imagem ficara fixa e qual ficara
// variavel.
Icon bug1 = new ImageIcon( getClass().getResource( "people1.gif" ) );
Icon bug2 = new ImageIcon( getClass().getResource( "people2.gif" ) );

// Se o estado da variavel indica que o botao esta no estado
// inicial, entao, indicar inversao de estado.
if (!pressed)
{
    pressed = true;
    btn1.setIcon(bug2);          // Configura imagem.
    btn1.setRolloverIcon( bug1 ); // Configura imagem de rollover.
}
else
{
    pressed = false;
    btn1.setIcon(bug1);          // Configura imagem.
    btn1.setRolloverIcon( bug2 ); // Configura imagem de rollover.
}
}

//-----//
// Verifica se o botão para colocar imagem de fundo foi acionado.
//-----//

if (event.getSource() == btn2)
{
    // Itens para tratar a abertura e leitura dos dados de um arquivo
    texto.
    File fileDir; // Diretorio ou arquivo especificado pelo usuário.
    JFileChooser chooser = new JFileChooser(".");
    int response; // resposta do usuario na GUI.
    String output = ""; // lista de arquivos com tamanhos.

    // Abrindo painel grafico para escolha de arquivos
    chooser.setSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
    response = chooser.showOpenDialog(null);

    // Verificando se o usuario selecionou pressionou o botao Open.
    if (response == JFileChooser.APPROVE_OPTION)
    {
        fileDir = chooser.getSelectedFile();
        // Verificando se um arquivo foi selecionado.
        if (fileDir.isFile())
        {
            // Tentando definir uma imagem como figura de fundo.

```

```

try
{
    img = ImageIO.read(new File(fileDir.getName()));
    output = "Dados recuperados com sucesso !";
}
// Mensagem de erro caso nao seja possivel.
catch (IOException ev)
{
    // JOptionPane.showMessageDialog(null, "Arquivo não encontrado!",
    // "File Sizes", JOptionPane.INFORMATION_MESSAGE);
}
}
// Verificando se um diretorio foi selecionado.
else if (fileDir.isDirectory())
{
    output = "Diretório Selecionado. Selecione um arquivo";
} // end else.
// Caso em que nem um arquivo nem um diretorio foram selecionados.
else
{
    output = "Escolha inválida. Não é diretório nem arquivo.";
}

// Caixa de mensagem que indica o que ocorreu.
JOptionPane.showMessageDialog(null, output,
    "Recuperação de Dados",JOptionPane.INFORMATION_MESSAGE);

// Redesenhar a tela em funcao da mudanca da imagem de fundo.
Graphics g = p2.getGraphics();
int width = p2.getWidth();
int height = p2.getHeight();
g.clearRect(0,0,width,height);
// Obtendo as cores de fundo do que o usuario
// escolheu na interface grafica ao inves de
// fixar a cor branca como padrao de fundo !!
g.setColor(Color.white);
// Previnir chamada da funcao quando o construtor
// ainda esta desenhando a interface.
if (g != null)
{
    g.fillRect(0,0,width,height);
    g.drawImage(img, 0, 0, null); // desenha a imagem original.
}
} // Fim do if do JFileChooser.
}
//-----//

```

```

} // Fim do metodo actionPerformed.

//-----//

//-----//
//-----//
//-----//
// Metodos para tratar o comportamento do mouse. //
//-----//
//-----//
//-----//
// Métodos da interface MouseListener
public void mousePressed(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}

//-----//
// Metodo que trata o evento de clique do mouse. //
//-----//
public void mouseClicked(MouseEvent event)
{
    // Obtendo coordenadas da posicao do mouse na janela.
    int x = event.getX();
    int y = event.getY();

    // Clicou com o botão direito do mouse: limpar tela.
    if (event.isMetaDown())
    {

    }

    // Clicou com o botão do meio: fornece a coordenada do mouse no JPanel.
    else if (event.isAltDown())
    {

    }

    // Clicou com o botão esquerdo do mouse: cria um desenho com a cor
    escolhida.
    else
    {

        // Obtendo o contexto gráfico do JPainel p2.
        Graphics g = p2.getGraphics();
        // Definindo qual imagem será construída.

```

```

// Para tanto, verifica-se o estado atual
// do botao 1.
// Estado inicial. Criar imagem azul.
ImageIcon image = new ImageIcon("people1.gif");
// Segundo estado. Criar imagem amarela.
if (pressed == true)
    image = new ImageIcon("people2.gif");

// Desenhando a imagem nas coordenadas x e y.
image.paintIcon(this, g, x, y);

}

}

//-----//

public static void main( String args[] )
{
    J11E4 f1 = new J11E4(); // Cria uma interface Inter2
}

}

```

Figura 4.2: Código Java da interface gráfica 4.1.

Exercício 5: Reunir as três interfaces definidas nos Exercícios 2, 3 e 4 em uma única interface gráfica.

Exercício 1: Criar uma interface gráfica tal que reproduz o Jogo do Caos. O jogo do caos foi elaborado por Bansley e funciona assim:

Passo (1): Crie um triângulo cujos vértices são indexados como ilustrado na Figura 1.1.

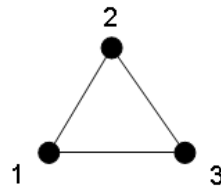


Figura 1.1: Triângulo cujas coordenadas dos vértices são dadas na Tabela 1.1.

Vértice	X	Y
1	0	0
2	WIDTH/2	HEIGHT/2
3	WIDTH	0

Tabela 1.1: Coordenadas (x, y) dos vértices do triângulo.

Passo (2): Gere um ponto inicial P0 de coordenadas (x0, y0) aleatórias tal que $x0 \in [0, WIDTH]$ e $y0 \in [0, HEIGHT]$. Desenhe este ponto tal como dado na Figura 1.2.

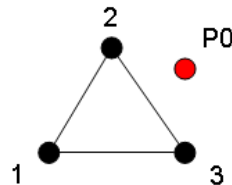


Figura 1.2: Desenhando o ponto inicial aleatório P0.

Passo 3: Selecione aleatoriamente, um dos três vértices. Crie um segmento de reta que une P0 a um dos vértices selecionados aleatoriamente. Depois, obtenha o ponto médio PM1 deste segmento de reta. Na Figura 1.3, o vértice escolhido foi o 1.

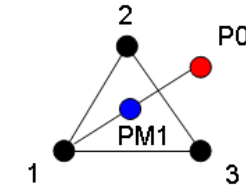


Figura 1.3: Desenhando o ponto médio PM1.

Passo 4: Realize um novo sorteio entre um dos três vértices do triângulo. Depois, a partir do ponto PM1, trace um segmento de reta até o ponto escolhido e obtenha o ponto médio do mesmo. Na Figura 1.4, por exemplo, o vértice 2 foi sorteado.

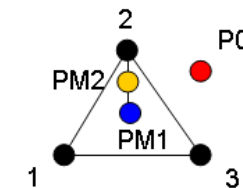


Figura 1.4: Desenhando o ponto médio PM2.

Passo 5: Repetir o processo de obtenção dos pontos médios até um número máximo de pontos médios.

Item (A) Qual será a figura resultante? Você espera que a figura tenha algum padrão? Para descobrir use o código dado na Figura 1.5.

Classe E12J1

```
import javax.swing.JFrame;
import java.awt.Graphics;
import javax.swing.JPanel;
import javax.swing.JOptionPane;
import java.util.Random;
import java.awt.Color;

public class J12E1 extends JFrame
{
    private JPanel p1; // Local onde sera construido o
                      // desenho resultante do Jogo do
                      // Caos.

    // Dimensoes iniciais do JFrame.
    private int WIDTH = 720;
    private int HEIGHT = 460;
```

```

public J12E1()
{
    setTitle("Jogo do Caos I");
    setSize(WIDTH,HEIGHT);
    setLocation(250,250);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents(); // Metodo que adiciona componentes.
    setVisible(true);
}

// Método que a criação do jpanel: desenho resultante.
public void createContents()
{
    p1 = createJPanel1();
    add(p1);
}

// Método que constrói o JPanel de controle.
public JPanel createJPanel1()
{
    // Criando o JPanel p1.
    p1 = new JPanel();

    // Retorna o JPanel p1 criado vazio.
    return p1;
}

public void desenharJogo()
{
    // Construindo o desenho com os dados fornecidos.
    // Obtendo o contexto gráfico do JPainel p1.
    Graphics g = p1.getGraphics();

    // Definindo a cor de fundo.
    p1.setBackground(Color.WHITE);

    // Criando variavel auxiliar para gerar numeros aleatorios.
    Random rand =new Random();

    // Gerando aleatoriamente um ponto inicial.
    int x=rand.nextInt(WIDTH+1);
    int y=rand.nextInt(HEIGHT+1);
    int vertice=0;

    // Perguntar ao usuario quantos pontos desenhar.
    String input=JOptionPane.showInputDialog(
        "Insira o número de iteracoes Desejada");
}

```

```

// Numero padrao de pontos.
int Npontos = 5000;

// Verificar se o numero digitado eh valido.
try
{
    Npontos = Integer.parseInt(input);
}
// Se input nao for valido assumir valor padrao.
catch(NumberFormatException e)
{
    // Mostrando o valor padrao que sera empregado.
    input=JOptionPane.showInputDialog(
        "Numero de pontos: " + Npontos);
}

// Laco para desenhar os pontos input vezes
for (int i=0;i<Npontos;i++)
{
    // Realizando sorteio dos 3 vertices.
    vertice=1+rand.nextInt(3);

    // Verificando qual vertice foi sorteado.
    switch (vertice)
    {
        // Vertice (0,0).
        case 1:
            // Ajuste para escala do java.
            //x=((WIDTH/2)+x)/2;
            //y=y/2;
            x = x + (0 - x) /2;
            y = y + (0 - y) /2;
            g.setColor(Color.GREEN);
            break;
        // Vertice (WIDTH/2, HEIGHT/2).
        case 2:
            // Ajuste para escala do java.
            //x=(WIDTH+x)/2;
            //y=(HEIGHT+y)/2;
            x = x + (WIDTH/2 - x) /2;
            y = y + (HEIGHT/2 - y) /2;
            g.setColor(Color.RED);
            break;
        // Vertice (WIDTH, 0).
        case 3:
            // Ajuste para escala do java.
            //x=x/2;
            //y=(HEIGHT+y)/2

```

```

        x = x + (WIDTH - x) / 2;
        y = y + (0 - y) / 2;;
        g.setColor(Color.BLUE);
        break;
    }
    // Desenho de uma elipse.
    g.fillOval(x, y, 1, 1);
}

public static void main(String args[])
{
    J12E1 janela1 = new J12E1();
    janela1.desenharJogo();
}

```

Figura 1.5 : Código Java para a interface gráfica da Jogo do Caos.

Item (B): Tire os comentários relativos ao ajuste de escala do Java que existem no cálculo das coordenadas dos pontos x e y. Verifique se o desenho gerado será como dado na Figura 1.6.

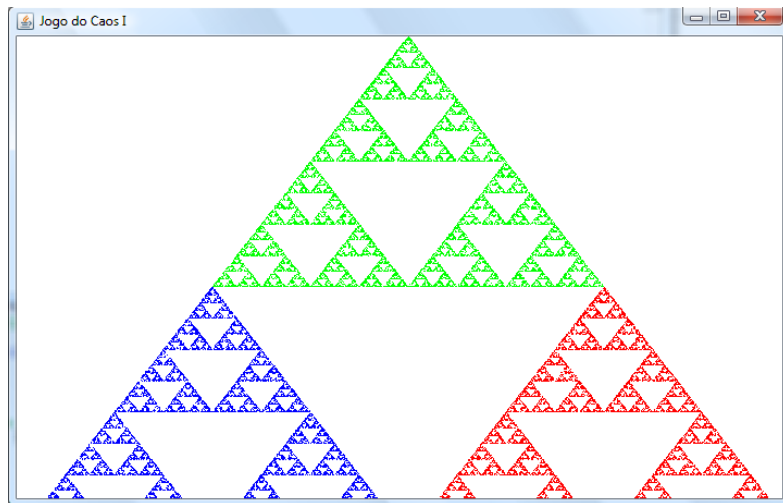


Figura 1.6 : Figura obtido com 50000 pontos para o Jogo do Caos.

Exercício 2: Criar uma interface gráfica que permita definir, para o Jogo do Caos, a cor do desenho, bem como o número de pontos a serem usados. Tal interface gráfica deve ter a aparência dada na Figura 2.1.

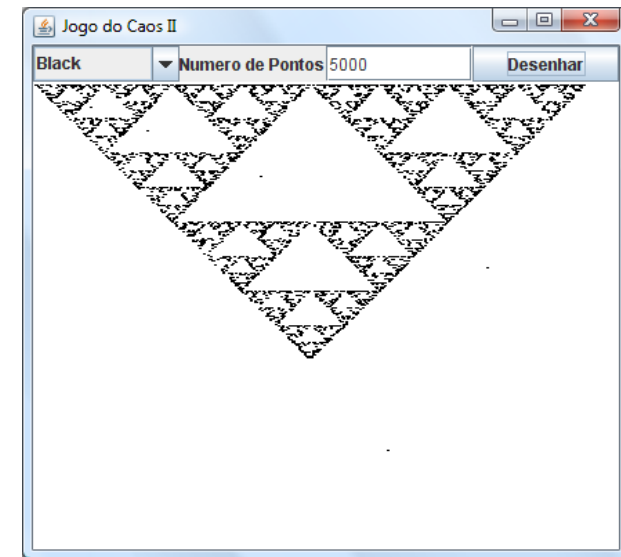


Figura 2.1: Nova interface gráfica para o Jogo do Caos.

Além disso, observe que para a interface anterior, se a janela que contém a figura fosse minimizada, então, a mesma seria apagada. Para evitar tal problema é necessário criar uma nova classe JFractalPanel que especializa o JPanel e, em particular, redefine o método paintComponent. Assim, se a interface gráfica for minimizada e depois maximizada (ou seja, redesenhada), o método paintComponent será chamado e se contiver instruções de como desenhar a figura, a mesma será, também redesenhada. Para tanto, use o código contido nas Figuras 2.2 e 2.3.

```

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JComboBox;
import java.awt.GridLayout;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.*;

```

```

public class J12E2 extends JFrame implements ActionListener
{
    private JFractalPanel p1;
    private JPanel p2, p3;
    // Definindo um JButton.
    private JButton btn1;
    // Definindo um JLabel.
    JLabel l1;
    // Definindo JTextField.
    TextField txt1;
    // Definindo um ComboBox.
    private JComboBox box1;
    // Definicao dos nomes a serem empregados no comboBox de cores.
    private final String nomes[] = { "Black", "Blue", "Cyan",
    "Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
    "Orange", "Pink", "Red", "White", "Yellow"};
    // Definicao das cores a serem empregadas no desenho das formas.
    private final Color colors[] = { Color.BLACK, Color.BLUE, Color.CYAN,
    Color.DARK_GRAY, Color.GRAY, Color.GREEN, Color.LIGHT_GRAY,
    Color.MAGENTA, Color.ORANGE, Color.PINK, Color.RED, Color.WHITE,
    Color.YELLOW };

    // Dimensoes iniciais do JFrame.
    private int WIDTH = 440;
    private int HEIGHT = 400;

    public J12E2()
    {
        setTitle("Jogo do Caos II");
        setSize(WIDTH,HEIGHT);
        setLocation(250,250);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        createContents(); // Metodo que adiciona componentes.
        setVisible(true);
    }

    // Método que a criação do jpanel: desenho resultante.
    public void createContents()
    {
        // Criando um JPanel de controle dos desenhos.
        p2 = createJPanel2();

        // Criando uma especializacao de JPanel.
        p1 = new JFractalPanel(0);

        // Organizando p1 e p2 dentro de p3.
        p3 = createJPanel3();

        // Adiciona o JPanel p3 na janela.
        add(p3);
    }
}

```

```

// Método que constrói o JPanel de controle.
public JPanel createJPanel2()
{
    // Criando o JPanel p2.
    p2 = new JPanel();

    // Definindo um Layout antes de inserir elementos.
    p2.setLayout(new GridLayout(1,4));

    // Criação dos elementos a serem inseridos na interface.
    // Menu de definição de cores.
    box1 = new JComboBox(nomes);

    // Criando JLabel e campo de texto para numero de pontos.
    l1 = new JLabel("Numero de Pontos");
    txt1 = new JTextField("0");

    // Criando botao.
    btn1 = new JButton("Desenhar");

    // Adicionando elementos na interface de controle de
    // criação de Desenhos.
    p2.add(box1);
    p2.add(l1);
    p2.add(txt1);
    // Adicionando evento ao botao.
    btn1.addActionListener(this);
    p2.add(btn1);

    return p2;
}

// Método que constrói o JPanel que engloba o
// controle e o desenho de Fractais.
public JPanel createJPanel3()
{
    // Criando o JPanel p3 com BorderLayout.
    p3 = new JPanel();
    p3.setLayout(new BorderLayout());

    // Adicionando os demais JPanels a p3.
    p3.add(p1,BorderLayout.CENTER);
    p3.add(p2,BorderLayout.NORTH);

    return p3;
}

```

```

//-----//
//-----//
//-----//
// Metodos para tratar o comportamento dos botoes. //
//-----//
//-----//
//-----//
public void actionPerformed(ActionEvent e)
{
    //-----//
    // O botão desenhar foi apertado.           //
    //-----//
    if (e.getSource() == btn1)
    {
        // Obtendo os dados da interface para criar o desenho.
        int c  = box1.getSelectedIndex();
        Color cor = colors[c];
        int n = Integer.parseInt(txt1.getText());

        // Redesenhando usando parametros atualizados.
        p1.setCor(cor);
        p1.setNpontos(n);
        // Redesenhando toda a interface.
        repaint();

    }
}

//-----//

public static void main(String args[])
{
    J12E2 janela1 = new J12E2();
}
}

```

Figura 2.2: Código da classe J12E2 que define o painel de controle do Jogo do Caos (**JComboBox**, **JLabel**, **TextField** e **Button**).

```

// JFractalPanel demonstra desenho do Jogo do Caos.
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Dimension;
import javax.swing.JPanel;
import java.util.Random;
public class JFractalPanel extends JPanel

```

```

{
    // Supondo valores padrao.
    private int Npontos = 0;
    private Color color = Color.RED;
    private final int WIDTH = 400; // define a largura do JPanel
    private final int HEIGHT = 400; // define a altura do JPanel

    // configura o nível do fractal inicial com o valor especificado
    // e configura as especificações do JPanel
    public JFractalPanel( )
    {
        // Chamada ao construtor sem parametros.
        this(0);
    } // fim do construtor JFractalPanel.

