Um aplicativo para desenho Geométrico

Para criarmos um programa de desenho geométrico, precisamos criar classes que representem as diversas figuras geométricas, como ponto, reta, circulo, elipse, dentre outras.

O programa deverá ler um arquivo texto que contenha as descrições geométricas das figuras, armazenando-as em um vetor de figuras geométricas e, em seguida, permitirá desenhar novas figuras geométricas, gravá-las em um arquivo texto e, em outras execuções, recuperar os objetos gravados e refazer as figuras na tela.

As figuras geométricas serão descritas vetorialmente, ou seja, levando em conta as suas coordenadas e medidas no plano bidimensional, com parâmetros geométricos de posicionamento.

Na Orientação a Objetos, uma classe descreve um modelo de um objeto, ou seja, de uma estrutura de armazenamento de informações e comportamentos, que são usadas para processamento por um programa.

Toda classe é criada a partir de outras classes ancestrais, usadas como base de criação. Dessa maneira, temos uma genealogia de classes, em que uma classe ancestral é usada como base para a descrição de outras classes. Em Java, toda classe é descendente de uma classe ancestral chamada Object que é, portanto, a classe ancestral de todas as demais classes do Java.

A classe base (classe ancestral ou classe superior) do nosso programa de desenho de figuras geométricas será a classe Ponto, que informa um local no espaço bidimensional, ou seja, na tela do programa. Um ponto, ou local no espaço bidimensional, é representado por uma coordenada (x,y), e uma cor, que será a cor de desenho do ponto.

Portanto, entre no IntelliJ, crie um novo projeto, chamado Grafico, e crie uma nova classe, chama Ponto e a salve. No início do arquivo importe o pacote java.awt.*.

O código inicial da classe Ponto vem a seguir:

```
import java.awt.*;  // Abstract Windowing Toolkit

public class Ponto
{
  private int x,  y;
  private Color cor;
}
```

Observe, portanto, que acima descrevemos a estrutura de uma classe, com 3 componentes: x, y e cor. Essas informações identificam um ponto no plano cartesiano.

A tela do computador, onde serão desenhadas as figuras geométricas, também é um plano cartesiano.

Uma classe é, de certa maneira, semelhante a um registro de Delphi. No entanto, ela não armazena apenas campos, como um registro, mas também permite que se descrevam as **operações** que esse registro realiza. Ou seja, um objeto, tanto em Delphi quanto em Java, é criado a partir de uma classe, e executa operações descritas nessa classe.

A primeira operação que um objeto deve realizar é, na verdade, vir a existir, ou seja, ser criado na memória do computador. A criação de um objeto é feita através de um procedimento especial, chamado construtor. O construtor de uma classe descreve as operações que são realizadas com os campos da classe quando um objeto dessa classe passa a ser usado.

Quando criamos um objeto da classe Ponto, precisamos colocar os valores de x, y e cor nos campos desse objeto. Assim, abaixo acrescentamos o construtor na classe Ponto:

```
public class Ponto
{
  private int x,  y;
  private Color cor;
```

```
        public Ponto(int cX, int cY, Color qualCor) {
            x = cX;
            y = cy;
            cor = Color.qualCor;
        }
    }
```

cX e cY são as coordenadas cartesianas x e y do ponto que estamos criando, e qualCor é a cor desse objeto. Num programa que use a classe Ponto, para criarmos um ponto, faríamos algo semelhante a:

```
  Ponto p;
…
  P = new Ponto(3, 2, Color.red);
…
```

O primeiro comando acima declara uma variável p, da classe Ponto, mas para criá-la e colocar valores dentro dela, precisamos executar o construtor da classe Ponto. A execução do construtor é feita com new Ponto(). Os parâmetros 3, 2 e Color.red indicam a coordenada x, a coordenada y e a cor desejados para o ponto.

A partir da execução dos comandos acima, um objeto p, baseado na classe Ponto, passa a existir na memória.