    // configura o nível do fractal inicial com o valor especificado
    // e configura as especificações do JPanel
    public JFractalPanel( int currentLevel )
    {
        setBackground( Color.WHITE );
        setPreferredSize( new Dimension( WIDTH, HEIGHT ) );
        setNpontos(currentLevel);
    } // fim do construtor FractalJPanel

    // Definindo o numero de pontos a ser usado no desenho.
    public void setNpontos(int np)
    {
        Npontos = (np>=0)?np:0;
    }

    // Obtendo o numero de pontos a ser usado no desenho.
    public int getNpontos()
    {
        return Npontos;
    }

    // Definindo a cor a ser usado no desenho.
    public void setCor(Color color)
    {
        this.color = color;
    }

    // Obtendo a cor a ser usado no desenho.
    public Color getCor()
    {
        return color;
    }
    // Retorna a dimensao WIDTH.
}

```

```

public int getWidth()
{return WIDTH;}

// Retorna a dimensao HEIGHT.
public int getHeight()
{return HEIGHT;}

// inicia o desenho de fractal
public void paintComponent( Graphics g )
{
    super.paintComponent( g );

    // desenha o padrão de fractal
    g.setColor( color );
    drawFractal( g );
} // fim do método paintComponent

public void drawFractal(Graphics g)
{
    // Preenche a area de desenho com um retangulo branco
    // e portanto, "limpa" a area de desenho !
    g.setColor(Color.WHITE);
    g.fillRect(0, 0, WIDTH, HEIGHT);

    // Definindo a cor a ser usada no desenho.
    g.setColor(color);

    // Criando variavel auxiliar para gerar numeros aleatorios.
    Random rand =new Random();

    // Gerando aleatoriamente um ponto inicial.
    int x=rand.nextInt(WIDTH+1);
    int y=rand.nextInt(HEIGHT+1);
    int vertice=0;

    // Laco para desenhar os pontos input vezes
    for (int i=0;i<Npontos;i++)
    {
        // Realizando sorteio dos 3 vertices.
        vertice=1+rand.nextInt(3);

        // Verificando qual vertice foi sorteado.
        switch (vertice)
        {
            // Vertice (0,0).
            case 1:
                // Ajuste para escala do java.
                //x=((WIDTH/2)+x)/2;

```

```

                //y=y/2;
                x = x + (0 - x) /2;
                y = y + (0 - y) /2;
                break;
            // Vertice (WIDTH/2, HEIGHT/2).
            case 2:
                // Ajuste para escala do java.
                //x=(WIDTH+x)/2;
                //y=(HEIGHT+y)/2;
                x = x + (WIDTH/2 - x) /2;
                y = y + (HEIGHT/2 - y) /2;
                break;
            // Vertice (WIDTH, 0).
            case 3:
                // Ajuste para escala do java.
                //x=x/2;
                //y=(HEIGHT+y)/2
                x = x + (WIDTH - x) /2;
                y = y + (0 - y) /2;;
                break;
        }

        // Desenho de uma elipse.
        g.fillOval(x, y, 2, 2);
    }
} // fim da classe JFractalPanel.

```

Figura 2.3: Código da classe **JFractalPanel** que especializa um **JPanel** de modo que se for necessário redesenhar uma janela, o **JPanel** será redesenhado e construíra a figura do Jogo do Caos.

Exercício 3: Criar uma interface gráfica que permita definir, para o Jogo do Caos, empregando um Menu de opções de cores tal como dado na Figura 3.1.

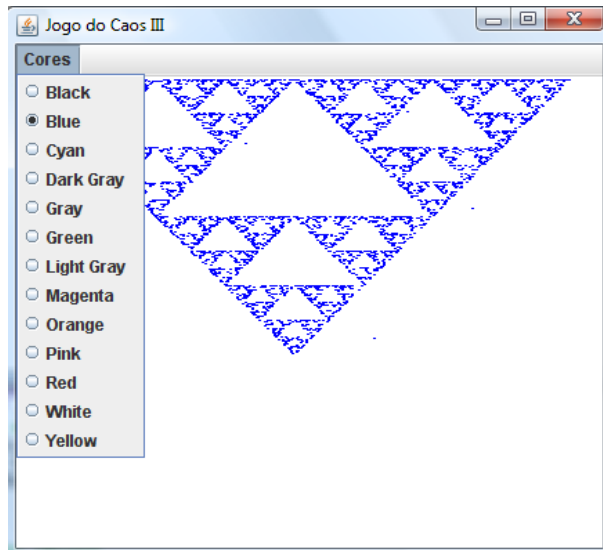


Figura 3.1: Nova interface gráfica para o Jogo do Caos que usa JMenu.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class J12E3 extends JFrame implements ActionListener
{
    private JFractalPanel p1;
    private JPanel p2, p3;
    // Definindo elementos para criar menu.
    private JMenuBar barra;
    private JMenu popup;
    private ButtonGroup corGrupo;
    private JRadioButtonMenuItem item[];

    // Definicao do numero de pontos a serem usados.
    private String pontos[] = {"1000", "5000", "10000", "50000"};
    // Definicao dos nomes a serem empregados no comboBox de cores.
    private final String nomes[] = { "Black", "Blue", "Cyan",
    "Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
    "Orange", "Pink", "Red", "White", "Yellow" };
    // Definicao das cores a serem empregadas no desenho das formas.
    private final Color colors[] = { Color.BLACK, Color.BLUE, Color.CYAN,
```

```
Color.DARK_GRAY, Color.GRAY, Color.GREEN, Color.LIGHT_GRAY,
Color.MAGENTA, Color.ORANGE, Color.PINK, Color.RED, Color.WHITE,
Color.YELLOW };
```

```
// Dimensoes iniciais do JFrame.
```

```
private int WIDTH = 440;
```

```
private int HEIGHT = 400;
```

```
public J12E3()
```

```
{
    setTitle("Jogo do Caos III");
    setSize(WIDTH,HEIGHT);
    setLocation(250,250);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents(); // Metodo que adiciona componentes.
    setVisible(true);
}
```

```
// Método que a criação do jpanel: desenho resultante.
```

```
public void createContents()
{
```

```
// Criando uma especializacao de JPanel.
```

```
p1 = new JFractalPanel(0);
```

```
// Criando um menu pop-up de controle dos desenhos.
```

```
createMenu();
```

```
// Organizando p1 e p2 dentro de p3.
```

```
p3 = createJPanel3();
```

```
// Adiciona o JPanel p3 na janela.
```

```
add(p3);
```

```
}
```

```
// Método que constrói o Menu pop-up de controle.
```

```
public void createMenu()
```

```
{
    JFrame frame = new JFrame();
```

```
// Barra que permite a construcao de JMenus.
```

```
barra = new JMenuBar();
```

```
// JMenu de cores.
```

```
popup = new JMenu("Cores");
```

```
// E suas opcoes.
```

```
corGrupo = new ButtonGroup();
```

```
item = new JRadioButtonMenuItem[nomes.length];
```

```

for (int i = 0; i < item.length; i++)
{
    item[i] = new JRadioButtonMenuItem(nomes[i]);
}
item[0].setSelected(true);

for (int i = 0; i < item.length; i++)
{
    popup.add(item[i]);
    corGrupo.add(item[i]);
    item[i].addActionListener(this);
}
// Adiciona a JMenu de cores.
barra.add(popup);
// Configura a barra de menus para o JFrame.
setJMenuBar(barra);
}

// Método que constrói o JPanel que engloba o
// controle e o desenho de Fractais.
public JPanel createJPanel3()
{
    // Criando o JPanel p3 com BorderLayout.
    p3 = new JPanel();
    p3.setLayout(new BorderLayout());

    // Adicionando os demais JPanels a p3.
    p3.add(p1, BorderLayout.CENTER);

    return p3;
}

//-----//
// Metodos para tratar o comportamento do menu. //
//-----//
public void actionPerformed(ActionEvent e)
{
    //-----//
    // Foi selecionado algum item do menu pop-up. //
    //-----//
    // Obtendo os dados da interface para criar o desenho.
    for (int i = 0; i < item.length; i++)
        if (e.getSource() == item[i])
        {
            // Fornecendo a cor escolhida.
            Color cor = colors[i];

```

```

// Redesenhando usando parametros atualizados.
p1.setCor(cor);
p1.setNpontos(5000);
// Redesenhando toda a interface.
repaint();
// Parar o laço.
break;
}

}

//-----//

public static void main(String args[])
{
    J12E3 janela1 = new J12E3();
}

}

```

Figura 3.2: Código da classe J12E3 que define o painel de controle do Jogo do Caos que emprega **JMenu**.

Exercício 4: Criar uma interface gráfica que permita definir, para o Jogo do Caos, empregando um Menu de opções de cores tal como dado na Figura 4.1 que emprega **JCheckBox** ao invés de **JRadioButton** (usado no **Exercício 3**).

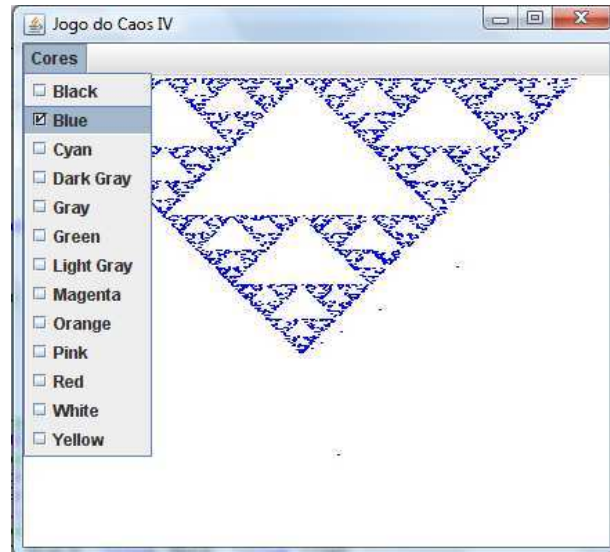


Figura 4.1: Nova interface gráfica para o Jogo do Caos que usa JMenu.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class J12E4 extends JFrame implements ActionListener
{
    private JFractalPanel p1;
    private JPanel p2, p3;
    // Definindo elementos para criar menu.
    private JMenuBar barra;
    private JMenu popup;
    private ButtonGroup corGrupo;
    private JCheckBoxMenuItem item[];

    // Definicao do numero de pontos a serem usados.
    private String pontos[] = {"1000", "5000", "10000", "50000"};
    // Definicao dos nomes a serem empregados no comboBox de cores.
    private final String nomes[] = { "Black", "Blue", "Cyan",
```

```
"Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
"Orange", "Pink", "Red", "White", "Yellow" };
// Definicao das cores a serem empregadas no desenho das formas.
private final Color colors[] = { Color.BLACK, Color.BLUE, Color.CYAN,
Color.DARK_GRAY, Color.GRAY, Color.GREEN, Color.LIGHT_GRAY,
Color.MAGENTA, Color.ORANGE, Color.PINK, Color.RED, Color.WHITE,
Color.YELLOW };

// Dimensoes iniciais do JFrame.
private int WIDTH = 440;
private int HEIGHT = 400;

public J12E4()
{
    setTitle("Jogo do Caos III");
    setSize(WIDTH, HEIGHT);
    setLocation(250, 250);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents(); // Metodo que adiciona componentes.
    setVisible(true);
}

// Método que a criação do jpanel: desenho resultante.
public void createContents()
{
    // Criando uma especializacao de JPanel.
    p1 = new JFractalPanel(0);

    // Criando um menu pop-up de controle dos desenhos.
    createMenu();

    // Organizando p1 e p2 dentro de p3.
    p3 = createJPanel3();

    // Adiciona o JPanel p3 na janela.
    add(p3);
}

// Método que constrói o Menu pop-up de controle.
public void createMenu()
{
    JFrame frame = new JFrame();

    // Barra que permite a construcao de JMenus.
    barra = new JMenuBar();
    // JMenu de cores.
    popup = new JMenu("Cores");
```

```

// E suas opcoes.
corGrupo = new ButtonGroup();
item = new JCheckBoxMenuItem[nomes.length];

for (int i = 0; i < item.length; i++)
{
    item[i] = new JCheckBoxMenuItem(nomes[i]);
}
item[0].setSelected(true);

for (int i = 0; i < item.length; i++)
{
    popup.add(item[i]);
    corGrupo.add(item[i]);
    item[i].addActionListener(this);
}
// Adiciona a JMenu de cores.
barra.add(popup);
// Configura a barra de menus para o JFrame.
setJMenuBar(barra);
}

// Método que constrói o JPanel que engloba o
// controle e o desenho de Fractais.
public JPanel createJPanel3()
{
    // Criando o JPanel p3 com BorderLayout.
    p3 = new JPanel();
    p3.setLayout(new BorderLayout());

    // Adicionando os demais JPanels a p3.
    p3.add(p1, BorderLayout.CENTER);

    return p3;
}

//-----//
// Metodos para tratar o comportamento do menu. //
//-----//
public void actionPerformed(ActionEvent e)
{
    //-----//
    // Foi selecionado algum item do menu pop-up. //
    //-----//
    // Obtendo os dados da interface para criar o desenho.
    for (int i = 0; i < item.length; i++)
        if (e.getSource() == item[i])
        {

```

```

// Fornecendo a cor escolhida.
Color cor = colors[i];

// Redesenhando usando parametros atualizados.
p1.setCor(cor);
p1.setNpontos(5000);
// Redesenhando toda a interface.
repaint();
// Parar o laço.
break;
}

}

//-----//

public static void main(String args[])
{
    J12E4 janela1 = new J12E4();
}

}

```

Figura 4.2: Código da classe J12E4 que define o painel de controle do Jogo do Caos que emprega **JMenu**.

Exercício 5: Criar uma interface gráfica que permita definir, para o Jogo do Caos, empregando um Menu de opções de cores flutuante (**JPopupMenu**) que é ativado com o clique do botão direito do mouse.

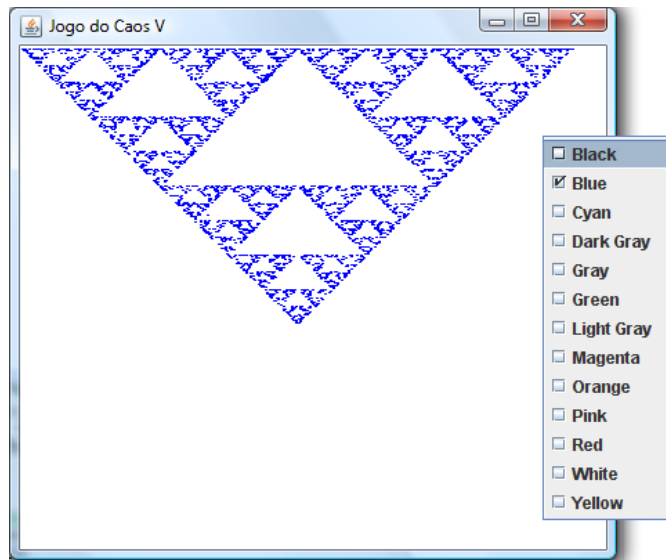


Figura 5.1: Nova interface gráfica para o Jogo do Caos que usa JPopupMenu.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class J12E5 extends JFrame implements ActionListener,
MouseListener
{
    private JFractalPanel p1;
    private JPanel p2, p3;
    // Definindo elementos para criar menu flutuante.
    private JPopupMenu popup;
    private ButtonGroup corGrupo;
    private JCheckBoxMenuItem item[];

    // Definicao dos nomes a serem empregados no comboBox de cores.
    private final String nomes[] = { "Black", "Blue", "Cyan",
    "Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
    "Orange", "Pink", "Red", "White", "Yellow" };
    // Definicao das cores a serem empregadas no desenho das formas.
    private final Color colors[] = { Color.BLACK, Color.BLUE, Color.CYAN,
```

```
Color.DARK_GRAY, Color.GRAY, Color.GREEN, Color.LIGHT_GRAY,
Color.MAGENTA, Color.ORANGE, Color.PINK, Color.RED, Color.WHITE,
Color.YELLOW };
```

// Dimensoes iniciais do JFrame.

```
private int WIDTH = 440;
```

```
private int HEIGHT = 400;
```

```
public J12E5()
```

```
{
    setTitle("Jogo do Caos V");
    setSize(WIDTH,HEIGHT);
    setLocation(250,250);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents(); // Metodo que adiciona componentes.
    setVisible(true);
}
```

// Método que a criação do jpanel: desenho resultante.

```
public void createContents()
```

```
{
    // Criando uma especializacao de JPanel.
    p1 = new JFractalPanel(0);
```

```
// Criando um menu pop-up de controle dos desenhos.
createMenu();
```

// Organizando p1 e p2 dentro de p3.

```
p3 = createJPanel3();
```

// Adiciona o JPanel p3 na janela.

```
add(p3);
```

```
}
```

// Método que constrói o Menu pop-up de controle.

```
public void createMenu()
```

```
{
```

```
    popup = new JPopupMenu();
    corGrupo = new ButtonGroup();
    item = new JCheckBoxMenuItem[nomes.length];
```

```
for (int i = 0; i < item.length; i++)
```

```
{
    item[i] = new JCheckBoxMenuItem(nomes[i]);
}
```

```
item[0].setSelected(true);
```

```

for (int i = 0; i < item.length; i++)
{
    popup.add(item[i]);
    corGrupo.add(item[i]);
    item[i].addActionListener(this);
}
p1.addMouseListener(this);
}

// Método que constrói o JPanel que engloba o
// controle e o desenho de Fractais.
public JPanel createJPanel3()
{ // Criando o JPanel p3 com BorderLayout.
    p3 = new JPanel();
    p3.setLayout(new BorderLayout());

    // Adicionando os demais JPanels a p3.
    p3.add(p1, BorderLayout.CENTER);

    return p3;
}

//-----//
// Metodos para tratar o comportamento do menu. //
//-----//
public void actionPerformed(ActionEvent e)
{
    //-----//
    // Foi selecionado algum item do menu pop-up. //
    //-----//
    // Obtendo os dados da interface para criar o desenho.
    for (int i = 0; i < item.length; i++)
        if (e.getSource() == item[i])
        {
            // Fornecendo a cor escolhida.
            Color cor = colors[i];

            // Redesenhando usando parametros atualizados.
            p1.setCor(cor);
            p1.setNpontos(5000);
            // Redesenhando toda a interface.
            repaint();
            // Parar o laço.
            break;
        }
}
}

```

```

//-----//
// Metodos para tratar o comportamento do mouse. //
//-----//
// Métodos da interface MouseListener
public void mousePressed(MouseEvent e)
{
    verificaGatilhoPopup(e);
}

public void mouseReleased(MouseEvent e)
{
    verificaGatilhoPopup(e);
}

public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mouseClicked(MouseEvent event) {}

//-----//

// Funcao auxiliar que faz aparecer o menu pop-up
// na posicao onde o mouse foi apertado ou solto.
private void verificaGatilhoPopup(MouseEvent e)
{
    // Clicou com o botão direito do mouse !
    if (e.isMetaDown())
    {
        popup.show(e.getComponent(), e.getX(), e.getY());
    }
}

public static void main(String args[])
{
    J12E5 janela1 = new J12E5();
}
}

```

Figura 5.2: Código da classe J12E5 que define o painel de controle do Jogo do Caos que emprega **JPopupMenu**.

Exercício 6: Criar uma interface gráfica que permita definir, para o Jogo do Caos, empregando um Menu de opções de número de iterações (**JPopupMenu**) que é ativado com o clique do botão direito do mouse tal como dado na Figura 6.1.

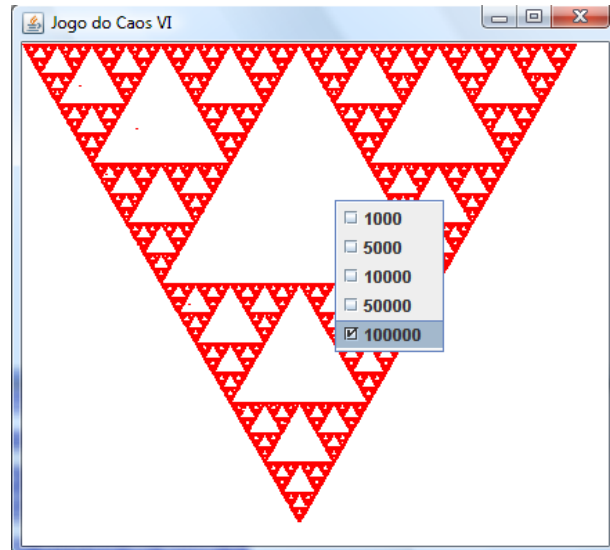


Figura 6.1: Nova interface gráfica para o Jogo do Caos que usa JPopupMenu.

Além disso, uma nova classe **JFractalPanel2** deverá redefinir o método **drawFractal** de modo a desenhar a figura do Jogo do Caos a partir das equações dadas por (1). Nesta nova classe, serão sorteados, ao invés de vértices do triângulo, 3 opções de coeficientes para as equações dadas em (1). Por exemplo, se for sorteado o valor 1, então, a atualização das coordenadas (x, y) empregará $a_{11} = 0.5$, $a_{12} = 0.0$, $a_{21} = 0.0$, $a_{22} = 0.5$, $b_1 = 0.0$ e $b_2 = 0.0$. Observe que a última coluna da Tabela 6.1 indica a probabilidade dos coeficientes serem escolhidos.

$$\begin{cases} x^{(k+1)} = a_{11}x^{(k)} + a_{12}y^{(k)} + b_1 \\ y^{(k+1)} = a_{21}x^{(k)} + a_{22}y^{(k)} + b_2 \end{cases} \quad (1)$$

Sorteio	A11	a12	a21	a22	b1	b2	p
1	0.5	0.0	0.0	0.5	0.000	0.000	0.33
2	0.5	0.0	0.0	0.5	0.500	0.000	0.33
3	0.5	0.0	0.0	0.5	0.250	0.433	0.34

Tabela 6.1: Coeficientes a serem utilizados para cada valor sorteado.

As Figuras 6.2 e 6.3 mostram o código da classe **J12E6**, com o novo menu de opções, e o novo código da classe **JFractalPanel2**, que redefine **drawFractal**, respectivamente.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class J12E6 extends JFrame implements ActionListener,
MouseListener
{
    private JFractalPanel2 p1;
    private JPanel p2, p3;
    // Definindo elementos para criar menu flutuante.
    private JPopupMenu popup;
    private ButtonGroup pontoGrupo;
    private JCheckBoxMenuItem item[];

    // Definicao dos nomes a serem empregados no comboBox de pontos.
    private final String nomes[] = { "1000", "5000", "10000",
    "50000", "100000" };

    // Definicao de numero de pontos a serem empregados no desenho
    // das formas.
    private final int pontos[] = { 1000, 5000, 10000, 50000, 100000 };

    // Dimensoes iniciais do JFrame.
    private int WIDTH = 440;
    private int HEIGHT = 400;

    public J12E6()
    {
        setTitle("Jogo do Caos VI");
        setSize(WIDTH, HEIGHT);
        setLocation(250, 250);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        createContents(); // Metodo que adiciona componentes.
        setVisible(true);
    }

    // Método que a criação do jpanel: desenho resultante.
    public void createContents()
    {
        // Criando uma especializacao de JPanel.
        p1 = new JFractalPanel2(0);

        // Criando um menu pop-up de controle dos desenhos.
        createMenu();
    }
}
```

```

// Organizando p1 e p2 dentro de p3.
p3 = createJPanel3();

// Adiciona o JPanel p3 na janela.
add(p3);
}

// Método que constrói o Menu pop-up de controle.
public void createMenu()
{
    popup = new JPopupMenu();
    pontoGrupo = new ButtonGroup();
    item = new JCheckBoxMenuItem[nomes.length];

    for (int i = 0; i < item.length; i++)
    {
        item[i] = new JCheckBoxMenuItem(nomes[i]);
    }
    item[0].setSelected(true);

    for (int i = 0; i < item.length; i++)
    {
        popup.add(item[i]);
        pontoGrupo.add(item[i]);
        item[i].addActionListener(this);
    }
    p1.addMouseListener(this);
}

// Método que constrói o JPanel que engloba o
// controle e o desenho de Fractais.
public JPanel createJPanel3()
{
    // Criando o JPanel p3 com BorderLayout.
    p3 = new JPanel();
    p3.setLayout(new BorderLayout());

    // Adicionando os demais JPanels a p3.
    p3.add(p1, BorderLayout.CENTER);

    return p3;
}

//-----//
// Metodos para tratar o comportamento do menu. //

```

```

//-----//
public void actionPerformed(ActionEvent e)
{
    // Foi selecionado algum item do menu pop-up.
    // Obtendo os dados da interface para criar o desenho.
    for (int i = 0; i < item.length; i++)
        if (e.getSource() == item[i])
        {
            // Fornecendo a cor escolhida.
            int np = pontos[i];

            // Redesenhando usando parametros atualizados.
            p1.setCor(Color.RED);
            p1.setNpontos(np);
            // Redesenhando toda a interface.
            repaint();
            // Parar o laço.
            break;
        }
}

//-----//
// Metodos para tratar o comportamento do mouse. //
//-----//
// Métodos da interface MouseListener
public void mousePressed(MouseEvent e)
{
    verificaGatilhoPopup(e);
}

public void mouseReleased(MouseEvent e)
{
    verificaGatilhoPopup(e);
}

public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mouseClicked(MouseEvent event) {}

//-----//

// Funcao auxiliar que faz aparecer o menu pop-up
// na posicao onde o mouse foi apertado ou solto.
private void verificaGatilhoPopup(MouseEvent e)
{
    // Clicou com o botão direito do mouse !
    if (e.isMetaDown())
    {
        popup.show(e.getComponent(), e.getX(), e.getY());
    }
}
}

```



```

public static void main(String args[])
{
    J12E6 janela1 = new J12E6();
}

```

Figura 6.2: Código da classe J12E6 que define o painel de controle do Jogo do Caos que emprega JPopupMenu.

```

// JFractalPanel2 demonstra desenho do Jogo do Caos
// realizado de forma diferente da realizada de JFractalPanel.
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Dimension;
import javax.swing.JPanel;
import java.util.Random;

public class JFractalPanel2 extends JFractalPanel
{
    // Construtor sem parametros usa o construtor com parametros.
    public JFractalPanel2( )
    {
        // Chamada ao construtor com parametros.
        this(0);
    } // fim do construtor JFractalPanel2.

    // Construtor e o mesmo de JFractalPanel (superclasse).
    public JFractalPanel2(int level)
    {
        // Chamada ao construtor com parametros.
        super(level);
    } // fim do construtor JFractalPanel2.

    public void drawFractal(Graphics g)
    {
        // Preenche a area de desenho com um retangulo branco
        // e portanto, "limpa" a area de desenho !
        g.setColor(Color.WHITE);
        g.fillRect(0, 0, getWidth(), getHeight());

        // Definindo a cor a ser usada no desenho.
        g.setColor(Color.RED);

        // Criando variavel auxiliar para gerar numeros aleatorios.
        Random rand = new Random();

```

```

// Gerando aleatoriamente um ponto inicial.
int x=rand.nextInt(getWIDTH()+1);
int y=rand.nextInt(getHEIGHT()+1);
int vertice=0;
int np = getNpontos();

// Laco para desenhar os pontos input vezes
for (int i=0;i<np;i++)
{
    // Realizando sorteio dos 3 vertices.
    vertice=1+rand.nextInt(3);

    // Verificando qual vertice foi sorteado.
    switch (vertice)
    {
        // Vertice (0,0).
        case 1:
            x = (0 + x) /2;
            y = (0 + y) /2;
            break;

        // Vertice (WIDTH/2, HEIGHT/2).
        case 2:
            x = x/2 + getWidth()/2;
            y = y/2;
            break;

        // Vertice (WIDTH, 0).
        case 3:
            x = x/2 + (int)(0.25*getWidth());
            y = y/2 + (int)(0.433*getHeight());
            break;
    }
    // Desenho de uma elipse.
    g.fillOval(x, y, 2, 2);
}
} // fim da classe JFractalPanel2.

```

Figura 6.3: Código da classe JFractalPanel2 que redefine drawFractal de JFractalPanel.

Exercício 7: Baseado no Exercícios anteriores construir uma das interfaces gráficas dadas a seguir. Para tanto, o seguinte algoritmo deverá ser seguido:

Passo 1: Some todos os números da sua matrícula e armazene em uma variável x. Se houver letras, use-as empregando a seguinte correspondência: a -> 1, b-> 2, c->3, e assim por diante.

Passo 2: Calcule o valor da variável m de acordo com a seguinte fórmula:
 $m = (x \% 2) + 1$.

Passo 3: Calcule o valor da variável n de acordo com a seguinte fórmula:
 $n = (x \% 4) + 1$.

Passo 4: Calcule o valor da variável p de acordo com a seguinte fórmula:
 $t = (x \% 11) + 1$.

Passo 5: Empregando as tabelas 7.1, 7.2 e 7.3, determine se a interface terá usado JMenu ou JPopupMenu, escolha de cores ou número de pontos e qual conjunto de coeficientes relativos a construção de uma figura.

Exemplo 1: Seja a seguinte matrícula: 08001

Passo 1: 08001 -> $x = 0 + 8 + 0 + 0 + 1 = 9$

Passo 2: $m = (9 \% 2) + 1 = 1 + 1 = 2$

Passo 3: $n = (9 \% 4) + 1 = 1 + 1 = 2$

Passo 4: $t = (9 \% 11) + 1 = 9 + 1 = 10$

Passo 5: Fazer a interface gráfica em acordo com as Tabela 7.1, 7.2 e 7.3, ou seja, a interface gráfica que usa JPopupMenu, escolha do número de iterações e a da figura a ser construída, respectivamente. Neste caso, **JMenu** com **número de pontos** e a figura **Pine Tree**.

m	Menu
1	Jmenu
2	JpopupMenu

Tabela 7.1: Correspondência entre m e o tipo de Menu.

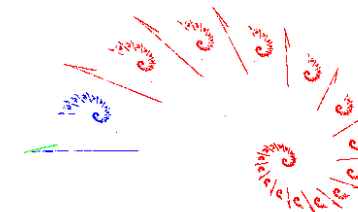
n	Escolha
1	Cores
2	Número de pontos
3	Número de pontos
4	Cores

Tabela 7.2: Correspondência entre n e o tipo de escolha no menu.

t	Figura
1	Nautilus
2	Spiralfun
3	Spiral
4	Crab
5	Eiffel Tower
6	Kevin's Fern
7	Kevin's Spiral
8	Leaf
9	Tree
10	Pine Tree
11	Crab Tree

Tabela 7.3: Correspondência entre t e o tipo de figura.

1. Nautilus



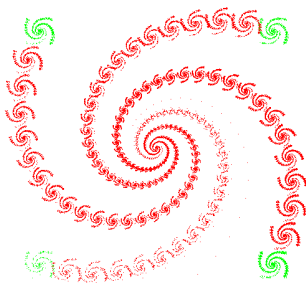
A11	A12	A21	A22	B1	B2	P
0.860671	0.401487	-0.402177	0.860992	0.108537	0.075138	0.930000
0.094957	-0.000995	0.237023	0.002036	-0.746911	0.047343	0.020000
0.150288	0.000000	0.000000	0.146854	-0.563199	0.032007	0.030000
0.324279	-0.002163	0.005846	0.001348	-0.557936	-0.139735	0.020000

2. Spiralfun



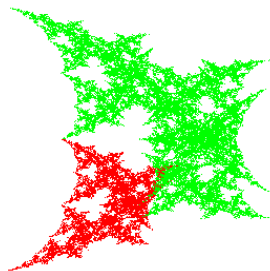
A11	A12	A21	A22	B1	B2	P
0.250000	0.000000	0.000000	0.250000	0.000000	0.500000	0.073459
0.822978	-0.475000	0.474955	0.822724	0.301140	-0.173839	0.926541

3. Spiral



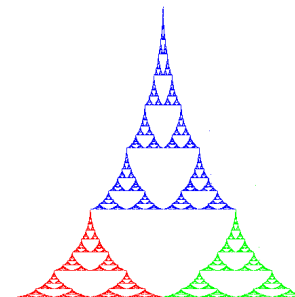
A11	A12	A21	A22	B1	B2	P
0.935567	-0.164966	0.164966	0.935567	0.000000	0.000000	0.919682
0.100000	0.000000	0.000000	0.100000	0.750000	-0.750000	0.020080
0.100000	0.000000	0.000000	0.100000	-0.750000	0.750000	0.020080
0.100000	0.000000	0.000000	0.100000	0.750000	0.750000	0.020080
0.100000	0.000000	0.000000	0.100000	-0.750000	-0.750000	0.020080

4. Crab



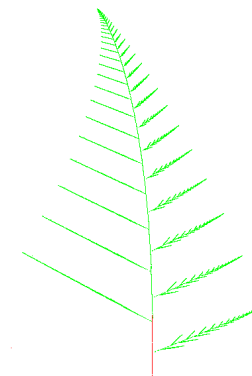
A11	A12	A21	A22	B1	B2	P
0.500000	0.200000	0.000000	0.500000	0.000000	0.000000	0.250000
-0.500000	0.000000	0.200000	0.500000	1.000000	0.000000	0.250000
0.500000	0.200000	0.000000	-0.500000	0.000000	1.000000	0.250000
-0.500000	0.000000	-0.200000	-0.500000	1.000000	1.000000	0.250000

5. Eiffel Tower



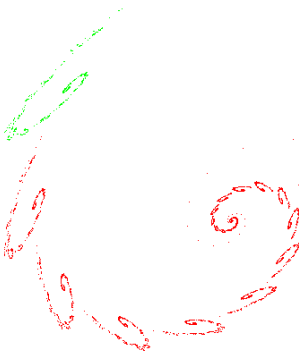
A11	A12	A21	A22	B1	B2	P
0.500000	0.000000	0.000000	0.300000	0.000000	0.000000	0.246154
0.500000	0.000000	0.000000	0.300000	0.500000	0.000000	0.246154
0.500000	0.000000	0.000000	0.700000	0.250000	0.300000	0.507692

6. Kevin's Fern



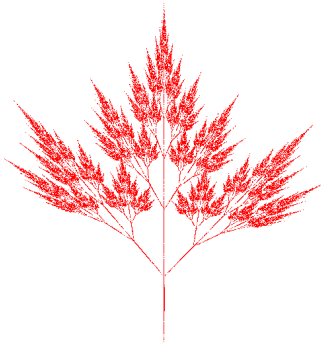
A11	A12	A21	A22	B1	B2	P
0.000000	0.000000	0.000000	0.172033	0.496139	-0.090510	0.010000
0.075906	0.312285	-0.257105	0.204233	0.494173	0.132616	0.075000
0.821130	-0.028405	0.029799	0.845280	0.087877	0.175709	0.840000
-0.023936	-0.356062	-0.323405	0.074403	0.470356	0.259738	0.075000

7. Kevin's Spiral



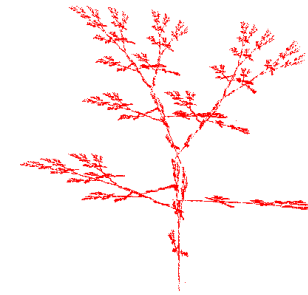
A11	A12	A21	A22	B1	B2	P
0.143268	0.372630	0.000000	0.411999	-0.424035	0.228385	0.050000
0.817528	-0.472000	0.472000	0.817528	-0.060000	0.051538	0.950000

8. Leaf



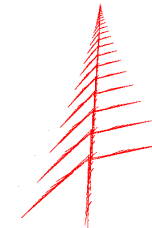
A11	A12	A21	A22	B1	B2	P
0.500000	0.000000	0.000000	0.759127	0.000000	0.542236	0.423635
0.001000	0.000000	0.000000	0.272000	0.000000	0.000000	0.050000
0.437567	0.397819	-0.086824	0.492404	-0.001575	0.428276	0.263000
-0.437567	-0.397819	-0.086824	0.492404	-0.003556	0.434301	0.263000

9. Tree



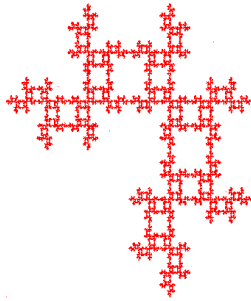
A11	A12	A21	A22	B1	B2	P
0.458805	-0.225654	0.072682	0.601870	-0.001772	0.319478	0.310000
0.342683	0.375610	-0.203368	0.546420	-0.022095	0.329764	0.200000
0.135817	0.502731	-0.313469	0.138456	-0.020039	0.217063	0.160000
0.253255	-0.489749	0.307978	0.349892	-0.007397	0.198101	0.230000
0.066406	0.000000	0.000000	0.479424	-0.014923	-0.024215	0.100000

10. Pine Tree



A11	A12	A21	A22	B1	B2	P
0.797881	0.000000	0.000000	0.833333	0.097828	0.168721	0.750000
-0.131628	-0.366787	0.399144	-0.140643	0.500098	0.180655	0.100000
0.099649	0.403824	0.408196	-0.100330	0.364628	0.157770	0.100000
0.049180	0.000000	0.000000	0.333333	0.398795	0.009695	0.050000
0.797881	0.000000	0.000000	0.833333	0.097828	0.168721	0.750000

11. Crab Tree



A11	A12	A21	A22	B1	B2	P
-0.500000	0.000000	0.000000	-0.500000	1.000000	1.000000	0.330000
0.500000	0.000000	0.000000	-0.500000	0.000000	1.000000	0.330000
-0.500000	0.000000	0.000000	0.500000	1.000000	0.000000	0.340000

Exercício 1: Criar uma interface gráfica que emprega JMenus para modificar a cor de fundo de um painel tal como dado na Figura 1.1. Os comandos para a construção dos JMenus em acordo com a hierarquia fornecida na Figura 1.1, são descritos esquematicamente na Figura 1.2. Para maiores detalhes de como construir a interface gráfico é fornecido o código Java na Figura 1.3.



Figura 1.1: Interface gráfica que permite definir a cor de fundo através de JMenus.

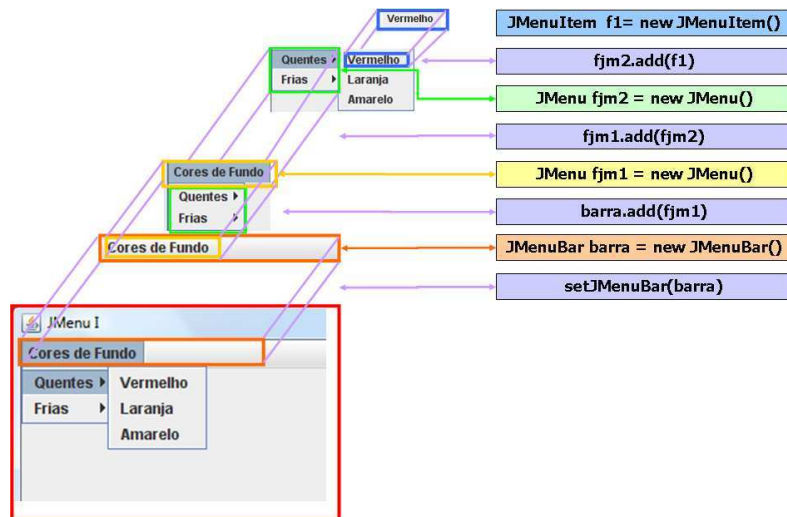


Figura 1.2: Desenho esquemático que resume como realizar a construção de JMenus de acordo a hierarquia fornecida na interface da Figura 1.1.

Classe E13J1

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class J12E1 extends JFrame implements ActionListener
{
    // Local de desenho.
    private JPanel p1;
    // Elementos para construção de JMenu.
    private JMenuBar barra;
    private JMenu fjm1, fjm2, fjm3;
    private JMenuItem fcor1, fcor2, fcor3, fcor4, fcor5, fcor6;

    // Dimensoes iniciais do JFrame.
    private int WIDTH = 720;
    private int HEIGHT = 460;

    public J13E1()
    {
        setTitle("JMenu I");
        setSize(WIDTH, HEIGHT);
        setLocation(250, 250);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        createContents(); // Metodo que adiciona componentes.
        setVisible(true);
    }

    // Criação do JPanel e dos JMenus.
    public void createContents()
    {
        // Criando JPanel de desenho e menus.
        p1 = createJPanel1();
        createJMenu1();
        // Adicionando p1 na janela.
        add(p1);
    }

    // Método que constrói o JPanel de controle.
    public JPanel createJPanel1()
    {
        // Criando o JPanel p1.
        p1 = new JPanel();

        // Retorna o JPanel p1 criado vazio.
        return p1;
    }
}
```

```

// Criando o JMenu de cores.
public void createJMenu1()
{
    // Criando objeto JMenuBar.
    barra = new JMenuBar();

    // Criando objeto JMenu principal.
    fjm1 = new JMenu("Cores de Fundo");

    //-----//
    // Criando objetos dos sub JMenu.      //
    //-----//
    fjm2 = new JMenu("Quentes");

    // Criando elementos do sub JMenu de cores quentes.
    fcor1 = new JMenuItem("Vermelho");
    fcor2 = new JMenuItem("Laranja");
    fcor3 = new JMenuItem("Amarelo");

    // Adicionando tratamento de evento aos itens de menu.
    fcor1.addActionListener(this);
    fcor2.addActionListener(this);
    fcor3.addActionListener(this);

    // Adicionando itens ao sub JMenu.
    fjm2.add(fcor1);
    fjm2.add(fcor2);
    fjm2.add(fcor3);

    //-----//
    //-----//
    // Criando objetos dos sub JMenu.      //
    //-----//
    fjm3 = new JMenu("Frias");

    // Criando elementos do sub JMenu de cores quentes.
    fcor4 = new JMenuItem("Verde");
    fcor5 = new JMenuItem("Azul");
    fcor6 = new JMenuItem("Branco");

    // Adicionando tratamento de evento aos itens de menu.
    fcor4.addActionListener(this);
    fcor5.addActionListener(this);
    fcor6.addActionListener(this);

```

```

// Adicionando itens ao sub JMenu.
fjm3.add(fcor4);
fjm3.add(fcor5);
fjm3.add(fcor6);

//-----//

// Adicionando os sub JMenus ao JMenu de Cores.
fjm1.add(fjm2);
fjm1.add(fjm3);

// Adicionando o JMenu de cores na barra.
barra.add(fjm1);

// Modificando a barra de menus para a Janela.
setJMenuBar(barra);

}

public void actionPerformed(ActionEvent event)
{
    // Verificando as fontes de evento, que sao
    // os itens de menu de cor de fundo, e para
    // cada uma definir a cor de fundo adequada.
    if (event.getSource() == fcor1)
        p1.setBackground(Color.RED);
    else if (event.getSource() == fcor2)
        p1.setBackground(Color.ORANGE);
    else if (event.getSource() == fcor3)
        p1.setBackground(Color.YELLOW);
    else if (event.getSource() == fcor4)
        p1.setBackground(Color.GREEN);
    else if (event.getSource() == fcor5)
        p1.setBackground(Color.BLUE);
    else if (event.getSource() == fcor6)
        p1.setBackground(Color.WHITE);
}

public static void main(String args[])
{
    J13E1 janela1 = new J13E1();
}
}

```

Figura 1.3 : Código Java para a interface gráfica da Figura 1.1.