Também é importante criarmos métodos de acesso aos atributos da classe, usando o conceito de getters e setters que aprendemos em Paradigmas de Programação. Assim, podemos controlar o acesso aos valores dos atributos quando necessário e impedir que o programa de aplicação possa deturpar esses valores através de atribuição de valores inválidos. Abaixo temos os getters e setters da classe Ponto:

```
        public void setX(int novoX) {
            x = novoX;
        }

        public void setY(int novoY) {
            y = novoY;
        }

        public void setCor(Color novaCor)
        {
            cor = novaCor;
        }

        public int getX() {
            return x;
        }

        public int getY() {
            return y;
        }

        public Color getCor()
        {
            return cor;
        }
```

Uma outra coisa que um objeto gráfico, como o Ponto, deve realizar, é desenhar-se. Em outras palavras, precisamos criar um procedimento da classe Ponto que desenho o ponto no local indicado pelas coordenadas cartesianas do ponto, e pintado na cor indicada pelo atributo cor.

```
        public void desenhar(Color cor, Graphics g) {
            g.setColor(cor);
            g.drawLine(getX(),getY(),getX(),getY());
        }
```

O desenho de figuras geométricas em Java é feito sobre uma área de desenho, chamada contexto gráfico e que é representada, no método acima, pelo parâmetro g, da classe Graphics. Todo objeto visível do Java possui esse contexto gráfico, que é um plano cartesiano, com coordenadas (x,y), sobre a qual o objeto é desenho. Podemos desenhar um ponto sobre qualquer objeto visível de um contexto gráfico, inclusive sobre botões, caixas de texto, etc. No entanto, em nossa aplicação desejamos fazer desenhos em uma região da tela do programa preparada para desenho. Um bom componente para realizar desenhos é o JPanel, e isso será explicado em seguida.

No método desenhar() acima, usamos g.drawLine para desenhar uma linha reta entre dois ppntos cartesianos. Java não possui um método para desenho de pontos, assim, para desenharmos um único ponto é preciso desenhar uma linha reta que comece e termine na mesma coordenada. getX() e getY() são os valores dos campos X e Y do TPonto que estamos desenhando.

Note que uma cor é passada como parâmetro para esse método. Se a cor passada como parâmetro ao método desenhar(), for a mesma cor de fundo que a usada no contexto gráfico onde se desenham as figuras, o ponto será apagado. Se a cor for diferente, o ponto será visível. Por exemplo, se o contexto gráfico de desenho tiver cor de fundo branca (definida pela constante Color.white) e o parâmetro cor tiver valor Color.black, aparecerá um ponto preto sobre o fundo branco. Por outro lado, se logo em seguida invocarmos o método com o mesmo contexto gráfico e o parâmetro cor valer Color.white, será desenhado um ponto branco que, por ser da mesma cor que o fundo da área de desenho, fará com que o ponto anterior deixe de ser visível.

Derivação de Classes para novas figuras geométricas

Como estamos usando o Paradigma de Orientação a Objetos, podemos usar os mecanismos de herança e polimorfismo para nos auxiliar na definição das figuras geométricas que nosso editor gráfico desenhará.

Uma linha reta, por exemplo, pode ser definida como a figura geométrica que exibe pontos colineares entre um ponto inicial e um ponto final. Ou seja, uma linha reta é uma figura geométrica que tem início em um ponto base inicial e se estende até um ponto final.

Portanto, a classe Linha pode ser descrita a partir da classe Ponto. Em outras palavras, Linha é uma herança de Ponto, como podemos ver na definição de classe abaixo:

```
import java.awt.*;  // para acessar Color e métodos de desenho

public class Linha extends Ponto {

  // herda (x, y) da classe Ponto, que são as coordenadas
  // do ponto inicial da reta; também herda a cor e, em
  // seguida define o ponto final:

      private Ponto pontoFinal;
}
```

Acima, o ponto inicial é o objeto ancestral que foi herdado da classe Ponto. Portanto, todo objeto da classe Linha possui os atributos x, y e cor, o construtor de Ponto e o método desenha() que, se chamado a partir de um objeto da classe Linha, desenharia apenas o ponto inicial da linha reta.

Além disso, uma linha reta precisa ter um ponto final (pois podemos desenhar na tela apenas segmentos de reta, e não retas "infinitas"). A linha também possui uma cor usada para desenhá-la. Assim, poderemos ter cores distintas em linhas distintas. Para desenhar uma linha reta, usaremos um método de Java que desenha linhas retas entre um ponto inicial e um ponto final.