Exercício 2: Modificar a interface gráfica do Exercício 1 de modo que todos os itens de menu sejam declarados em um vetor. Para tanto será necessário realizar modificações nos seguintes trechos do código:

Modificação 1: No campo da classe

Trocar

```
private JMenuItem fcor1, fcor2, fcor3, fcor4, fcor5, fcor6;
```

Por

```
private JMenuItem fcor[] = new JMenuItem[6];
private String nomes[] = {"Vermelho", "Laranja", "Amarelo", "Verde", "Azul", "Branco"};
private Color cores[] = {Color.RED, Color.ORANGE, Color.YELLOW, Color.GREEN, Color.BLUE, Color.WHITE};
```

Modificação 2: No método createJMenu1

Trocar

```
// Criando elementos do sub JMenu de cores quentes.
fcor1 = new JMenuItem("Vermelho");
fcor2 = new JMenuItem("Laranja");
fcor3 = new JMenuItem("Amarelo");

// Criando elementos do sub JMenu de cores frias.
fcor4 = new JMenuItem("Verde");
fcor5 = new JMenuItem("Azul");
fcor6 = new JMenuItem("Branco");
```

Por

```
// Criando elementos do sub JMenu de cores quentes e frias.
for(int i=0; i < cores.length; i++)
    fcor[i] = new JMenuItem(nomes[i]);
```

Modificação 3: No método createJMenu1

Trocar

```
// Adicionando tratamento de evento aos itens de menu.
fcor1.addActionListener(this);
fcor2.addActionListener(this);
fcor3.addActionListener(this);

// Adicionando tratamento de evento aos itens de menu.
fcor4.addActionListener(this);
fcor5.addActionListener(this);
fcor6.addActionListener(this);
```

Por

```
// Criando elementos do sub JMenu de cores quentes e frias.
for(int i=0; i < cores.length; i++)
    fcor[i].addActionListener(this);
```

Modificação 4: No método createJMenu1

Trocar

```
// Adicionando itens ao sub JMenu.
fjm2.add(fcor1);
fjm2.add(fcor2);
fjm2.add(fcor3);

// Adicionando itens ao sub JMenu.
fjm2.add(fcor4);
fjm2.add(fcor5);
fjm2.add(fcor6);
```

Por

```
// Adicionando itens ao sub JMenu cores quentes.
for(int i=0; i < 3; i++)
    fjm2.add(fcor[i]);

// Adicionando itens ao sub JMenu cores frias.
for(int i=3; i < cores.length; i++)
    fjm3.add(fcor[i]);
```

Modificação 5: No método actionPerformed

Trocar

```
// Verificando as fontes de evento, que são
// os itens de menu de cor de fundo, e para
// cada uma definir a cor de fundo adequada.
if (event.getSource() == fcor1)
    p1.setBackground(Color.RED);
else if (event.getSource() == fcor2)
    p1.setBackground(Color.ORANGE);
else if (event.getSource() == fcor3)
    p1.setBackground(Color.YELLOW);
else if (event.getSource() == fcor4)
    p1.setBackground(Color.GREEN);
else if (event.getSource() == fcor5)
    p1.setBackground(Color.BLUE);
else if (event.getSource() == fcor6)
    p1.setBackground(Color.WHITE);
```


Por

```
// Laco para identificar a fonte de evento e realizar a ação correspondente.  
for(int i=0; i < cores.length; i++)  
    if (event.getSource() == fcor[i])  
        p1.setBackground(cores[i]);
```

Exercício 3: O propósito deste exercício é o de apresentar quatro interfaces gráficas que empregam JMenu para alterar as propriedades de um texto contido em um JPanel. Cada uma das interfaces é parte de uma interface mais complexa. Para integrar as quatro interfaces propõem-se os seguintes passos:

Passo 1: Separar a turma em quatro grupos distintos: G1, G2, G3 e G4.

Passo 2: Cada grupo deverá responder as perguntas do questionário abaixo do modo a entender o funcionamento da sua interface.

Passo 3: Serão formados novos grupos de modo que estes novos grupos tenham pelo menos um integrante de cada dos grupos G1, G2, G3 e G4. Estes novos grupos terão como objetivo construir a interface completa que emprega o código das interfaces G1, G2, G3 e G4.

Uma observação importante é que esta atividade incentiva e procura treinar em seus participantes a habilidade de trabalhar em equipe e que servirá para atividades a serem desenvolvidas em laboratórios posteriores.

Todos os grupos deverão responder as perguntas contidas no questionário a seguir para entender sua interface e posteriormente integrar a mesma com as demais interfaces.

Questionário

- (i) O que faz a sua interface gráfica?
- (ii) Coloque em uma tabela o tipo dos componentes empregados e seus nomes.
- (iii) Utilize uma figura similar a Figura 1.2 para explicar a hierarquia existente entre os componentes da sua interface.
- (iv) Qual o nome dos componentes para os quais é realizado o tratamento de eventos?
- (v) Quais são as ações realizadas pela interface?

Se tiver dúvidas em relação aos comandos apresentados faça uma lista e chame o professor para realizar eventuais esclarecimentos.

GRUPO 1 - Empregando Menus – O Menu Arquivo

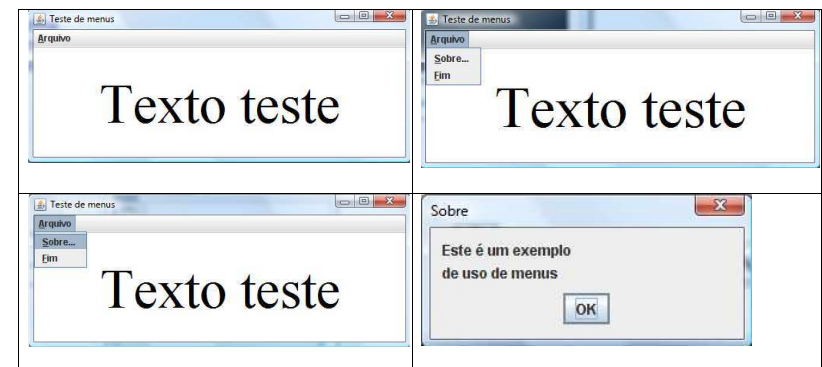


Figura 3.1: Telas da Interface para o Menu Arquivo.

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class J13G1 extends JFrame implements ActionListener  
{  
    private JPanel p1;  
    private Color cores[] = {Color.black, Color.blue, Color.red, Color.green};  
    private String cor[] = {"Preto", "Azul", "Vermelho", "Verde"};  
    private JLabel mostra;  
    private JMenu arqMenu;  
    private JMenuItem sobreItem, fimItem;  
    private JMenuBar barra;  
  
    // Dimensoes iniciais do JFrame.  
    private int WIDTH = 500;  
    private int HEIGHT = 200;  
  
    // Definindo configuracoes da janela.  
    public J13G1()  
    {  
        setTitle("JMenu G1");  
        setSize(WIDTH,HEIGHT);  
        setLocation(250,250);  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
        createContents(); // Metodo que adiciona componentes.  
        setVisible(true);  
    }  
}
```

```

public void createContents()
{
    p1 = createJPanel1();
    createJMenu1();
    add(p1);
}

public JPanel createJPanel1()
{
    p1 = new JPanel();
    return p1;
}

public void createJMenu1()
{
    JFrame frame = new JFrame("Teste de menus");
    barra = new JMenuBar(); // a barra de menus é um contêiner

    //-----//
    // Menu Arquivo
    //-----//
    arqMenu = new JMenu("Arquivo");
    sobreItem = new JMenuItem("Sobre...");
    sobreItem.addActionListener(this);
    fimItem = new JMenuItem("Fim");
    fimItem.addActionListener(this);

    // Menu Arquivo
    arqMenu.setMnemonic('A');
    sobreItem.setMnemonic('S');
    fimItem.setMnemonic('F');
    arqMenu.add(sobreItem);
    arqMenu.add(fimItem);
    //-----//

    // Anexa o arqMenu ao contêiner
    barra.add(arqMenu);
    // Configura a barra de menus para a Janela.
    setJMenuBar(barra);

    // Definições iniciais acerca do texto que ira aparecer no JPanel p1.
    mostra = new JLabel("Texto teste", SwingConstants.CENTER);
    mostra.setForeground(cores[0]);
    mostra.setFont(new Font("TimesRoman", Font.PLAIN, 72));
    p1.setBackground(Color.white);
    p1.add(mostra, BorderLayout.CENTER);
    //-----//
}

```

```

public void actionPerformed(ActionEvent e)
{
    // Aparece caixa de mensagem.
    if (e.getSource() == sobreItem)
        JOptionPane.showMessageDialog(null,
            "Este é um exemplo\nde uso de menus", "Sobre",
            JOptionPane.PLAIN_MESSAGE);
    // Termina o programa.
    else if (e.getSource() == fimItem)
        System.exit( 0 );
}

public static void main(String args[])
{
    J13G1 j1 = new J13G1();
}
}

```

GRUPO 2 - Empregando Menus – O Menu Formatar e Submenu Cores

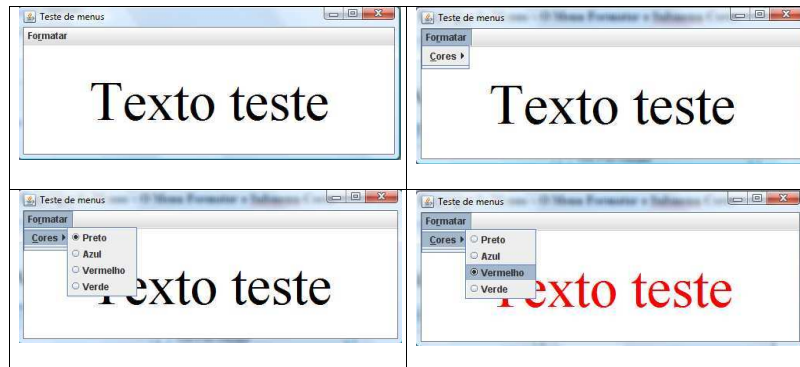


Figura 3.2: Telas da Interface para o Menu Formatar e o Submenu Cores.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class J13G2 extends JFrame implements ActionListener
{
    private JPanel p1;
    private JLabel mostra;
    private Color cores[] = {Color.black, Color.blue, Color.red, Color.green};
    private String cor[] = {"Preto", "Azul", "Vermelho", "Verde"};
    private String fonte[] = {"TimesRoman", "Courier", "Helvetica"};
    private String estilos[] = {"Negrito", "Itálico"};
    private JRadioButtonMenuItem corItem[], fontItem[];
    private JCheckBoxMenuItem estItem[];
    private JMenu arqMenu, formatMenu, corMenu, fontMenu;
    private JMenuItem sobreItem, fimItem;
    private JMenuBar barra;
    private ButtonGroup fontGrupo, corGrupo;
    private int estilo;

    // Dimensoes iniciais do JFrame.
    private int WIDTH = 500;
    private int HEIGHT = 200;

    // Definindo configuracoes da janela.
    public J13G2()
    {
        setTitle("JMenu G2");
        setSize(WIDTH,HEIGHT);
```

```
        setLocation(250,250);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        createContents(); // Metodo que adiciona componentes.
        setVisible(true);
    }
```

```
public void createContents()
{
    p1 = createJPanel1();
    createJMenu2();
    add(p1);
}
```

```
public JPanel createJPanel1()
{
    p1 = new JPanel();
    return p1;
}
```

```
public void createJMenu2()
{
```

```
    JFrame frame = new JFrame("JMenu G2");
```

```
    barra = new JMenuBar(); // a barra de menus é um contêiner
```

```
    //-----//
    // Menu Formatar
    //-----//
    formatMenu = new JMenu("Formatar");
    // Menu Formatar
    formatMenu.setMnemonic('r');
    //-----//
```

```
    // Submenu Cores
    corMenu = new JMenu("Cores");
    corItem = new JRadioButtonMenuItem[cor.length];
    corGrupo = new ButtonGroup();
    for (int i = 0; i < cor.length; i++)
    {
        corItem[i] = new JRadioButtonMenuItem(cor[i]);
    }
```

```

//-----//
// Submenu Cores
//-----//
corMenu.setMnemonic('C');
for (int i = 0; i < cor.length; i++)
{
    corMenu.add(corItem[i]);
    corGrupo.add(corItem[i]);
    corItem[i].addActionListener(this);
}
corItem[0].setSelected(true);
// Adiciona opcoes de cores ao Format Menu.
formatMenu.add(corMenu);
formatMenu.addSeparator();
// Adiciona o Format Menu ao conjunto de Menus.
barra.add(formatMenu);

//-----//
// Configura a barra de menus para o JFrame
//-----//
setJMenuBar(barra);
// Definições iniciais sobre o texto que aparece no p1.
mostra = new JLabel("Texto teste", SwingConstants.CENTER);
mostra.setForeground(cores[0]);
mostra.setFont(new Font("TimesRoman", Font.PLAIN, 72));
p1.setBackground(Color.white);
p1.add(mostra, BorderLayout.CENTER);
//-----//
}

public void actionPerformed(ActionEvent e)
{
    for (int i = 0; i < corItem.length; i++)
    {
        if (corItem[i].isSelected())
        {
            mostra.setForeground(cores[i]);
            break;
        }
    }
    repaint();
}

public static void main(String args[])
{
    J13G2 exemplo = new J13G2();
}
}

```

GRUPO 3 - Empregando Menus – O Menu Formatar e Submenu Fontes

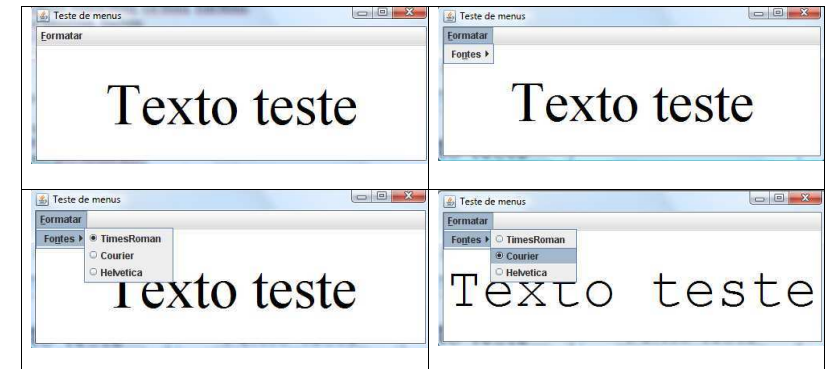


Figura 3.3: Telas da Interface para o Menu Formatar e o Submenu Fontes.

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class J13G3 extends JFrame implements ActionListener
{
    private JPanel p1;
    private JLabel mostra;
    private Color cores[] = {Color.black, Color.blue, Color.red, Color.green};
    private String cor[] = {"Preto", "Azul", "Vermelho", "Verde"};
    private String fonte[] = {"TimesRoman", "Courier", "Helvetica", "Arial"};
    private String estilos[] = {"Negrito", "Itálico"};
    private JRadioButtonMenuItem corItem[], fontItem[];
    private JCheckBoxMenuItem estItem[];
    private JMenu arqMenu, formatMenu, corMenu, fontMenu;
    private JMenuItem sobreItem, fimItem;
    private JMenuBar barra;
    private ButtonGroup fontGrupo, corGrupo;
    private int estilo;

    // Dimensoes iniciais do JFrame.
    private int WIDTH = 500;
    private int HEIGHT = 200;
}

```

```
// Definindo configuracoes da janela.
public J13G3()
{
    setTitle("JMenu G2");
    setSize(WIDTH,HEIGHT);
    setLocation(250,250);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents(); // Metodo que adiciona componentes.
    setVisible(true);
}

public void createContents()
{
    p1 = createJPanel1();
    createJMenu3();
    add(p1);
}

public JPanel createJPanel1()
{
    p1 = new JPanel();
    return p1;
}

public void createJMenu3()
{
    JFrame frame = new JFrame("Teste de menus");
    barra = new JMenuBar(); // a barra de menus é um contêiner

    //-----//
    // Menu Formatar
    //-----//
    formatMenu = new JMenu("Formatar");
    // Menu Formatar
    formatMenu.setMnemonic('F');
    //-----//

    //-----//
    // Submenu Fontes.
    //-----//
    fontItem = new JRadioButtonMenuItem[fonte.length];
    fontMenu = new JMenu("Fontes");
    // Submenu Fontes.
    fontMenu.setMnemonic('n');

    //-----//
    // Submenu Negrito ou Italico.
    //-----//
```

```
fontGrupo = new ButtonGroup();
for (int i = 0; i < fonte.length; i++)
{
    fontItem[i] = new JRadioButtonMenuItem(fonte[i]);
    fontMenu.add(fontItem[i]);
    fontGrupo.add(fontItem[i]);
    fontItem[i].addActionListener(this);
}
fontItem[0].setSelected(true);
//-----//

// Adiciona o Font Menu ao Format Menu.
// O Format Menu, por sua vez, ja foi
// colocado na barra de menus.
formatMenu.add(fontMenu);
// Adiciona o Format Menu ao conjunto de Menus.
barra.add(formatMenu);
// Configura a barra de menus para o JFrame.
setJMenuBar(barra);

// Definições iniciais sobre o texto que aparece em p1.
mostra = new JLabel("Texto teste", SwingConstants.CENTER);
mostra.setForeground(cores[0]);
mostra.setFont(new Font("TimesRoman", Font.PLAIN, 72));
p1.setBackground(Color.white);
p1.add(mostra, BorderLayout.CENTER);
//-----//
}

public void actionPerformed(ActionEvent e)
{
    for (int i = 0; i < fontItem.length; i++)
    {
        if (e.getSource() == fontItem[i])
        {
            mostra.setFont(new Font(fontItem[i].getText(), Font.PLAIN, 72));
            break;
        }
    }
    repaint();
}

public static void main(String args[])
{
    J13G3 j1 = new J13G3();
}
}
```

GRUPO 4 - Empregando Menus – O Menu Formatar e Submenu Fontes

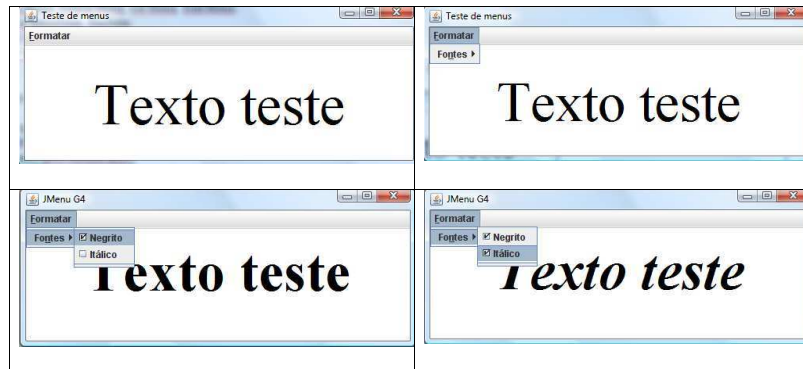


Figura 3.4: Telas da Interface para o Menu Formatar e o Submenu Fontes.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class J13G4 extends JFrame implements ActionListener
{
    private JPanel p1;
    private JLabel mostra;
    private Color cores[] = {Color.black, Color.blue, Color.red, Color.green};
    private String cor[] = {"Preto", "Azul", "Vermelho", "Verde"};
    private String fonte[] = {"TimesRoman", "Courier", "Helvetica", "Arial"};
    private String estilos[] = {"Negrito", "Itálico"};
    private JRadioButtonMenuItem corItem[], fontItem[];
    private JCheckBoxMenuItem estItem[];
    private JMenu arqMenu, formatMenu, corMenu, fontMenu;
    private JMenuItem sobreItem, fimItem;
    private JMenuBar barra;
    private ButtonGroup fontGrupo, corGrupo;
    private int estilo = 0;

    // Dimensoes iniciais do JFrame.
    private int WIDTH = 500;
    private int HEIGHT = 200;

    // Definindo configuracoes da janela.
    public J13G4()
    {
        setTitle("JMenu G4");
        setSize(WIDTH, HEIGHT);
        setLocation(250, 250);
    }
}
```

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
createContents(); // Metodo que adiciona componentes.
setVisible(true);
}

public void createContents()
{
    p1 = createJPanel1();
    createJMenu3();
    add(p1);
}

public JPanel createJPanel1()
{
    p1 = new JPanel();
    return p1;
}

public void createJMenu3()
{
    JFrame frame = new JFrame("Teste de menus");
    barra = new JMenuBar(); // a barra de menus é um contêiner

    //-----//
    // Menu Formatar
    //-----//
    formatMenu = new JMenu("Formatar");
    // Menu Formatar
    formatMenu.setMnemonic('F');
    //-----//
    // Submenu Fontes.
    //-----//
    fontItem = new JRadioButtonMenuItem[fonte.length];
    fontMenu = new JMenu("Fontes");
    // Submenu Fontes.
    fontMenu.setMnemonic('n');
    //-----//
    // Submenu Estilos.
    //-----//
    estItem = new JCheckBoxMenuItem[estilos.length];
    for (int i = 0; i < estilos.length; i++)
    {
        estItem[i] = new JCheckBoxMenuItem(estilos[i]);
        fontMenu.add(estItem[i]);
        estItem[i].addActionListener(this);
    }
    //-----//
}
```

```

// Adiciona barra de separacao entre submenus.
fontMenu.addSeparator();
// Adiciona o Font Menu ao Format Menu.
// O Format Menu sera adicionado na barra de menus.
formatMenu.add(fontMenu);
// Adiciona o Format Menu ao conjunto de Menus.
barra.add(formatMenu);
// Configura a barra de menus para o JFrame
setJMenuBar(barra);

// Definições iniciais sobre o texto que aparece em p1.
mostra = new JLabel("Texto teste", SwingConstants.CENTER);
mostra.setForeground(cores[0]);
mostra.setFont(new Font("TimesRoman", Font.PLAIN, 72));
p1.setBackground(Color.white);
p1.add(mostra, BorderLayout.CENTER);
//-----//
}

public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == estItem[0] || e.getSource() == estItem[1])
    {
        estilo = 0;
        if (estItem[0].isSelected()) estilo += Font.BOLD;
        if (estItem[1].isSelected()) estilo += Font.ITALIC;
        mostra.setFont(new Font(mostra.getFont().getName(), estilo, 72));
    }
    repaint();
}

public static void main(String args[])
{
    J13G4 j1 = new J13G4();
}
}

```

GRUPO 5 - Empregando Todos os Menus

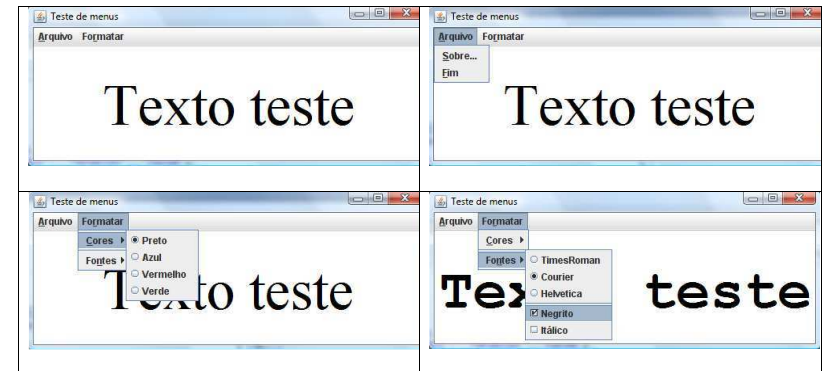


Figura 3.5: Telas da Interface para os JMenus Arquivo e Formatar.

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class J13G5 extends JFrame implements ActionListener
{
    private JPanel p1;
    private JLabel mostra;
    private Color cores[] = {Color.black, Color.blue, Color.red, Color.green};
    private String cor[] = {"Preto", "Azul", "Vermelho", "Verde"};
    private String fonte[] = {"TimesRoman", "Courier", "Helvetica", "Arial"};
    private String estilos[] = {"Negrito", "Itálico"};
    private JRadioButtonMenuItem corItem[], fontItem[];
    private JCheckBoxMenuItem estItem[];
    private JMenu arqMenu, formatMenu, corMenu, fontMenu;
    private JMenuItem sobreItem, fimItem;
    private JMenuBar barra;
    private ButtonGroup fontGrupo, corGrupo;
    private int estilo = 0;

    // Dimensoes iniciais do JFrame.
    private int WIDTH = 500;
    private int HEIGHT = 200;

    // Definindo configuracoes da janela.
    public J13G5()
    {
        setTitle("JMenu G5");
        setSize(WIDTH, HEIGHT);
    }
}

```

```

setLocation(250,250);
setDefaultCloseOperation(EXIT_ON_CLOSE);
createContents(); // Metodo que adiciona componentes.
setVisible(true);
}

public void createContents()
{
    p1 = createJPanel1();
    createJMenu3();
    add(p1);
}

public JPanel createJPanel1()
{
    p1 = new JPanel();
    return p1;
}

public void createJMenu3()
{
    JFrame frame = new JFrame("Teste de menus");
    barra = new JMenuBar(); // a barra de menus é um contêiner

    //-----//
    // Menu Formatar
    //-----//
    formatMenu = new JMenu("Formatar");
    // Menu Formatar
    formatMenu.setMnemonic('F');
    //-----//

    //-----//
    // Menu Arquivo
    //-----//
    arqMenu = new JMenu("Arquivo");
    sobreItem = new JMenuItem("Sobre...");
    sobreItem.addActionListener(this);
    fimItem = new JMenuItem("Fim");
    fimItem.addActionListener(this);

    // Menu Arquivo
    arqMenu.setMnemonic('A');
    sobreItem.setMnemonic('S');
    fimItem.setMnemonic('F');
    arqMenu.add(sobreItem);
    arqMenu.add(fimItem);

```

```

//-----//
// Submenu Cores
//-----//

// Submenu Cores
corMenu = new JMenu("Cores");
corItem = new JRadioButtonMenuItem[cor.length];
corGrupo = new ButtonGroup();
for (int i = 0; i < cor.length; i++)
{
    corItem[i] = new JRadioButtonMenuItem(cor[i]);
}

corMenu.setMnemonic('C');
for (int i = 0; i < cor.length; i++)
{
    corMenu.add(corItem[i]);
    corGrupo.add(corItem[i]);
    corItem[i].addActionListener(this);
}
corItem[0].setSelected(true);
// Adiciona opcoes de cores ao Format Menu.
formatMenu.add(corMenu);
formatMenu.addSeparator();
// Adiciona o Format Menu ao conjunto de Menus.
barra.add(formatMenu);

//-----//
// Submenu Fontes.
//-----//
fontItem = new JRadioButtonMenuItem[fonte.length];
fontMenu = new JMenu("Fontes");
// Submenu Fontes.
fontMenu.setMnemonic('n');

//-----//
// Submenu Negrito ou Italico.
//-----//
fontGrupo = new ButtonGroup();
for (int i = 0; i < fonte.length; i++)
{
    fontItem[i] = new JRadioButtonMenuItem(fonte[i]);
    fontMenu.add(fontItem[i]);
    fontGrupo.add(fontItem[i]);
    fontItem[i].addActionListener(this);
}
fontItem[0].setSelected(true);
//-----//

```



```

// Adiciona barra de separacao entre submenus.
fontMenu.addSeparator();

//-----//
// Submenu Estilos.
//-----//
estItem = new JCheckBoxMenuItem(estilos.length];
for (int i = 0; i < estilos.length; i++)
{
    estItem[i] = new JCheckBoxMenuItem(estilos[i]);
    fontMenu.add(estItem[i]);
    estItem[i].addActionListener(this);
}
//-----//

// Anexa o arqMenu ao contêiner
barra.add(arqMenu);
// Adiciona o Font Menu ao Format Menu.
// O Format Menu sera adicionado na barra de menus.
formatMenu.add(fontMenu);
// Adiciona o Format Menu ao conjunto de Menus.
barra.add(formatMenu);
// Configura a barra de menus para o JFrame
setJMenuBar(barra);

// Definições iniciais sobre o texto que aparece em p1.
mostra = new JLabel("Texto teste", SwingConstants.CENTER);
mostra.setForeground(cores[0]);
mostra.setFont(new Font("TimesRoman", Font.PLAIN, 72));
p1.setBackground(Color.white);
p1.add(mostra, BorderLayout.CENTER);
//-----//
}

public void actionPerformed(ActionEvent e)
{
    // Aparece caixa de mensagem.
    if (e.getSource() == sobreItem)
        JOptionPane.showMessageDialog(null,
            "Este é um exemplo\nde uso de menus", "Sobre",
            JOptionPane.PLAIN_MESSAGE);
    // Termina o programa.
    else if (e.getSource() == fimItem)
        System.exit( 0 );
}

```

```

// Laco para verificar a cor escolhida.
for (int i = 0; i < corItem.length; i++)
{
    if (corItem[i].isSelected())
    {
        mostra.setForeground(cores[i]);
        break;
    }
}

// Laco para verificar estilo de texto.
for (int i = 0; i < fontItem.length; i++)
{
    if (e.getSource() == fontItem[i])
    {
        mostra.setFont(new Font(fontItem[i].getText(), Font.PLAIN, 72));
        break;
    }
}

// Verifica se foi escolhido Negrito ou Italico.
if (e.getSource() == estItem[0] || e.getSource() == estItem[1])
{
    estilo = 0;
    if (estItem[0].isSelected()) estilo += Font.BOLD;
    if (estItem[1].isSelected()) estilo += Font.ITALIC;
    mostra.setFont(new Font(mostra.getFont().getName(), estilo, 72));
}
repaint();
}

public static void main(String args[])
{
    J13G5 j1 = new J13G5();
}
}

```

Exercício 1: Explorar e entender as estruturas de um programa que emprega os seguintes elementos de interface gráfica: JTabbedPane (divisão da tela em painéis) e JTable (para armazenar dados e opções selecionadas em campos anteriores) em conjunto com a leitura de dados armazenados em arquivos (através de JMenu). A descrição dos elementos e a aparência da interface são ilustradas nas Figuras 1.1 e 1.2. Deve existir um arquivo CadastroEmpresa.txt no mesmo diretório do programa.

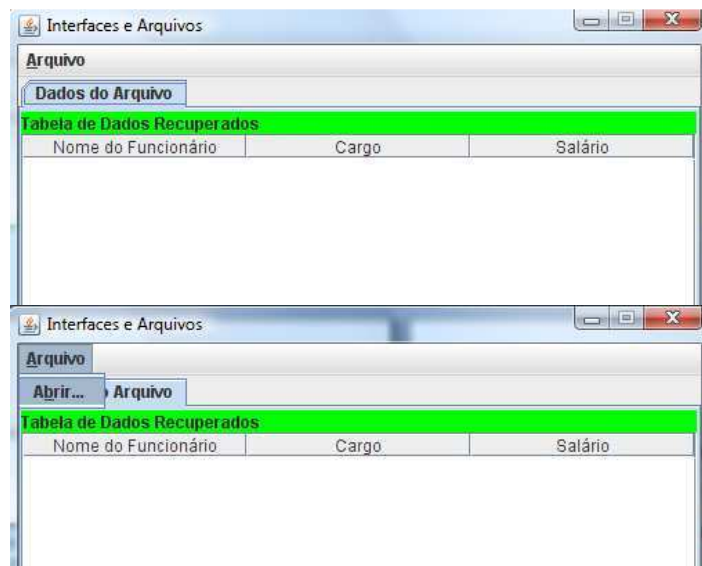


Figura 1.1: Interface do programa proposto.

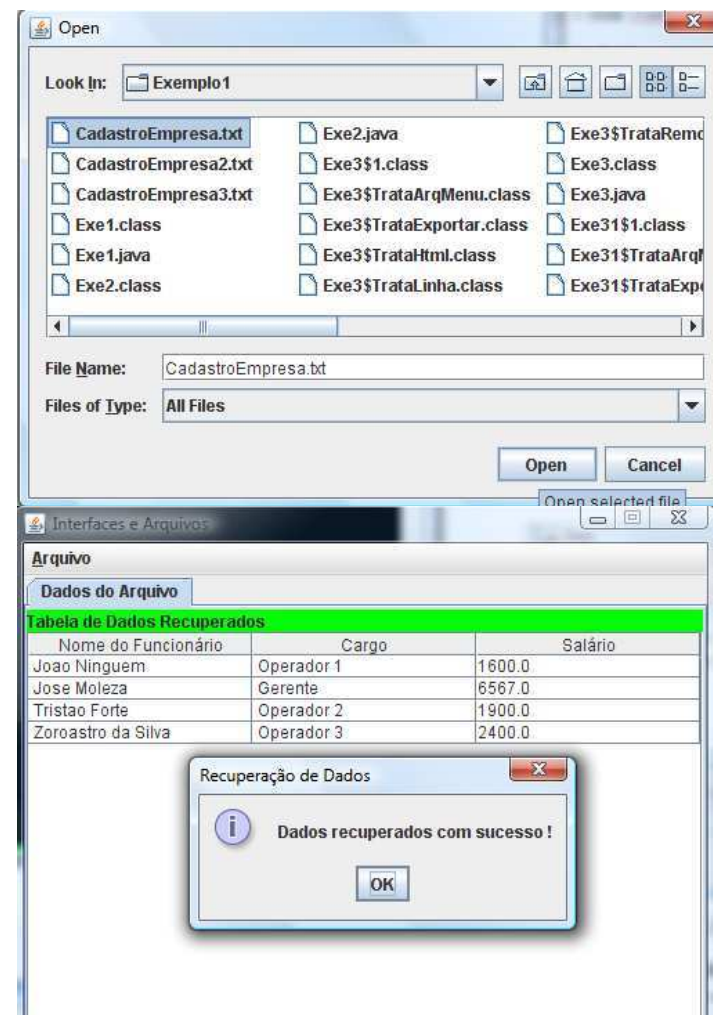


Figura 1.2: Interface do programa proposto.

Com o código da Figura 1.3 descobrir como a GUI gera as Figuras 1.1 e 1.2.

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import java.io.*;
import java.util.Scanner;

public class Exe31 extends JFrame
{
    private JMenu arqMenu;
    private JMenuItem abrirItem, salvarItem;
    private JMenuBar barra;
    private final int WIDTH = 500; // Largura da GUI.
    private final int HEIGHT = 550; // Altura da GUI.
    // Criando um JTabbedPane.
    JTabbedPane tabPane = new JTabbedPane();
    JPanel p1;
    // Criando uma JTable onde ficaram armazenadas as informacoes de relatorio.
    JTable jTableEntrada;
    // Criando um modelo de JTable a partir do qual podemos manipular os dados
    // de JTable.
    DefaultTableModel modeloEntrada;
    // Itens para tratar a abertura e leitura dos dados de um arquivo texto.
    File fileDir; // Diretorio ou arquivo especificado pelo usuário.
    int response; // resposta do usuario na GUI.
    String output = ""; // lista de arquivos com tamanhos.
    JFileChooser chooser = new JFileChooser(".");
    // Itens para abrir e ler conteudo de arquivos ou abrir e escrever dados.
    Scanner fileIn;
    PrintWriter fileOut;
    String line;

    public Exe31()
    {
        setTitle("Interfaces e Arquivos");
        setSize(WIDTH,HEIGHT);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocation(100,50);
        setResizable(false);
        createContents(); // Metodo que adiciona componentes.
        setVisible(true);
    }
}

```

```

public void createContents()
{
    //-----//
    // CRIANDO JMENUS.
    //-----//

    barra = new JMenuBar(); // a barra de menus é um contêiner

    // Cria JMenu arquivo.
    arqMenu = createJMenu1();
    // Anexa o arqMenu a Barra de Menus.
    barra.add(arqMenu);

    // Configura a barra de menus para o JFrame
    //-----//
    setJMenuBar(barra);

    //-----//

    //-----//
    // CRIANDO JPANELS.
    //-----//

    // Cria o panel com informacoes carregadas em arquivo.
    p1 = createJPanel1();

    // Adicionando o JTabbedPane a Janela.
    tabPane.addTab("Dados do Arquivo",null,p1,"Dados carregados");
    add(tabPane);

    //-----//
}

//-----//
// CRIANDO JMENUS.
//-----//

public JMenu createJMenu1()
{
    //-----//
    // Menu Arquivo
    //-----//

    arqMenu = new JMenu("Arquivo");
    arqMenu.setMnemonic('A');
}

```

```

abrirItem = new JMenuItem("Abrir...");
abrirItem.setMnemonic('b');
abrirItem.addActionListener(new TrataArqMenu());

// Adicionando itens ao JMenu arqMenu.
arqMenu.add(abrirItem);

// Retornando o arqMenu preenchido.
return arqMenu;
//-----//
}

private JPanel createJPanel1()
{
    p1 = new JPanel();
    JLabel label2 = new JLabel("Tabela de Dados Recuperados");
    p1.setBackground(Color.GREEN);

// Construindo a JTable.
String[] colunas = new String[] {"Nome do Funcionário", "Cargo", "Salário"};

Object[][] dados = new Object[][]{};

// Ao inves de passar direto, colocamos os dados em um modelo
modeloEntrada = new DefaultTableModel(dados, colunas);
// e passamos o modelo para criar a jtable.
jtableEntrada = new JTable( modeloEntrada );

// Colocando a tabela dentro de uma barra de rolagem,
// senão o cabeçalho não ira aparecer.
JScrollPane jsp = new JScrollPane(jtableEntrada);
jtableEntrada.setFillsViewportHeight(true);

// Adicionando os elementos na interface.
p1.setLayout(new BorderLayout());
p1.add(label2,BorderLayout.NORTH);
// Adicionando a barra de rolagem que contém a jtable.
p1.add(jsp,BorderLayout.CENTER);
return p1;
}

//-----//

```

```

//-----//
// TRATAMENTO DE EVENTOS.
//-----//

//-----//
// TRATAMENTO DE EVENTOS DO MENU ARQUIVO.
//-----//

private class TrataArqMenu implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {

//-----//
// Tratando eventos de abertura de arquivos e recuperacao de dados.
//-----//
        if (e.getSource() == abrirItem)
        {
            // Abrindo painel grafico para escolha de arquivos
            chooser.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
            response = chooser.showOpenDialog(null);

// Verificando se o usuario selecionou pressionou o botao Open.
            if (response == JFileChooser.APPROVE_OPTION)
            {
                fileDir = chooser.getSelectedFile();

// Verificando se um arquivo foi selecionado.
                if (fileDir.isFile())
                {

// Tenta abrir arquivo para leitura de dados.
                    try
                    {
                        // Declaracao de variaveis a serem empregadas na
                        // leitura dos dados de um arquivo.
                        String nome;
                        String cargo;
                        double salario;

                        fileIn = new Scanner(new FileReader(fileDir));

                        while (fileIn.hasNextLine())
                        {
                            // Leitura dos dados do arquivo.
                            line = fileIn.nextLine();
                            nome = line.substring(0,23);
                            cargo = line.substring(24,39);
                        }
                    }
                }
            }
        }
    }
}

```

```

        salario = Double.parseDouble(line.substring(39,49).trim());

        // Colocando os dados na ultima linha da tabela de
        // entrada de dados.
        modeloEntrada.insertRow(jtableEntrada.getRowCount(),
        new Object[] {new String(nome), new String(cargo),
        new Double(salario)});
    }
    fileIn.close();
    output = "Dados recuperados com sucesso !";

}
// Caso o arquivo nao tenha sido encontrado.
catch(FileNotFoundException ex)
{
    JOptionPane.showMessageDialog(null, "Arquivo não encontrado!",
    "File Sizes", JOptionPane.INFORMATION_MESSAGE);

}

}

// Verificando se um diretorio foi selecionado.
else if (fileDir.isDirectory())
{
    output = "Diretório Selecionado. Selecione um arquivo";
} // end else.
// Caso em que nem um arquivo nem um diretorio foram selecionados.
else
{
    output = "Escolha inválida. Não é diretório nem arquivo.";
}

JOptionPane.showMessageDialog(null, output,
    "Recuperação de Dados",
JOptionPane.INFORMATION_MESSAGE);
}
}
} // FIM DO ACTIONPERFORMED.
} // FIM DA CLASSE TRATAARQMENU.