O método construtor abaixo recebe como parâmetros os componentes x1, y1 e x2, y2 das coordenadas dos pontos inicial e final da reta sendo criada, bem como sua cor e cria um objeto Linha, usando, para isso, o construtor **herdado** da classe Ponto (**super()**) e, em seguida, instanciando o ponto final dessa linha reta que estamos criando.

```
public Linha(int x1, int y1, int x2, int y2, Color novaCor)
{
  super(x1,y1, novaCor);
  pontoFinal = new Ponto(x2,y2, novaCor);
}
```

Já o método desenhar() de Linha, como explicado acima, sobrepõe-se (**overrides**) ao mesmo método da classe ancestral Ponto e efetua o desenho de uma linha reta entre (x1,y1) e (x2,y2), usando o método drawLine() do contexto gráfico **g**, passado como parâmetro:

```
public void desenha(Color corDesenho, Graphics g)
{
      g.setColor(corDesenho);
      g.drawLine(super.getX(),super.getY(),   // ponto inicial
                  pontoFinal.getX(), pontoFinal.getY());
}
```

Crie os getters e setters adequados para a classe Linha. Assim sendo, temos pronta a primeira versão das classes Ponto e Linha, que definem e desenham, respectivamente, pontos e segmentos de retas num plano cartesiano gráfico (contexto gráfico g).

Além dessa classe, temos abaixo a descrição da classe Circulo:

```
import java.awt.*;

public class Circulo extends Ponto {

      // herda o ponto central (x, y) da classe Ponto

      int raio;
      Color cor;

      public Circulo(int xCentro, int yCentro, int novoRaio, Color novaCor)
      {
            super(xCentro, yCentro, novaCor);  // construtor de Ponto(x,y)
            setRaio(novoRaio);
      }

      public void setRaio(int novoRaio) {
            raio = novoRaio;
      }

      public void setCor(Color novaCor) {
            cor = novaCor;
      }

      public void desenha(Color corDesenho, Graphics g) {
            g.setColor(corDesenho);
            g.drawOval(getX()-raio, getY()-raio,      // centro - raio
                        2*raio,2*raio);               // centro + raio
      }
}
```

A seguir temos a classe Oval, que descreve elipses:

```
import java.awt.*;
```

```
public class Oval extends Ponto {
      int raioA,
          raioB;

      public void desenha(Color corDesenho, Graphics g) {
            g.setColor(corDesenho);
            g.drawOval(getX()-raioA, getY()-raioB,  // centro - raio
                        2*raioA,2*raioB);  // centro + raio

      }

      public Oval()
      {
            super();
            setRaioA(0);
            setRaioB(0);
            setCor(Color.black);
      }

      public void setRaioA(int novoRaio) {
            raioA = novoRaio;
      }

      public void setRaioB(int novoRaio) {
            raioB = novoRaio;
      }

      public Oval(int xCentro, int yCentro, int novoRaioA,
                  int novoRaioB, Color novaCor)
      {
            super(xCentro, yCentro, novaCor);  // construtor de Ponto(x,y)
            setRaioA(novoRaioA);
            setRaioB(novoRaioB);
      }
}
```

Pacote Swing – Introdução

Swing é uma melhoria do pacote AWT – Abstract Windowing Toolkit – que vem com Java. Awt era o único pacote disponível para criar interfaces gráficas com o usuário (GUI). No entanto, um aplicativo que usa AWT muda de aparência e de comportamento dependendo do ambiente em que é executado, de forma que um aplicativo Java executado em Windows teria aparência diferente do mesmo aplicativo executado em Linux, além da possibilidade de os controles visuais terem comportamento diferente nos sistemas.

Isso acontece porque AWT é feito baseando-se no código nativo de cada sistema operacional.

Já o Swing é feito em Java, mantém a aparência e o comportamento dos controles, em qualquer ambiente ou sistema operacional.

JFrame

Para criar formulários em Java, como os que usamos em Delphi ou Visual Basic (Windows Forms) devemos criar uma classe que seja herança da classe JFrame. JFrame é um container para componentes visuais de interface com o usuário.

JFrame está disponível no pacote javax.swing, que deve ser importado para podermos usar JFrame.

Cria-se um aplicativo Java (classe com public void main()) e importam-se os pacotes abaixo:

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

public class Editor extends JFrame
{
      public static void main(String[] args) {
      }
}
```

Vamos criar o formulário, através do construtor da classe Editor, que chamará o método construtor da classe superior (super()). Esse método construtor do JFrame pode receber um parâmetro que é o título da janela. Também devemos executar o método setSize para estabelecer o tamanho da janela e o método show() para exibir a janela.

```
public class Editor extends JFrame
{
  public Editor() // construtor de Editor que criará o JFrame, colocará seu
  {               // título, estabelecerá um tamanho para o formulário e o
                  // exibirá
      super("Editor Gráfico");      // cria o JFrame e coloca um título
      setSize(700,500);             // dimensões do formulário em pixels
      show();                       // exibe o formulário
  }
```

No método main(), devemos criar um objeto que represente a aplicação; esse objeto é da classe que estamos criando (no caso, Editor):

```
      public static void main(String[] args) {
            Editor aplicacao = new Editor();
      }
```

Isso cria o objeto e os métodos do construtor são executados, de forma que o formulário aparecerá, como na figura abaixo:

Agora, declararemos os controles visuais (Botões, rótulos e caixas de edição).

```
public class Editor extends JFrame
{
  private JButton btnPonto, btnLinha, btnCirculo, btnElipse, btnCor, btnAbrir,
                  btnSalvar,btnApaga, btnSair;
```

Esses controles terão de ser instanciados (criados), e associados a um container. Esse container pode ser um JPanel, que é um controle que armazena outros controles, como o TPanel de Delphi. Declaramos o JPanel com o comando abaixo:

```
private JPanel pnlBotoes;     // como o TPanel do Delphi
```

Entre a chamada do construtor super e a configuração do tamanho do JFrame, devemos instanciar (criar) os JButtons e o JPanel.
Os botões terão ícones. Para isso, criamos um objeto da classe Icon, usando o seu construtor que recebe o nome de um arquivo de imagem como parâmetro. Em seguida, criamos cada botão e passamos o ícone como parâmetro, além do título que desejamos para o botão:

```
super("Editor Gráfico");      // cria o JFrame e coloca um título

// cria os botões do editor

Icon imgAbrir = new ImageIcon("abrir.bmp");
btnAbrir = new JButton("Abrir", imgAbrir);
btnSalvar = new JButton("Salvar", new ImageIcon("salvar.bmp"));
btnPonto = new JButton("Ponto", new ImageIcon("ponto.bmp"));
btnLinha = new JButton("Linha", new ImageIcon("linha.bmp"));
btnCirculo = new JButton("Circulo", new ImageIcon("circulo.bmp"));
btnElipse = new JButton("Elipse", new ImageIcon("elipse.bmp"));
btnCor = new JButton("Cores", new ImageIcon("cores.bmp"));
btnApagar = new JButton("Apagar", new ImageIcon("apagar.bmp"));
btnSair = new JButton("Sair", new ImageIcon("sair.bmp"));
```

Para exibir os botões, eles devem estar armazenados em um container, como dito acima. Esse container será o JPanel pnlBotoes. No entanto, todo container precisa ter uma disposição física pré-estabelecida para os seus componentes. Essa disposição física estabelece como os componentes são dispostos e é chamada de layout. Existem 3 tipos de layout : FlowLayout, GridLayout e BorderLayout.

FlowLayout é um layout livre, onde os componentes vão sendo dispostos da esquerda para a direita, de cima para baixo, e se adaptam ao tamanho e formato da tela conforme ele muda.

GridLayout define uma matriz (número de linhas x número de colunas) onde os componentes são armazenados, da esquerda para a direita, de cima para baixo, dentro das células dessa matriz. Os componentes, portanto, são dispostos em formato tabular.

BorderLayout utiliza o formulário como uma mapa, definindo posições superior (NORTH), inferior (SOUTH), esquerda (WEST), direita (EAST), central (CENTER) onde podemos colocar os componentes nas *fronteiras* do mapa.

No caso do editor gráfico, usaremos BorderLayout como o formato para colocar o painel de botões e a área onde, posteriormente, faremos os desenhos. O container dos botões, pnlBotoes, ficará na posição NORTH, enquanto a área de desenho, no pnlDesenho, ficará na posição SOUTH.

Dentro do pnlBotoes, usaremos o layout FlowLayout, para colocar os botões no formato livre. Criamos o JPanel pnlBotoes e, em seguida, um formato de FlowLayout flwBotoes. Associamos o JPanel com esse formato, de modo que quando colocarmos componentes no Panel, eles sejam colocados nas células da matriz cujo formato foi definido pelo gridBotoes:

```
// cria o JPanel que armazenará os botões
pnlBotoes = new JPanel();
// cria o layout usado para dispor fisicamente os botões
FlowLayout flwBotoes = new FlowLayout();
// informa que os componentes do pnlBotoes serão dispostos em forma livre
pnlBotoes.setLayout(flwBotoes);
```

Agora, devemos colocar os botões no pnlBotoes, usando seu método add():