public static void main( String args[] )
{
    Exe31 jt1 = new Exe31();
}
}

```

Exercício 2: Explorar e entender as estruturas de um programa que emprega os seguintes elementos de interface gráfica: JTabbedPane (divisão da tela em painéis) e JTable (para armazenar dados e opções selecionadas em campos anteriores) em conjunto com a gravação de dados em arquivos (através de JMenu). A descrição dos elementos e a aparência da interface são ilustradas na Figura 2.1. Deve existir um arquivo CadastroEmpresa.txt no mesmo diretório do programa. O código para a GUI da Figura 2.1 está na Figura 2.2.

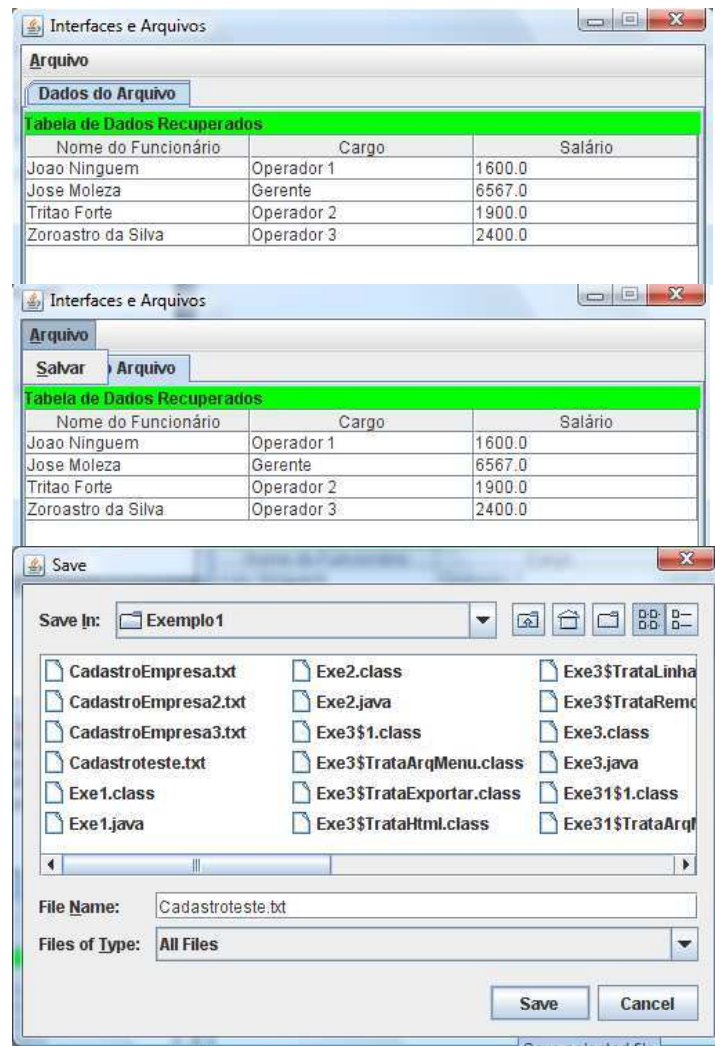


Figura 2.1: Interface do programa proposto.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import java.io.*;
import java.util.Scanner;
```

```
public class Exe32 extends JFrame
{
    private JMenu arqMenu;
    private JMenuItem abrirItem, salvarItem;
    private JMenuBar barra;
    private final int WIDTH = 500; // Largura da GUI.
    private final int HEIGHT = 550; // Altura da GUI.
    // Criando um JTabbedPane.
    JTabbedPane tabPane = new JTabbedPane();
    JPanel p1; // paineis para preencher os JTabbedPane().
    // Criando uma JTable onde ficaram armazenadas as informacoes de relatorio.
    JTable jTableEntrada;
    // Criando um modelo de JTable a partir do qual podemos manipular os dados
    // de JTable.
    DefaultTableModel modeloEntrada;
    // Itens para tratar a abertura e leitura dos dados de um arquivo texto.
    File fileDir; // Diretorio ou arquivo especificado pelo usuário.
    int response; // resposta do usuario na GUI.
    String output = ""; // lista de arquivos com tamanhos.
    JFileChooser chooser = new JFileChooser(".");
    // Itens para abrir e ler conteudo de arquivos ou abrir e escrever dados.
    Scanner fileIn;
    PrintWriter fileOut;
    String line;

    public Exe32()
    {
        setTitle("Interfaces e Arquivos");
        setSize(WIDTH, HEIGHT);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocation(100, 50);
        setResizable(false);
        createContents(); // Metodo que adiciona componentes.
        setVisible(true);
    }
}
```

```

public void createContents()
{
    //-----//
    // CRIANDO JMENU. //
    //-----//
    barra = new JMenuBar(); // a barra de menus é um contêiner

    // Cria JMenu arquivo.
    arqMenu = createJMenu1();
    // Anexa o arqMenu a Barra de Menus.
    barra.add(arqMenu);

    // Configura a barra de menus para o JFrame
    setJMenuBar(barra);
    //-----//

    //-----//
    // CRIANDO JPANELS. //
    //-----//
    // Cria o panel com informacoes carregadas em arquivo.
    p1 = createJPanel1();
    tabPane.addTab("Dados do Arquivo",null,p1,"Dados carregados");
    add(tabPane);

    //-----//
}

//-----//
// CRIANDO JMENU. //
//-----//
public JMenu createJMenu1()
{
    //-----//
    // Menu Arquivo //
    //-----//
    arqMenu = new JMenu("Arquivo");
    arqMenu.setMnemonic('A');
    salvarItem = new JMenuItem("Salvar");
    salvarItem.setMnemonic('S');
    salvarItem.addActionListener(new TrataArqMenu());

    // Adicionando itens ao JMenu arqMenu.
    arqMenu.add(salvarItem);

    // Retornando o arqMenu preenchido.
    return arqMenu;
}
//-----//

```

```

private JPanel createJPanel1()
{
    p1 = new JPanel();
    JLabel label2 = new JLabel("Tabela de Dados Recuperados");
    p1.setBackground(Color.GREEN);

    // Construindo a JTable.
    String[] colunas = new String[] {"Nome do
Funcionário","Cargo","Salário"};

    // Preenchendo a JTable com elementos.
    Object[][] dados = {
        {new String("Joao Ninguem"),new String("Operador 1"), new
Double(1600.0)},
        {new String("Jose Moleza"), new String("Gerente"), new
Double(6567.0)},
        {new String("Tritao Forte"), new String("Operador 2"), new
Double(1900.0)},
        {new String("Zoroastro da Silva"), new String("Operador 3"), new
Double(2400.0)}};

    // Ao inves de passar direto, colocamos os dados em um modelo
    modeloEntrada = new DefaultTableModel(dados, colunas);
    // e passamos o modelo para criar a jtable.
    jTableEntrada = new JTable( modeloEntrada );

    // Colocando a tabela dentro de uma barra de rolagem,
    // senão o cabeçalho não ira aparecer.
    JScrollPane jsp = new JScrollPane(jtableEntrada);
    jTableEntrada.setFillsViewportHeight(true);

    // Adicionando os elementos na interface.
    p1.setLayout(new BorderLayout());
    p1.add(label2,BorderLayout.NORTH);
    // Adicionando a barra de rolagem que contém a jtable.
    p1.add(jsp,BorderLayout.CENTER);
    return p1;
}

//-----//
// TRATAMENTO DE EVENTOS. //
//-----//

//-----//
// TRATAMENTO DE EVENTOS DO MENU ARQUIVO. //
//-----//

```

```

private class TrataArqMenu implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        //-----//
        // Tratamento de evento para salvar informacoes em arquivo.
        //-----//
        if (e.getSource() == salvarItem)
        {
            // Abrindo painel grafico para escolha de arquivos
            chooser.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
            response = chooser.showSaveDialog(null);

            // Verificando se o usuario selecionou pressionou o botao Open.
            if (response == JFileChooser.APPROVE_OPTION)
            {
                fileDir = chooser.getSelectedFile();
                // Criando o arquivo, se for possivel.
                try
                {
                    fileOut = new PrintWriter(fileDir);

                    // Declaracao de variaveis a serem empregadas na
                    // leitura dos dados de um arquivo.
                    String value = "";

                    // Percorrendo a tabela e colocando seus elementos
                    // em um arquivo.
                    for (int i=0; i < modeloEntrada.getRowCount(); i++)
                    {
                        for (int j=0; j < modeloEntrada.getColumnCount(); j++)
                        {
                            if (j == 0)
                                value = String.format("%24s",modeloEntrada.getValueAt(i,j));
                            else if (j == 1)
                                value+= String.format("%16s",modeloEntrada.getValueAt(i,j));
                            else if (j == 2)
                                value+= String.format("%10s",modeloEntrada.getValueAt(i,j));
                        }

                        // Salvando as informacoes em um arquivo.
                        fileOut.println(value);
                    }
                    fileOut.close();
                    output = "Dados salvos com sucesso !";
                }
            }
        }
    }
}

```

```

// Caso o arquivo nao tenha sido encontrado
catch (FileNotFoundException ex)
{
    output = "Arquivo não encontrado!";
}

JOptionPane.showMessageDialog(null, output,
    "Recuperação de Dados",
    JOptionPane.INFORMATION_MESSAGE);
} // FIM DO IF.

} // FIM DO IF SalvarItem.
//-----//

} // FIM DO ACTIONPERFORMED.
} // FIM DA CLASSE TRATAARQMENU.

//-----//
//-----//
//-----//

public static void main( String args[] )
{
    Exe32 jt1 = new Exe32();
}

}

```


Exercício 3: Explorar e entender as estruturas de um programa que emprega os seguintes elementos de interface gráfica: JTabbedPane (divisão da tela em painéis) e JTable (para armazenar dados e opções selecionadas em campos anteriores) tal que um JButton exporta informações de um JTable para outra. A descrição dos elementos e a aparência da interface são ilustradas na Figura 3.1.

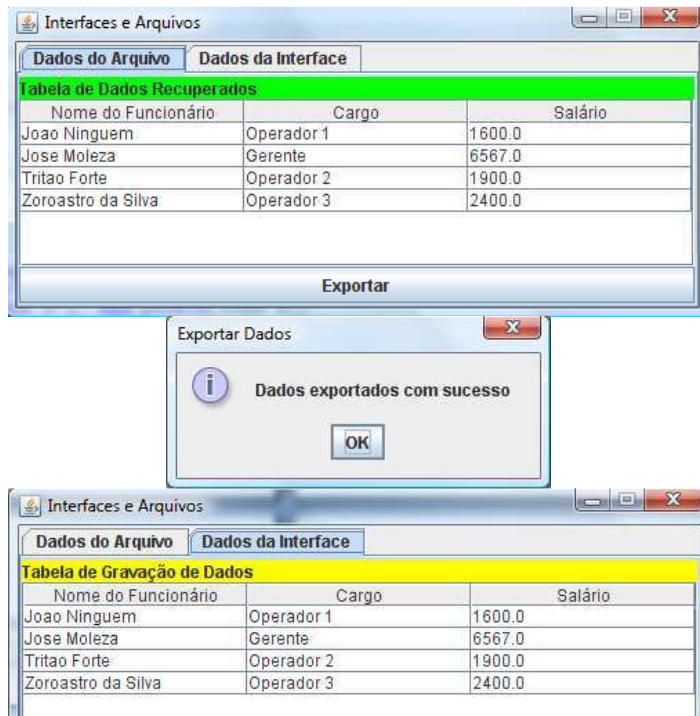


Figura 3.1: Comportamento das GUIs para o programa fornecido.

O código para a GUI da Figura 3.1 está na Figura 3.2.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import java.io.*;
import java.util.Scanner;
```

```
public class Exe33 extends JFrame
{

    private final int WIDTH = 500; // Largura da GUI.
    private final int HEIGHT = 550; // Altura da GUI.
    // Criando um JTabbedPane.
    JTabbedPane tabPane = new JTabbedPane();
    JPanel p1, p2; // paineis para preencher os JTabbedPane().
    // Criando uma JTable onde ficaram armazenadas as informacoes de relatorio.
    JTable jTableEntrada, jTableSaida;
    // Criando um modelo de JTable a partir do qual podemos manipular os dados
    // de JTable.
    DefaultTableModel modeloEntrada, modeloSaida;
    // Botao exporta dados de uma JTable para outra: da Entrada para Saida.
    JButton btnExp;

    public Exe33()
    {
        setTitle("Interfaces e Arquivos");
        setSize(WIDTH,HEIGHT);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocation(100,50);
        setResizable(false);
        createContents(); // Metodo que adiciona componentes.
        setVisible(true);
    }

    public void createContents()
    {

        //-----//
        // CRIANDO JPANELS.
        //-----//

        // Cria o panel com informacoes carregadas em arquivo.
        p1 = createJPanel1();

        // Cria o panel com informacoes a serem gravada em arquivo.
        p2 = createJPanel2();

        tabPane.addTab("Dados do Arquivo",null,p1,"Dados carregados");
        tabPane.addTab("Dados da Interface",null,p2,"Dados a serem gravados");
        add(tabPane);

        //-----//
    }
}
```

```

private JPanel createJPanel1()
{
    p1 = new JPanel();
    JLabel label2 = new JLabel("Tabela de Dados Recuperados");
    p1.setBackground(Color.GREEN);

    // Construindo a JTable.
    String[] colunas = new String[] {"Nome do Funcionário", "Cargo", "Salário"};

    Object[][] dados = {
        {new String("Joao Ninguem"), new String("Operador 1"), new
Double(1600.0)},
        {new String("Jose Moleza"), new String("Gerente"), new
Double(6567.0)},
        {new String("Tritao Forte"), new String("Operador 2"), new
Double(1900.0)},
        {new String("Zoroastro da Silva"), new String("Operador 3"), new
Double(2400.0)}};

    // Ao inves de passar direto, colocamos os dados em um modelo
    modeloEntrada = new DefaultTableModel(dados, colunas);
    // e passamos o modelo para criar a jtable.
    jTableEntrada = new JTable( modeloEntrada );

    // Colocando a tabela dentro de uma barra de rolagem,
    // senão o cabeçalho não ira aparecer.
    JScrollPane jsp = new JScrollPane(jTableEntrada);
    jTableEntrada.setFillsViewportHeight(true);

    // Criando botao para exportar dados para outra jtable.
    btnExp = new JButton("Exportar");
    btnExp.addActionListener(new TrataExportar());

    // Adicionando os elementos na interface.
    p1.setLayout(new BorderLayout());
    p1.add(label2, BorderLayout.NORTH);
    // Adicionando a barra de rolagem que contém a jtable.
    p1.add(jsp, BorderLayout.CENTER);
    // Adicionando botao para exportar dados de uma tabela para outra.
    p1.add(btnExp, BorderLayout.SOUTH);
    return p1;
}

```

```

private JPanel createJPanel2()
{
    p2 = new JPanel();
    JLabel label2 = new JLabel("Tabela de Gravação de Dados");
    p2.setBackground(Color.YELLOW);

    // Construindo a JTable.
    String[] colunas = new String[] {"Nome do Funcionário", "Cargo", "Salário"};

    Object[][] dados = new Object[][]{};

    // Ao inves de passar direto, colocamos os dados em um modelo
    modeloSaida = new DefaultTableModel(dados, colunas);
    // e passamos o modelo para criar a jtable.
    jTableSaida = new JTable( modeloSaida );

    // Colocando a tabela dentro de uma barra de rolagem,
    // senão o cabeçalho não ira aparecer.
    JScrollPane jsp = new JScrollPane(jTableSaida);
    jTableSaida.setFillsViewportHeight(true);

    // Adicionando os elementos na interface.
    p2.setLayout(new BorderLayout());
    p2.add(label2, BorderLayout.NORTH);
    p2.add(jsp, BorderLayout.CENTER);
    return p2;
}

//-----//
// TRATAMENTO DE EVENTOS DO BOTAO EXPORTAR. //
//-----//

private class TrataExportar implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        int initi = 0;
        // Verificando se ja existem elementos na tabela de saida de dados.
        if (modeloSaida.getRowCount() > 0)
            initi = modeloSaida.getRowCount();

        // Copiar elementos de uma tabela para outra.
        // Percorrendo a tabela e colocando seus elementos
        // em um arquivo.
        for (int i=0; i < modeloEntrada.getRowCount(); i++)
        {
            // Adicionando uma linha vazia na tabela de saida de dados.

```

```

modeloSaida.insertRow(jtableSaida.getRowCount(),
new Object[] {new String(""), new String(""),
new Double(0.0)});

// Preenchendo a linha da tabela de saida com dados da
// tabela de entrada.
for (int j=0; j < modeloEntrada.getColumnCount(); j++)
{
    // Colocando os dados na ultima linha da tabela de
    // Saida de dados.
    modeloSaida.setValueAt(modeloEntrada.getValueAt(i,j),initi+i,j);
}

JOptionPane.showMessageDialog(null, "Dados exportados com sucesso",
"Exportar Dados", JOptionPane.INFORMATION_MESSAGE);

// Mudando o foco para a segunda aba: dados exportados.
tabPane.setSelectedIndex(1);

}
}

//-----//
//-----//
//-----//

public static void main( String args[] )
{
    Exe33 jt1 = new Exe33();
}

}

```

Exercício 4: Explorar e entender as estruturas de um programa que emprega os seguintes elementos de interface gráfica: JTabbedPane (divisão da tela em painéis) e JTable (para armazenar dados e opções selecionadas em campos anteriores) tal que dois JButtons incluem e removem informações de uma JTable. A descrição dos elementos e a aparência da interface são ilustradas na Figura 4.1.



Figura 4.1: Comportamento das GUIs para o programa fornecido.

O código para a GUI da Figura 4.1 está na Figura 4.2.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import java.io.*;
import java.util.Scanner;

public class Exe34 extends JFrame
{
    private JMenu arqMenu;
    private JMenuItem abrirItem, salvarItem;
    private JMenuBar barra;
    private final int WIDTH = 500; // Largura da GUI.
    private final int HEIGHT = 550; // Altura da GUI.
    // Criando um JTabbedPane.
    JTabbedPane tabPane = new JTabbedPane();
    JPanel p2; // painéis para preencher os JTabbedPane().
    // Criando uma JTable onde ficaram armazenadas as informacoes de relatorio.
    JTable jTableSaida;
    // Criando um modelo de JTable a partir do qual podemos manipular os dados
    // de JTable.
    DefaultTableModel modeloSaida;
    // Botao de acao no JTable Saida: Adicionar ou Remover linhas.
    JButton btnAddL, btnRmvL;

    public Exe34()
    {
        setTitle("Interfaces e Arquivos");
        setSize(WIDTH,HEIGHT);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocation(100,50);
        setResizable(false);
        createContents(); // Metodo que adiciona componentes.
        setVisible(true);
    }

    public void createContents()
    {
        // CRIANDO JPANELS.
        // Cria o panel com informacoes a serem gravadas em arquivo.
        p2 = createJPanel2();
        tabPane.addTab("Dados da Interface",null,p2,"Dados a serem gravados");
        add(tabPane);

        //-----//
    }
}
```

```

private JPanel createJPanel2()
{
    p2 = new JPanel();
    JLabel label2 = new JLabel("Tabela de Gravação de Dados");
    p2.setBackground(Color.YELLOW);

    // Construindo a JTable.
    String[] colunas = new String[] {"Nome do Funcionário", "Cargo", "Salário"};

    Object[][] dados = {
        {new String("Joao Ninguem"), new String("Operador 1"), new
        Double(1600.0)},
        {new String("Jose Moleza"), new String("Gerente"), new
        Double(6567.0)},
        {new String("Tritao Forte"), new String("Operador 2"), new
        Double(1900.0)},
        {new String("Zoroastro da Silva"), new String("Operador 3"), new
        Double(2400.0)}};

    // Ao inves de passar direto, colocamos os dados em um modelo
    modeloSaida = new DefaultTableModel(dados, colunas);
    // e passamos o modelo para criar a jtable.
    jTableSaida = new JTable( modeloSaida );

    // Colocando a tabela dentro de uma barra de rolagem,
    // senão o cabeçalho não ira aparecer.
    JScrollPane jsp = new JScrollPane(jTableSaida);
    jTableSaida.setFillsViewportHeight(true);

    // Criando os botoes para adicionar e remover colunas na JTable Saida.
    btnAddL = new JButton("Adicionar Linhas");
    btnRmvL = new JButton("Remover Linhas");
    btnAddL.addActionListener(new TrataLinha());
    btnRmvL.addActionListener(new TrataLinha());

    // Adicionando os elementos na interface.
    p2.setLayout(new BorderLayout());
    p2.add(label2, BorderLayout.NORTH);
    p2.add(jsp, BorderLayout.CENTER);
    JPanel p21 = new JPanel();
    p21.setLayout(new FlowLayout());
    p21.add(btnAddL);
    p21.add(btnRmvL);
    p2.add(p21, BorderLayout.SOUTH);
    return p2;
}