```
// adiciona os controles visuais (botões) ao painel de botões, de cima
// para baixo, da esquerda para direita.
pnlBotoes.add(btnAbrir);
pnlBotoes.add(btnSalvar);
pnlBotoes.add(btnPonto);
pnlBotoes.add(btnLinha);
pnlBotoes.add(btnCirculo);
pnlBotoes.add(btnElipse);
pnlBotoes.add(btnCores);
pnlBotoes.add(btnApagar);
pnlBotoes.add(btnSair);
```

Agora, estabelecemos as dimensões do Formulário e o exibimos, com os dois comandos abaixo:

```
setSize(700,500);       // dimensões do formulário em pixels
setVisible(true);       // exibe o formulário
```

Se executarmos o aplicativo agora, apenas o formulário aparecerá, sem os botões que criamos.

Para que todos os botões apareçam, o painel que os contêm (pnlBotoes) precisa ser colocado dentro do Frame. Para isso, é usado um objeto da classe Container, que armazena controles visuais e conterá os objetos que serão exibidos no Frame. O painel de botões será colocado na parte superior do formulário, portanto usaremos no Container o layout BorderLayout:

```
Container cntForm = getContentPane(); // acessa o painel de conteúdo do frame
cntForm.setLayout(new BorderLayout());
cntForm.add(pnlBotoes , BorderLayout.NORTH);
```

As figuras geométricas que nosso editor desenhará serão traçadas numa janela auxiliar, no modelo MDI (Multiple Document Interface). Em princípio, quando o programa começa a ser executado, haverá uma janela sem desenho algum, e não associada a nenhum arquivo de figuras geométricas (o tipo de arquivo que criaremos com o programa).

Para um programa gerenciar janelas-filhas no modelo MDI, java utiliza a classe JDesktopPane. Declaramos uma variável dessa classe, chamada panDesenho, a instanciamos dessa classe e a associamos ao formulário com os dois comandos abaixo:

```
JDesktopPane panDesenho = new JDesktopPane();
cntForm.add(panDesenho);
```

Após criarmos o JDesktopPane, criaremos uma janela-filha vazia. Essa janela-filha deve ser declarada como uma variável global da classe Editor, no início do texto, após a declaração do JPanel pnlBotoes, com o comando abaixo:

```
static private JInternalFrame frame;
```

Voltando ao código do construtor, instanciamos a variável frame e associamos, após essa instanciação, a janela-filha chamada *frame com o* JDesktopPane que a gerenciará:

```
frame = new JInternalFrame("Nenhum arquivo aberto", true, true, true, true);
panDesenho.add(frame);
```

Os parâmetros do construtor de JInternalFrame que usamos acima significam:

String – título da janela
Boolean – janela-filha pode ser redimensionada pelo usuário?
Boolean – janela-filha pode ser fechada pelo usuário?
Boolean – janela-filha pode ser maximizada pelo usuário?
Boolean – janela-filha pode ser minimizada pelo usuário?

Após isso, devemos dimensionar a janela-filha e exibi-la, com os dois comandos abaixo:

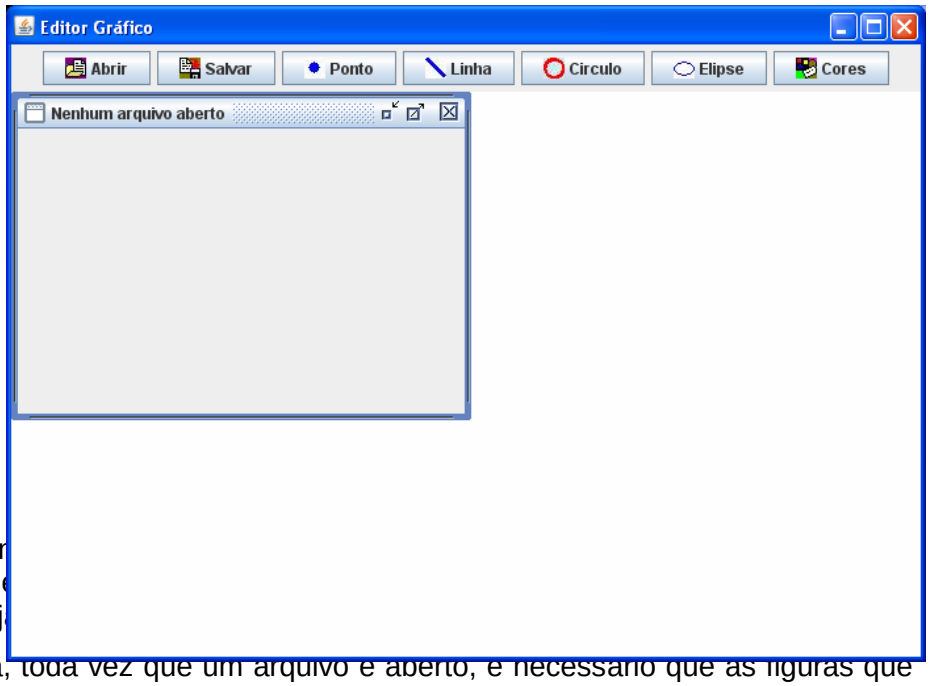
```
frame.setSize(this.getWidth() / 2,this.getHeight() / 2);
frame.show();
```

Observe que definimos, no método setSize() da janela-filha (frame), para metade da largura e metade da altura do formulário (como estamos no construtor da classe Editor, e esta estende JFrame (é uma herança de JFrame), this indica justamente o objeto JFrame que estamos construindo no momento).

Quando uma janela-filha é criada, o padrão é que ela seja transparente, ou seja, quando ela é maximizada, outras janelas e ícones do JDesktopPane são visíveis através do fundo da janela-filha maximizada. Para impedir esse efeito, devemos executar o método setOpaque com parâmetro true:

```
frame.setOpaque(true);
```

Executando o programa, teremos o resultado da figura ao lado:



Agora, devemos criar a área de desenho das figuras geométricas, dentro da janela-filha *frame*. Essa área será um painel.

Todo componente visual de java possui um evento paintComponent(), que desenha o conteúdo do componente. paintComponen
que redesenhar um compon
outra e depois é retirada. A j

No caso do nosso programa, toda vez que um arquivo é aberto, é necessário que as figuras que nele estão armazenadas sejam desenhadas no painel de desenho. Quando se muda o tamanho da janela-filha (maximiza, minimiza, desloca, etc), as figuras devem ser também redesenhadas.

Mas o evento paintComponent() padrão do java não sabe buscar os dados sobre as figuras geométricas e novamente desenhá-las. Para tanto, teremos de criar um evento paintComponent() específico do painel de desenho, que busque os objetos gráficos numa **lista de figuras** e os desenhe. Para sobrepormos o evento padrão paintComponent() com um evento específico, teremos que criar uma **classe interna** à classe Editor, que estenderá JPanel. Nessa classe interna, que chamaremos de MeuJPanel, faremos a codificação de um evento paintComponent() que busca os dados das figuras geométricas na lista de figuras e as desenha no painel.

Ainda não preenchemos essa lista. Isso será feito quando abrirmos um arquivo de figuras geométricas (o que aprenderemos logo mais), ou quando nosso programa for usado para efetuar desenhos (com os botões que criamos acima).

Portanto, para adiantarmos nosso trabalho, retorne ao início da classe, na parte de declaração de variáveis e declare o MeuJPanel pnlDesenho, um vetor simples para armazenar objetos da classe base (Ponto) e o instancie. Isso é feito pelos comandos abaixo:

```
static private MeuJPanel pnlDesenho;
private static Ponto[] figuras = new Ponto[tamanho inicial];
```

A classe interna MeuJPanel deve ser declarada logo depois do void main(), antes do } da classe Editor:

```
private class MeuJPanel extends JPanel
{
	public void paintComponent(Graphics g)
	{
		for (int atual = 0; atual < quantidade de figuras no vetor; atual++)
		{
			Ponto figuraAtual = (Ponto) figuras[atual];
			figuraAtual.desenhar(figuraAtual.getCor(), g);
		}
	}
}
```

Observe o uso do método desenhar() de cada elemento recuperado do vetor. Cada figura é obtida da posição do vetor atualmente acessada e tratado como um objeto da classe ancestral Ponto, que funciona como uma classe unificadora de todas as classes dela derivadas.

Como todas as figuras geométricas que derivamos na seção anterior são heranças (extensões) de Ponto, e todas elas possuem um método desenhar() próprio, o método que será chamado será aquele da classe da figura geométrica específica\. Em outras palavras, se a figura for uma linha reta, será chamado o método desenhar() da classe Linha, e se a figura for uma instância de círculo, será chamado o método desenhar() de Circulo.

Logo após a chamada de frame.setOpaque(true), devemos instanciar o painel de desenhos, criar um container dentro da janela-filha frame, e associar o painel a esse container. Isso é feito pelos comandos abaixo:

```
		pnlDesenho = new MeuJPanel();
		Container cntFrame = frame.getContentPane();
		cntFrame.add(pnlDesenho);
```

Execute seu programa para testar seu funcionamento.

Tratamento de Eventos

Para tratarmos eventos, temos que criar uma classe “Ouvinte” para cada controle visual que terá eventos. Essa classe fica continuamente testando os componentes para verificar se ocorreu uma interação com o usuário, ou seja, um evento. Cada “ouvinte” deverá também ter associado um “Tratador” (handler) do evento, que realiza as operações referentes ao evento que foi “ouvido”.

O primeiro evento que trataremos é o fechamento da aplicação, após termos criado o objeto que representa a classe Editor (variável aplicação):

```
aplicacao.addWindowListener ( ... );  // cria um ouvinte para janelas e o
                                      // adiciona para o objeto aplicação
```

O método addWindowListener usa como parâmetro uma instância da classe WindowAdapter, que é uma interface que manipula eventos de janelas. Como é uma interface, temos que implementar os métodos dessa interface, com o código abaixo:

```
Editor aplicacao = new Editor();
aplicacao.addWindowListener (
	new WindowAdapter () {   //  cria instância da interface
		public void windowClosing(WindowEvent e){
			System.exit(0);
		}
	}
);
```

Para tratarmos os demais eventos, temos que **declarar** uma classe interna ouvinte que implemente a interface ActionListener para cada evento que queiramos tratar. Essa interface estabelece o método actionPerformed que deve ser implementado (pois é um método dessa interface), e que é o tratador do evento, ou seja, ele contém o código que será executado quando o evento for disparado.

O botão btnAbrir exibe um diálogo de abertura de arquivo, que permitirá selecionar um arquivo com as figuras geométricas salvas anteriormente pelo aplicativo. Quando esse botão for pressionado, devemos exibir a janela de diálogo de abertura de arquivo e obter o nome do arquivo selecionado. Essa janela é exibida a partir da criação de uma instância da classe JFileChooser. Caso o usuário escolha um arquivo, devemos recuperar o nome desse arquivo e lê-lo, armazenando as figuras geométricas num vetor de figuras geométricas, conforme elas forem lidas.

Portanto, essa seqüência de operações forma o código de tratamento do evento disparado pelo clique desse botão. Temos que criar uma classe ouvinte para fazer essa operação:

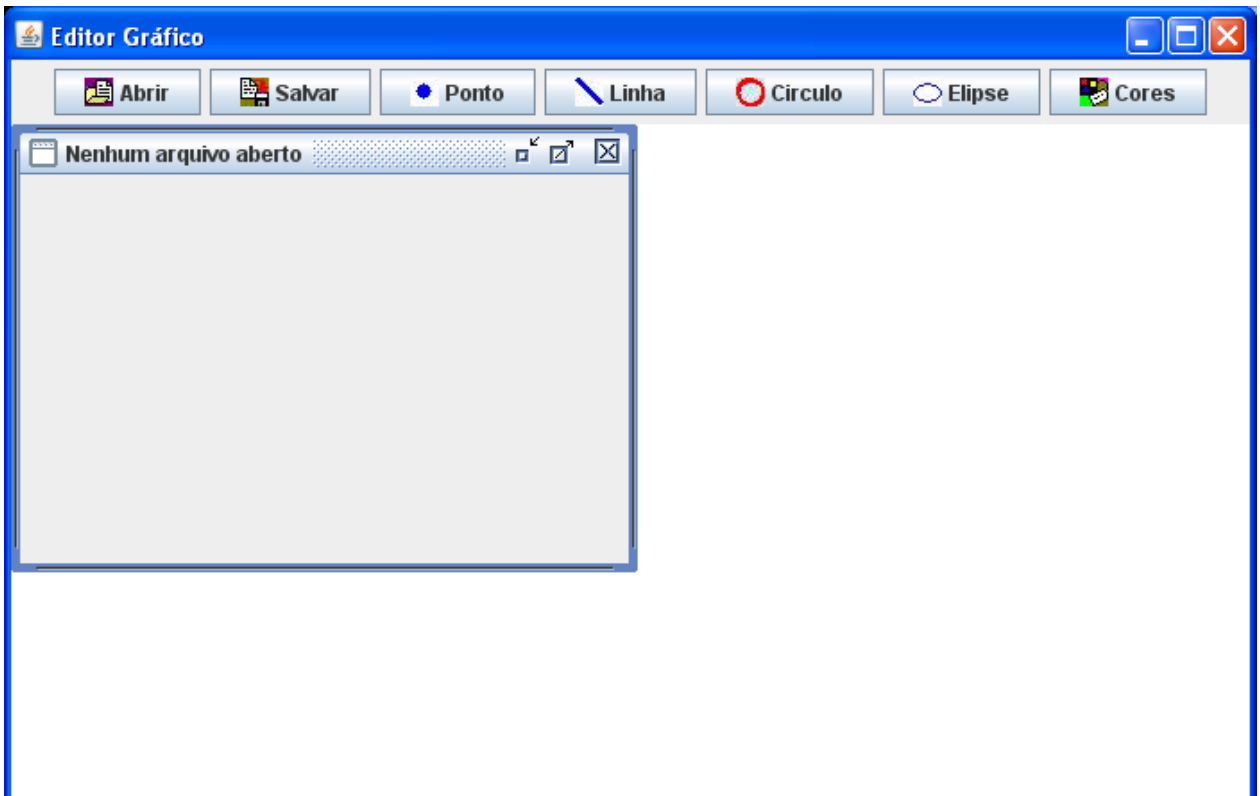
```
private class FazAbertura implements ActionListener {
   public void actionPerformed(ActionEvent e)   // código executado no evento
  {
      JFileChooser arqEscolhido = new JFileChooser ();
      arqEscolhido.setFileSelectionMode(JFileChooser.FILES_ONLY);