```

```

//-----//
// TRATAMENTO DE EVENTOS: //
// BOTOES PARA ADICIONAR E REMOVER LINHAS. //
//-----//

private class TrataLinha implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        // Obtendo qual linha da tabela foi selecionada.
        int rowIndex = jTableSaida.getSelectedRow();

        // Verificando se realmente uma coluna foi selecionada.
        if (rowIndex != -1)
        {
            // ADICIONANDO LINHAS NA TABELA.
            if (e.getSource() == btnAddL)
            {
                addRow(rowIndex);
            }

            // REMOVENDO AS LINHAS DA TABELA.
            if (e.getSource() == btnRmvL)
            {
                removeRow(rowIndex);
            }
        }
        // Nenhuma linha foi selecionada !
        else
        {
            // Verificando se eh porque todas linhas foram eliminadas.
            if (modeloSaida.getRowCount() == 0)
            {
                // Adicionar linha na primeira linha da tabela.
                addRow(0);
            }
        }

        // METODO ESPECIFICO PARA ADICIONAR LINHAS NA TABELA.
        public void addRow(int vColIndex)
        {
            // Remove a linha da tabela sem nenhum conteúdo.
            modeloSaida.insertRow(vColIndex, new Object[]{});
            // Atualiza as informacoes da tabela.
            modeloSaida.fireTableStructureChanged();
        }
    }
}

```

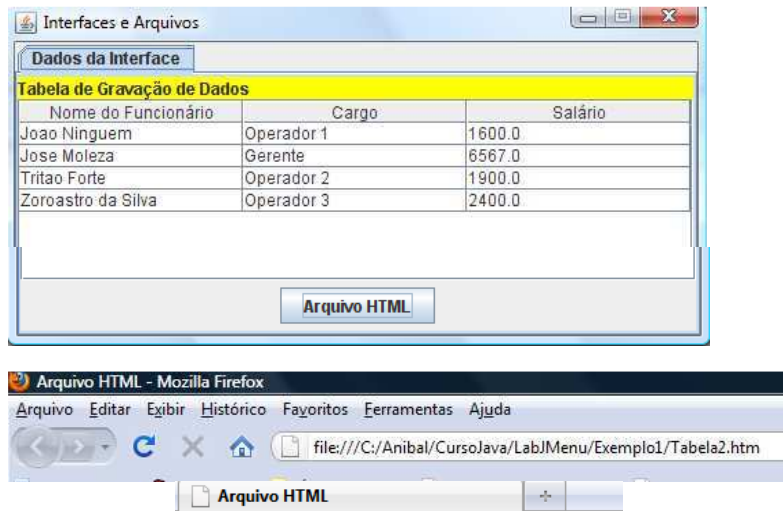
```
// METODO ESPECIFICO PARA REMOVER AS LINHAS DA TABELA.
public void removeRow(int vColIndex)
{
    // Remove a linha da tabela.
    modeloSaida.removeRow(vColIndex);
    // Atualiza as informacoes da tabela.
    modeloSaida.fireTableStructureChanged();
}

}

//-----//
//-----//
//-----//

public static void main( String args[] )
{
    Exe34 jt1 = new Exe34();
}
}
```

Exercício 5: Explorar e entender as estruturas de um programa que emprega os seguintes elementos de interface gráfica: JTabbedPane (divisão da tela em painéis) e JTable (para armazenar dados e opções selecionadas em campos anteriores) tal que um JButton armazena os dados em um arquivo HTML. A descrição dos elementos e a aparência da interface são ilustradas na Figura 5.1.



Dados da Tabela

Tabela com os dados de saída.

	Coluna 1	Coluna 2	Coluna 3
Linha 1	Joao Ninguem	Operador 1	1600.0
Linha 2	Jose Moleza	Gerente	6567.0
Linha 3	Tritao Forte	Operador 2	1900.0
Linha 4	Zoroastro da Silva	Operador 3	2400.0

Figura 5.1: Comportamento das GUIs para o programa fornecido.

O código para a GUI da Figura 5.1 está na Figura 5.2.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import java.io.*;
import java.util.Scanner;

public class Exe35 extends JFrame
{
    private final int WIDTH = 500; // Largura da GUI.
    private final int HEIGHT = 550; // Altura da GUI.
    // Criando um JTabbedPane.
    JTabbedPane tabPane = new JTabbedPane();
    JPanel p2; // painéis para preencher os JTabbedPane().
    // Criando uma JTable onde ficaram armazenadas as informacoes de relatorio.
    JTable jTableSaida;
    // Criando um modelo de JTable a partir do qual podemos manipular os dados
    // de JTable.
    DefaultTableModel modeloSaida;
    // Botao de acao no JTable Saida: Salvar em arquivo html.
    JButton btnHtml;
    // Itens para tratar a abertura e leitura dos dados de um arquivo texto.
    File fileDir; // Diretorio ou arquivo especificado pelo usuário.
    int response; // resposta do usuario na GUI.
    String output = ""; // lista de arquivos com tamanhos.
    JFileChooser chooser = new JFileChooser(".");
    // Itens para abrir e ler conteudo de arquivos ou abrir e escrever dados.
    Scanner fileIn;
    PrintWriter fileOut;
    String line;

    public Exe35()
    {
        setTitle("Interfaces e Arquivos");
        setSize(WIDTH,HEIGHT);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocation(100,50);
        setResizable(false);
        createContents(); // Metodo que adiciona componentes.
        setVisible(true);
    }
}
```

```

public void createContents()
{
    // Cria o panel com informacoes a serem gravada em arquivo.
    p2 = createJPanel2();
    tabPane.addTab("Dados da Interface", null, p2, "Dados a serem gravados");
    add(tabPane);
}

private JPanel createJPanel2()
{
    p2 = new JPanel();
    JLabel label2 = new JLabel("Tabela de Gravação de Dados");
    p2.setBackground(Color.YELLOW);

    // Construindo a JTable.
    String[] colunas = new String[] {"Nome do Funcionário", "Cargo", "Salário"};
    Object[][] dados = {
        {new String("Joao Ninguem"), new String("Operador 1"), new
        Double(1600.0)},
        {new String("Jose Moleza"), new String("Gerente"), new
        Double(6567.0)},
        {new String("Tritao Forte"), new String("Operador 2"), new
        Double(1900.0)},
        {new String("Zoroastro da Silva"), new String("Operador 3"), new
        Double(2400.0)}};

    // Ao inves de passar direto, colocamos os dados em um modelo
    modeloSaida = new DefaultTableModel(dados, colunas);
    // e passamos o modelo para criar a jtable.
    jTableSaida = new JTable(modeloSaida);

    // Colocando a tabela dentro de uma barra de rolagem,
    // senão o cabeçalho não ira aparecer.
    JScrollPane jsp = new JScrollPane(jTableSaida);
    jTableSaida.setFillsViewportHeight(true);

    // Criando os botoes para salvar o conteudo da JTable em um arquivo HTML.
    btnHtml = new JButton("Arquivo HTML");
    btnHtml.addActionListener(new TrataHtml());

    // Adicionando os elementos na interface.
    p2.setLayout(new BorderLayout());
    p2.add(label2, BorderLayout.NORTH);
    p2.add(jsp, BorderLayout.CENTER);
    JPanel p21 = new JPanel();
    p21.setLayout(new FlowLayout());
    p21.add(btnHtml);
    p2.add(p21, BorderLayout.SOUTH);
    return p2;
}

```

```

//-----//
// TRATAMENTO DE EVENTOS DO BOTAO: DADOS DE TABELA EM HTML. //
//-----//
private class TrataHtml implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        // Passando a informacao contida na Tabela para um arquivo HTML.
        // Criando o arquivo, se for possivel.
        try
        {
            String fileNameOut = "Tabela2.htm";
            PrintWriter fileOut = new PrintWriter(fileNameOut);

            // Construindo o cabeçalho do arquivo HTML.
            // Inclusao de "tags" que permitem a inclusao do
            // conteudo no arquivo.
            //-----//
            fileOut.println("<html>");
            fileOut.println("<head>");
            fileOut.println("<title>" + " Arquivo HTML");
            fileOut.println("</head>");
            // Começo do corpo do html:
            // onde serao colocadas as informacoes da tabela.
            fileOut.println("<body>");
            fileOut.println("<h1>Dados da Tabela</h1>");
            //-----//

            //-----//
            // "Tag" adicional para imprimir os dados em formato Tabela de HTML.
            //-----//
            fileOut.println("<table>");
            // Legenda da Tabela.
            fileOut.println("<caption>" + "Tabela com os dados de
            saida." + "</caption>");

            // Inicio da tabela.
            fileOut.println("<tr>");
            // Inserindo celula vazia para alinhar cabeçalho das colunas.
            fileOut.println("<td> </td>");
            // Cabeçalho de cada coluna da tabela html.
            for (int j=0; j < modeloSaida.getColumnCount(); j++)
            {
                fileOut.println("<th> Coluna " + (j+1) + "</th>");
            }
            fileOut.println("</tr>");
            //-----//

```



```

// Declaracao de variaveis a serem empregadas na
// leitura dos dados de um arquivo.
String value = "";

// Percorrendo a tabela e colocando seus elementos
// em um arquivo.
for (int i=0; i < modeloSaida.getRowCount(); i++)
{
    fileOut.println("<tr>"); // Comeca uma linha da tabela html.
    // Comeca o cabecalho da linha da tabela html.
    fileOut.println("<th> Linha "+(i+1)+"</th>");
    for (int j=0; j < modeloSaida.getColumnCount(); j++)
    {
        if (j == 0)
            value = String.format("%24s",modeloSaida.getValueAt(i,j));
        else if (j == 1)
            value = String.format("%16s",modeloSaida.getValueAt(i,j));
        else if (j == 2)
            value = String.format("%10s",modeloSaida.getValueAt(i,j));

        // Salvando as informacoes no arquivo html.
        fileOut.println("<td>"+value+"</td>");
    }
    fileOut.println("</tr>"); // termina uma linha da tabela html.
}

// Apos imprimir todos os dados, fechar a "tag" table do html.
fileOut.println("</table>");

// Codigo necessario para terminar o arquivo HTML.
fileOut.println("</body>");
fileOut.println("</html>");
//-----//

fileOut.close();
// Mensagem para o usuario saber se o arquivo foi gravado.
output = "Dados salvos com sucesso !";

}

// Caso o arquivo nao tenha sido encontrado
catch (FileNotFoundException ex)
{
    // Mensagem caso o arquivo nao possa ter sido criado.
    output = "Não foi possivel criar o arquivo!";
}

```

```

JOptionPane.showMessageDialog(null, output,
    "Gravar Dados em HTML",
    JOptionPane.INFORMATION_MESSAGE);
}

//-----//
//-----//
//-----//

public static void main( String args[] )
{
    Exe35 jt1 = new Exe35();
}

```

Exercício 6: Reunir todos os elementos de interface gráfica (JMenus, JTable, JTabbedPane e JButton) e suas respectivas funcionalidades em único programa tal como descrito nas Figura 6.1 e 6.2.



Figura 6.1: Componentes GUIs para o programa fornecido.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import java.io.*;
import java.util.Scanner;

public class Exe3 extends JFrame
{
    private JMenu arqMenu;
    private JMenuItem abrirItem, salvarItem;
    private JMenuBar barra;
    private final int WIDTH = 500; // Largura da GUI.
    private final int HEIGHT = 550; // Altura da GUI.
    // Criando um JTabbedPane.
    JTabbedPane tabPane = new JTabbedPane();
    JPanel p1, p2; // paineis para preencher os JTabbedPane().
    // Criando uma JTable onde ficaram armazenadas as informacoes de relatorio.
    JTable jTableEntrada, jTableSaida;
    // Criando um modelo de JTable a partir do qual podemos manipular os dados
    // de JTable.
    DefaultTableModel modeloEntrada, modeloSaida;
    // Botao para exportar dados de uma JTable para outra: da Entrada para
    Saida.
    JButton btnExp;
    // Botao de acao no JTable Saida: Adicionar ou Remover linhas com
    informacoes.
    JButton btnAddL, btnRmvL;
    // Botao de acao no JTable Saida: Salvar em arquivo html.
    JButton btnHtml;
    // Itens para tratar a abertura e leitura dos dados de um arquivo texto.
    File fileDir; // Diretorio ou arquivo especificado pelo usuário.
    int response; // resposta do usuario na GUI.
    String output = ""; // lista de arquivos com tamanhos.
    JFileChooser chooser = new JFileChooser(".");
    // Itens para abrir e ler conteudo de arquivos ou abrir e escrever dados.
    Scanner fileIn;
    PrintWriter fileOut;
    String line;

    public Exe3()
    {
        setTitle("Interfaces e Arquivos");
        setSize(WIDTH,HEIGHT);
    }
}
```

```

setDefaultCloseOperation(EXIT_ON_CLOSE);
setLocation(100,50);
setResizable(false);
createContents(); // Metodo que adiciona componentes.
setVisible(true);
}

public void createContents()
{

    //-----//
    // CRIANDO JMENU. //
    //-----//

    barra = new JMenuBar(); // a barra de menus é um contêiner

    // Cria JMenu arquivo.
    arqMenu = createJMenu1();
    // Anexa o arqMenu a Barra de Menus.
    barra.add(arqMenu);

    // Configura a barra de menus para o JFrame
    //-----//
    setJMenuBar(barra);

    //-----//

    //-----//
    // CRIANDO JPANELS. //
    //-----//

    // Cria o panel com informacoes carregadas em arquivo.
    p1 = createJPanel1();

    // Cria o panel com informacoes a serem gravada em arquivo.
    p2 = createJPanel2();

    tabPane.addTab("Dados do Arquivo",null,p1,"Dados carregados");
    tabPane.addTab("Dados da Interface",null,p2,"Dados a serem
gravados");
    add(tabPane);

    //-----//
}

```

```

//-----//
// CRIANDO JMENU. //
//-----//

public JMenu createJMenu1()
{

    //-----//
    // Menu Arquivo //
    //-----//
    arqMenu = new JMenu("Arquivo");
    arqMenu.setMnemonic('A');
    abrirItem = new JMenuItem("Abrir...");
    abrirItem.setMnemonic('b');
    abrirItem.addActionListener(new TrataArqMenu());
    salvarItem = new JMenuItem("Salvar");
    salvarItem.setMnemonic('S');
    salvarItem.addActionListener(new TrataArqMenu());

    // Adicionando itens ao JMenu arqMenu.
    arqMenu.add(abrirItem);
    arqMenu.add(salvarItem);

    // Retornando o arqMenu preenchido.
    return arqMenu;
    //-----//
}

private JPanel createJPanel1()
{
    p1 = new JPanel();
    JLabel label2 = new JLabel("Tabela de Dados Recuperados");
    p1.setBackground(Color.GREEN);

    // Construindo a JTable.
    String[] colunas = new String[] {"Nome do Funcionário","Cargo","Salário"};

    Object[][] dados = new Object[][]{};

    // Ao inves de passar direto, colocamos os dados em um modelo
    modeloEntrada = new DefaultTableModel(dados, colunas);
}

```

```

// e passamos o modelo para criar a jTable.
jtableEntrada = new JTable( modeloEntrada );

//jtableEntrada.setAutoCreateRowSorter(true);
//Insert last position
//modeloEntrada.insertRow(jtableEntrada.getRowCount(),new
Object[]{"Sushil",new Boolean(true), new Double(120.0)});

// Colocando a tabela dentro de uma barra de rolagem,
// senão o cabeçalho não ira aparecer.
JScrollPane jsp = new JScrollPane(jtableEntrada);
jtableEntrada.setFillsViewportHeight(true);

// Criando botao para exportar dados para outra jTable.
btnExp = new JButton("Exportar");
btnExp.addActionListener(new TrataExportar());

// Adicionando os elementos na interface.
p1.setLayout(new BorderLayout());
p1.add(label2,BorderLayout.NORTH);
// Adicionando a barra de rolagem que contém a jTable.
p1.add(jsp,BorderLayout.CENTER);
// Adicionando botao para exportar dados de uma tabela para outra.
p1.add(btnExp,BorderLayout.SOUTH);
return p1;
}

private JPanel createJPanel2()
{
    p2 = new JPanel();
    JLabel label2 = new JLabel("Tabela de Gravação de Dados");
    p2.setBackground(Color.YELLOW);

    // Construindo a JTable.
    String[] colunas = new String[] {"Nome do Funcionário","Cargo","Salário"};

    Object[][] dados = new Object[][]{};

    // Ao inves de passar direto, colocamos os dados em um modelo
    modeloSaida = new DefaultTableModel(dados, colunas);
    // e passamos o modelo para criar a jTable
    jTableSaida = new JTable( modeloSaida );

    // Colocando a tabela dentro de uma barra de rolagem,
    // senão o cabeçalho não ira aparecer.

```

```

JScrollPane jsp = new JScrollPane(jtableSaida);
jtableSaida.setFillsViewportHeight(true);

// Criando os botoes para adicionar e remover colunas na JTable Saida.
// Ou ainda, salvar o conteudo da JTable em um arquivo HTML.
btnAddL = new JButton("Adicionar Linhas");
btnRmvL = new JButton("Remover Linhas");
btnHtml = new JButton("Arquivo HTML");
btnAddL.addActionListener(new TrataLinha());
btnRmvL.addActionListener(new TrataLinha());
btnHtml.addActionListener(new TrataHtml());

// Adicionando os elementos na interface.
p2.setLayout(new BorderLayout());
p2.add(label2,BorderLayout.NORTH);
p2.add(jsp,BorderLayout.CENTER);
JPanel p21 = new JPanel();
p21.setLayout(new FlowLayout());
p21.add(btnAddL);
p21.add(btnRmvL);
p21.add(btnHtml);
p2.add(p21,BorderLayout.SOUTH);
return p2;
}

//-----//
//-----//
// TRATAMENTO DE EVENTOS. //
//-----//

//-----//
// TRATAMENTO DE EVENTOS DO MENU ARQUIVO. //
//-----//

private class TrataArqMenu implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {

        //-----//
        // Tratando eventos de abertura de arquivos e recuperacao de dados.
        //-----//
        if (e.getSource() == abrirItem)
        {

```

```

// Abrindo painel grafico para escolha de arquivos
chooser.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
response = chooser.showOpenDialog(null);

// Verificando se o usuario selecionou pressionou o botao Open.
if (response == JFileChooser.APPROVE_OPTION)
{
    fileDir = chooser.getSelectedFile();

    // Verificando se um arquivo foi selecionado.
    if (fileDir.isFile())
    {
        //output += String.format("%-25s%12s%n",
        //fileDir.getName(), fileDir.length() + "bytes");

        // Tenta abrir arquivo para leitura de dados.
        try
        {
            // Declaracao de variaveis a serem empregadas na
            // leitura dos dados de um arquivo.
            String nome;
            String cargo;
            double salario;

            fileIn = new Scanner(new FileReader(fileDir));

            while (fileIn.hasNextLine())
            {
                // Leitura dos dados do arquivo.
                line = fileIn.nextLine();

                nome = line.substring(0,23);
                cargo = line.substring(24,39);
                salario = Double.parseDouble(line.substring(39,49).trim());

                // Colocando os dados na ultima linha da tabela de
                // entrada de dados.
                modeloEntrada.insertRow(jtableEntrada.getRowCount(),
                new Object[] {new String(nome), new String(cargo),
                new Double(salario)});

            }
            fileIn.close();
            output = "Dados recuperados com sucesso !";

```

```

        }
        // Caso o arquivo nao tenha sido encontrado
        catch(FileNotFoundException ex)
        {
            JOptionPane.showMessageDialog(null, "Arquivo não
            encontrado!",
            "File Sizes", JOptionPane.INFORMATION_MESSAGE);
        }

    }

    // Verificando se um diretorio foi selecionado.
    else if (fileDir.isDirectory())
    {
        output = "Diretório Selecionado. Selecione um
        arquivo";

    } // end else.

    // Caso em que nem um arquivo nem um diretorio foram
    selecionados.
    else
    {
        output = "Escolha inválida. Não é diretório nem
        arquivo.";
    }

    JOptionPane.showMessageDialog(null, output,
    "Recuperação de Dados",
    JOptionPane.INFORMATION_MESSAGE);
}

//-----//
// Tratamento de evento para salvar informacoes em arquivo.
//-----//
else if (e.getSource() == salvarItem)
{
    // Abrindo painel grafico para escolha de arquivos
    chooser.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
    response = chooser.showSaveDialog(null);

    // Verificando se o usuario selecionou pressionou o botao Open.
    if (response == JFileChooser.APPROVE_OPTION)

```

```

    {
        fileDir = chooser.getSelectedFile();
        // Criando o arquivo, se for possivel.
        try
        {
            fileOut = new PrintWriter(fileDir);

            // Declaracao de variaveis a serem empregadas na
            // leitura dos dados de um arquivo.
            String value = "";

            // Percorrendo a tabela e colocando seus elementos
            // em um arquivo.
            for (int i=0; i < modeloEntrada.getRowCount(); i++)
            {
                for (int j=0; j < modeloEntrada.getColumnCount();
j++)
                {
                    if (j == 0)
                        value = String.format("%24s",modeloEntrada.getValueAt(i,j));
                    else if (j == 1)
                        value +=
String.format("%16s",modeloEntrada.getValueAt(i,j));
                    else if (j == 2)
                        value +=
String.format("%10s",modeloEntrada.getValueAt(i,j));
                }

                // Salvando as informacoes em um arquivo.
                fileOut.println(value);
            }
            fileOut.close();
            output = "Dados salvos com sucesso !";

        }
        // Caso o arquivo nao tenha sido encontrado
        catch(FileNotFoundException ex)
        {
            output = "Arquivo não encontrado!";
        }

        JOptionPane.showMessageDialog(null, output,
"Recuperação de Dados",
JOptionPane.INFORMATION_MESSAGE);

    } // FIM DO IF.

```

```

    }
    //-----//

    } // FIM DO ACTIONPERFORMED.
    } // FIM DA CLASSE TRATAARQMENU.

    //-----//
    //-----//
    //-----//

    //-----//
    // TRATAMENTO DE EVENTOS DO BOTAO EXPORTAR.
    //-----//

private class TrataExportar implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        int initi = 0;
        // Verificando se ja existem elementos na tabela de saida de dados.
        if (modeloSaida.getRowCount() > 0)
            initi = modeloSaida.getRowCount();

        // Copiar elementos de uma tabela para outra.
        // Percorrendo a tabela e colocando seus elementos
        // em um arquivo.
        for (int i=0; i < modeloEntrada.getRowCount(); i++)
        {
            // Adicionando uma linha vazia na tabela de saida de dados.
            modeloSaida.insertRow(jtableSaida.getRowCount(),
            new Object[] {new String(""), new String(""),
            new Double(0.0)});

            // Preenchendo a linha da tabela de saida com dados da
            // tabela de entrada.
            for (int j=0; j < modeloEntrada.getColumnCount(); j++)
            {
                // Colocando os dados na ultima linha da tabela de
                // Saida de dados.
                modeloSaida.setValueAt(modeloEntrada.getValueAt(i,j),initi+i,j);
            }
        }

        JOptionPane.showMessageDialog(null, "Dados exportados com sucesso",
"Exportar Dados", JOptionPane.INFORMATION_MESSAGE);
    }
}

```

```

// Mudando o foco para a segunda aba: dados exportados.
tabPane.setSelectedIndex(1);

}
}

//-----//
//-----//
//-----//

//-----//
// TRATAMENTO DE EVENTOS DO BOTOES PARA ADICIONAR E REMOVER
LINHAS. //
//-----//

private class TrataLinha implements ActionListener
{

    public void actionPerformed(ActionEvent e)
    {
        // Obtendo qual linha da tabela foi selecionada.
        int rowIndex = jTableSaida.getSelectedRow();

        // Verificando se realmente uma coluna foi selecionada.
        if (rowIndex != -1)
        {

            // ADICIONANDO LINHAS NA TABELA.
            if (e.getSource() == btnAddL)
            {
                addRow(rowIndex);
            }

            // REMOVENDO AS LINHAS DA TABELA.
            if (e.getSource() == btnRmvL)
            {
                removeRow(rowIndex);
            }

        }
        // Nenhuma coluna foi selecionada !
        else
        {
            // Verificando se eh porque todas linhas foram eliminadas.
            if (modeloSaida.getRowCount() == 0)
            // Adicionar linha na primeira linha da tabela.
                addRow(0);
        }
    }
}

```

```

}

}

// METODO ESPECIFICO PARA ADICIONAR LINHAS NA TABELA.
public void addRow(int vColIndex)
{
    // Remove a linha da tabela sem nenhum conteúdo.
    modeloSaida.insertRow(vColIndex, new Object[]{});
    // Atualiza as informacoes da tabela.
    modeloSaida.fireTableStructureChanged();
}

// METODO ESPECIFICO PARA REMOVER AS LINHAS DA TABELA.
public void removeRow(int vColIndex)
{
    // Remove a linha da tabela.
    modeloSaida.removeRow(vColIndex);
    // Atualiza as informacoes da tabela.
    modeloSaida.fireTableStructureChanged();
}

}

//-----//
//-----//
//-----//

//-----//
// TRATAMENTO DE EVENTOS DO BOTOES PARA SALVAR DADOS DE TABELA
EM HTML.//
//-----//

private class TrataHtml implements ActionListener
{

    public void actionPerformed(ActionEvent e)
    {
        // Passando a informacao contida na Tabela para um arquivo HTML.
        // Criando o arquivo, se for possivel.
        try
        {
            String fileNameOut = "Tabela2.htm";
            PrintWriter fileOut = new PrintWriter(fileNameOut);

            // Construindo o cabecalho do arquivo HTML.

```

```

// Inclusao de "tags" que permitem a inclusao do
// conteudo no arquivo.
//-----//
fileOut.println("<html>");
fileOut.println("<head>");
fileOut.println("<title>" + " Arquivo HTML");
fileOut.println("</head>");
// Comeco do corpo do html: onde serao colocadas as
informacoes da tabela.
fileOut.println("<body>");
fileOut.println("<h1>Dados da Tabela</h1>");
//-----//

//-----//
// "Tag" adicional para imprimir os dados em formato
Tabela de HTML.
//-----//
fileOut.println("<table>");
// Legenda da Tabela.
fileOut.println("<caption>"+"Tabela com os dados de
saida."+"</caption>");

// Inicio da tabela.
fileOut.println("<tr>");
// Inserindo celula vazia para alinhar cabecalho das
colunas.
fileOut.println("<td> </td>");
// Cabecalho de cada coluna da tabela html.
for (int j=0; j < modeloSaida.getColumnCount(); j++)
{
    fileOut.println("<th> Coluna "+(j+1)+"</th>");
}
fileOut.println("</tr>");
//-----//

// Declaracao de variaveis a serem empregadas na
// leitura dos dados de um arquivo.
String value = "";

// Percorrendo a tabela e colocando seus elementos
// em um arquivo.
for (int i=0; i < modeloSaida.getRowCount(); i++)
{
    fileOut.println("<tr>"); // Comeca uma linha da tabela
html.
    fileOut.println("<th> Linha "+(i+1)+"</th>"); //
Comeca o cabecalho da linha da tabela html.

```

```

        for (int j=0; j < modeloSaida.getColumnCount(); j++)
        {
            if (j == 0)
                value = String.format("%24s",modeloSaida.getValueAt(i,j));
            else if (j == 1)
                value = String.format("%16s",modeloSaida.getValueAt(i,j));
            else if (j == 2)
                value = String.format("%10s",modeloSaida.getValueAt(i,j));

            // Salvando as informacoes no arquivo html.
            fileOut.println("<td>" +value+"</td>");
        }
        fileOut.println("</tr>"); // termina uma linha da tabela html.

    }

    // Apos imprimir todos os dados, fechar a "tag" table do
html.
    fileOut.println("<table>");

    //Codigo necessario para terminar o arquivo HTML.
    fileOut.println("</body>");
    fileOut.println("</html>");
    //-----//

    fileOut.close();

    // Mensagem para o usuario saber se o arquivo foi
gravado.
    output = "Dados salvos com sucesso !";

}
// Caso o arquivo nao tenha sido encontrado
catch(FileNotFoundException ex)
{
    // Mensagem caso o arquivo nao possa ter sido criado.
    output = "Não foi possivel criar o arquivo!";

}

JOptionPane.showMessageDialog(null, output,
    "Gravar      Dados      em      HTML",
JOptionPane.INFORMATION_MESSAGE);

}

```



```
}  
//-----//  
//-----//  
//-----//
```

```
public static void main( String args[] )  
{  
    Exe3 jt1 = new Exe3();  
}
```

```
}
```

Exercício 7: Explorar e entender as estruturas de um programa que emprega os seguintes elementos de interface gráfica: JTabbedPane (divisão da tela em painéis), JTable (para armazenar dados e opções selecionadas em campos anteriores), JTree (árvore com opções pré-definidas) e um JButton tal que seleciona opções dadas na JTree. A descrição dos elementos e a aparência da interface são ilustradas na Figura 7.1.

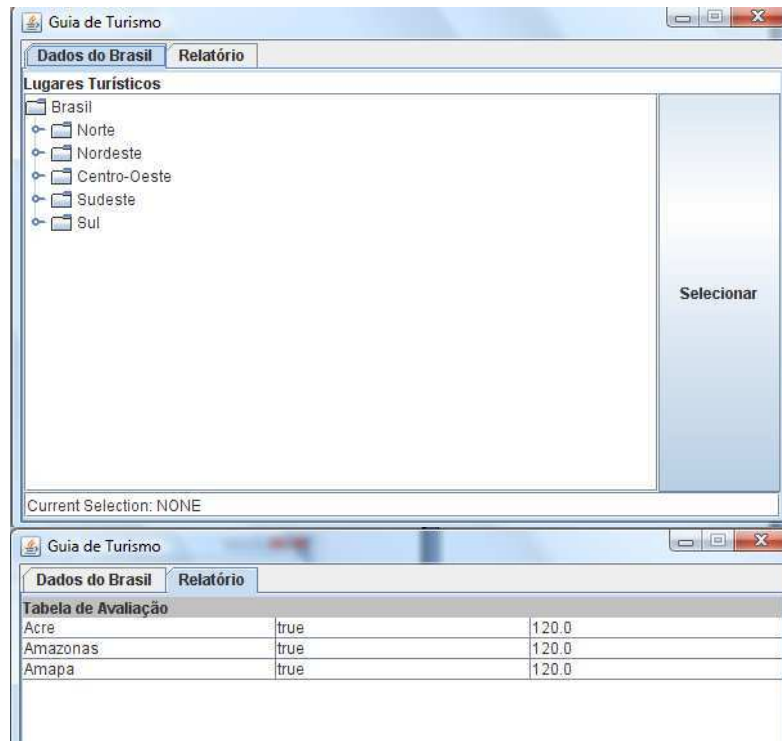


Figura 7.1: Comportamento das GUIs para o programa fornecido.

O código para a GUI da Figura 7.1 está na Figura 7.2.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class Exe4 extends JFrame implements TreeSelectionListener,
ActionListener
{
    // Criando um JTabbedPane.
    JTabbedPane tabPane = new JTabbedPane();
    JPanel p1, p2;
    JTextField currentSelectionField;
    // Criando um JTree.
    JTree arvore;
    // Criando JTextField para colocar em p2.
    JTextField f[] = new JTextField[5];
    //-----//
    // Criando a árvore e seus respectivos nós.
    // Nó raiz.
    DefaultMutableTreeNode pais = new DefaultMutableTreeNode("Brasil");
    // Criando os nós secundários.
    // Nomes das regioes a serem usadas na criacao de cada no de regioao.
    String nomesR[] = {"Norte", "Nordeste", "Centro-Oeste", "Sudeste", "Sul"};
    // Matriz de nomes dos estados a ser usado em conjunto com o vetor de
    nomes das regioes.
    String estadosR0[] = { "Acre", "Amazonas", "Amapa", "Para", "Roraima",
    "Rondonia"};
    String estadosR1[] = {"Alagoas", "Bahia", "Ceará", "Maranhão", "Piaui",
    "Pernambuco", "Rio Grande do Norte"};
    String estadosR2[] = {"Goiás", "Mato Grosso", "Mato Grosso do Sul",
    "Tocantins", "Distrito Federal"};
    String estadosR3[] = {"Espírito Santo", "Minas Gerais", "Rio de Janeiro", "São
    Paulo"};
    String estadosR4[] = {"Paraná", "Rio Grande do Sul", "Santa Catarina"};

    // Vetor de nós das regiões.
    DefaultMutableTreeNode regioao[] = new
    DefaultMutableTreeNode[nomesR.length];
    // Vetor de nós dos estados.
    DefaultMutableTreeNode estados[] = new DefaultMutableTreeNode[30];
    // Numero de estados para cada regioao.
    int nestados[] = {6, 7, 5, 4, 3};
    // Botao de acao para selecionar regioao e estado e outro para gerar relatorio.
    JButton botao1, botao2;
    // Variavel que conta o numero de itens selecionados da arvore.
```

```

int number = 0;
// Criando uma JTable onde ficaram armazenadas as informacoes de
relatorio.
JTable jTable;
// Criando um modelo de JTable a partir do qual podemos manipular os
dados
// de JTable.
DefaultTableModel modelo;

// Configura a GUI
public Exe4()
{
    setTitle("Guia de Turismo");
    setSize(600,400);
    setLocation(100,50);
    setResizable(false);
    createContents();
    setVisible(true);
}

public void createContents()
{
    // Cria o panel a ser mostrado na primeira aba: "Dados do Brasil".
    p1 = createJPanel1();

    // Cria o panel a ser mostrado na segunda aba: "Relatorio".
    p2 = createJPanel2();

    tabPane.addTab("Dados do Brasil",null,p1,"Dados turísticos");
    tabPane.addTab("Relatório",null,p2,"Tabela de Precos dos lugares");
    add(tabPane);
}

private JPanel createJPanel1()
{
    p1 = new JPanel();
    // Criando os elementos gráficos.
    // Texto de legenda.
    JLabel label1 = new JLabel("Lugares Turísticos");
    currentSelectionField = new JTextField("Current Selection: NONE");

    int k = 0;
    // Laco para efetivar a construcao do vetor de regioes.
    for (int i = 0; i < nomesR.length; i++)
    {

```

```

// Criacao efetiva dos nós com os nomes contidos em nomesR.
regiao[i] = new DefaultMutableTreeNode(nomesR[i]);
// Adicionando os nós secundários ao nó raiz.
pais.add(regiao[i]);

// Criacao efetiva dos nós com os nomes contidos em estados.
for (int j=0; j < nestados[i]; j++)
{
    if (i == 0)
        // Criacao efetiva dos nós com os nomes contidos em nomesR.
        estados[k] = new DefaultMutableTreeNode(estadosR0[j]);
    if (i == 1)
        // Criacao efetiva dos nós com os nomes contidos em nomesR.
        estados[k] = new DefaultMutableTreeNode(estadosR1[j]);
    if (i == 2)
        // Criacao efetiva dos nós com os nomes contidos em nomesR.
        estados[k] = new DefaultMutableTreeNode(estadosR2[j]);
    if (i == 3)
        // Criacao efetiva dos nós com os nomes contidos em nomesR.
        estados[k] = new DefaultMutableTreeNode(estadosR3[j]);
    if (i == 4)
        // Criacao efetiva dos nós com os nomes contidos em nomesR.
        estados[k] = new DefaultMutableTreeNode(estadosR4[j]);

    // Adicionando eventos nas opcoes de estado da arvore.
    // estados[k].addActionListener(new TrataEstado());

    // Adicionando estados[i] em cada regiao[i].
    regiao[i].add(estados[k]);

    // Criando mais um elemento no vetor estados[k].
    k++;
}

}

// Definindo com inserir os elementos graficos na interface grafica.
p1.setLayout(new BorderLayout());
p1.add(label1,BorderLayout.NORTH);
arvore = new JTree(pais);
// Listen for when the selection changes.
arvore.addTreeSelectionListener(this);

// Adicionando a arvore ao primeiro JPanel.

```

```

        p1.add(new JScrollPane(arvore),BorderLayout.CENTER);
        // Adicionando o texto de quem foi selecionado.
        p1.add(currentSelectionField,BorderLayout.SOUTH);
        // Adicionando botao que fornece o nome do estado ou
        // da regioa que sera incluido no JTabbedPane seguinte.
        botao1 = new JButton("Selecionar");
        // Adicao de evento no botao.
        botao1.addActionListener(this);
        p1.add(botao1,BorderLayout.EAST);
        p1.setBackground(Color.white);
        return p1;
    }

    private JPanel createJPanel2()
    {
        p2 = new JPanel();
        JLabel label2 = new JLabel("Tabela de Avalia  o");
        p2.setBackground(Color.LIGHT_GRAY);

        // Construindo a JTable.
        String[] colunas = new String []{"Regi  o/Estado","Confirmado","Pre  o"};

        //Object[][] dados = {{new String(""), new Boolean(true), new
        Double(500.0)}}};
        Object[][] dados = new Object[][]{};

        // Ao inves de passar direto, colocamos os dados em um modelo
        modelo = new DefaultTableModel(dados, colunas);
        // e passamos o modelo para criar a jtable
        jtable = new JTable( modelo );

        p2.setLayout(new BorderLayout());
        p2.add(label2,BorderLayout.NORTH);
        p2.add(jtable,BorderLayout.CENTER);
        return p2;
    }

    public void actionPerformed(ActionEvent e)
    {

```

```

        // Evento no qual o bot  o de sele   o de estado/regi  o foi
        pressionado.
        if (e.getSource() == botao1)
        {
            String nomeitem =
            arvore.getLastSelectedPathComponent().toString();

            modelo.insertRow(jtable.getRowCount(),new Object[]{nomeitem,new
            Boolean(true), new Double(120.0)}});

        }
        // Evento no qual o bot  o de sele   o de gerar relat  rio foi pressionado.
        if (e.getSource() == botao2)
        {

        }

    }

    public void valueChanged(TreeSelectionEvent e)
    {

        String nomeitem = arvore.getLastSelectedPathComponent().toString();
        currentSelectionField.setText("Sele   o atual:" +
        nomeitem);

    }

    public static void main( String args[] )
    {
        Exe4 jt1 = new Exe4();

    }

}

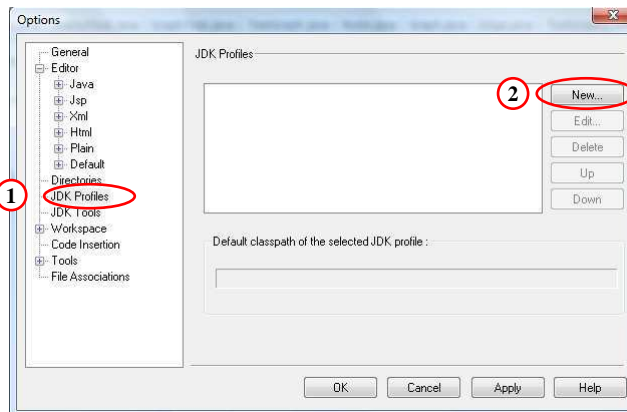
```

1. Configurando o JDK no JCreator

Passo 1: Selecione o menu **Configure** e depois a opção **Options...**



Passo 2: Na Janela que aparecer selecione **JDK Profiles** e depois aperte **New...**

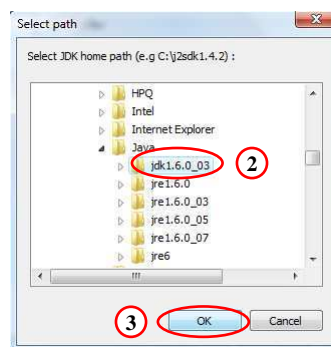
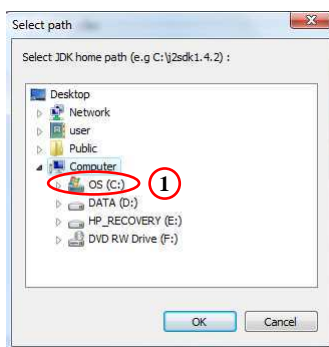


© Azevedo

1

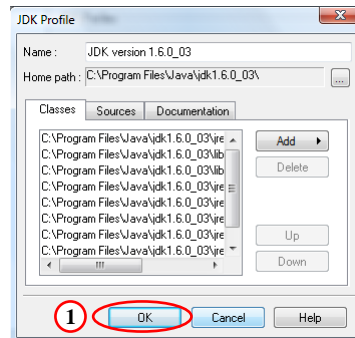
1. Configurando o JDK no JCreator

Passo 3: Selecione a pasta **C:\ArquivosdeProgramas\Java\jdk** e OK



1. Configurando o JDK no JCreator

Passo 4: Na nova janela clique OK novamente



1. Configurando o JDK no JCreator

Passo 5: Na nova janela clique OK novamente