      int result = arqEscolhido.showOpenDialog(Editor.this);
//… código de verificação se um arquivo foi selecionado e obtenção de seu nome
  }
}
```

O código acima deve ser digitado após o fecha chave do void main(), ainda dentro da classe Editor.

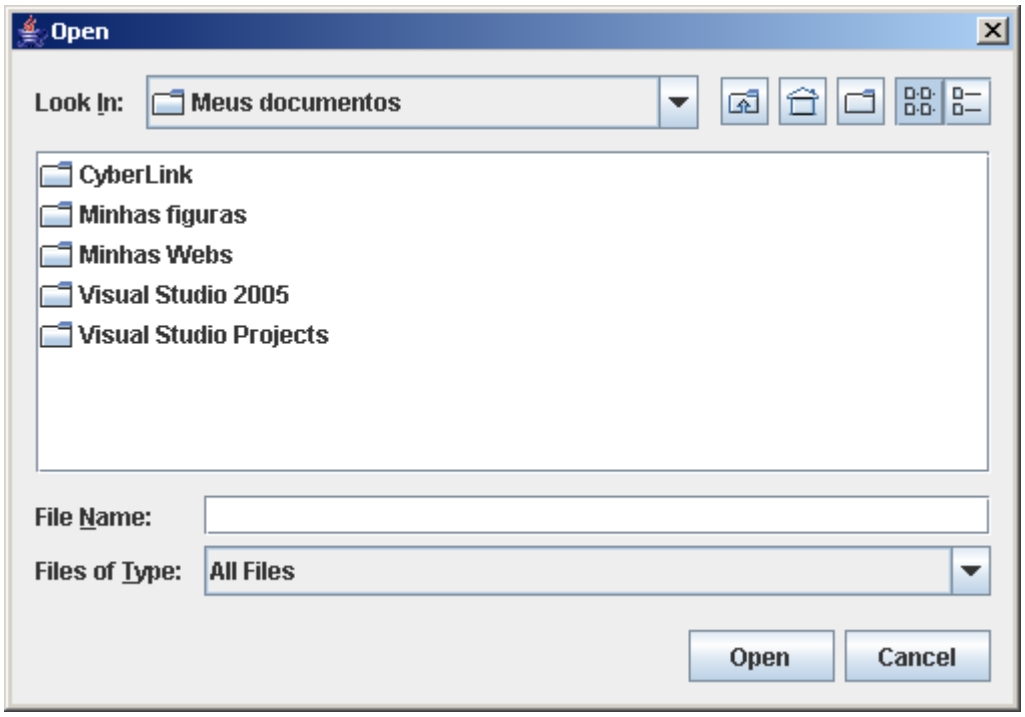
Mas esse código ainda não está associado ao btnAbrir. Isso deve ser feito no construtor da classe Editor, após termos criado esse botão. Mas como esse código é uma descrição de classe, para associá-lo ao botão, é preciso instanciar a classe, o que é feito dentro do construtor, e antes de se chamar o método show().

```
      btnAbrir.addActionListener(new FazAbertura());
```

Ao digitarmos essas linhas de código e executarmos o aplicativo, teremos a janela da figura abaixo na tela.



Pressionando o botão [Abrir], o evento associado a ele será disparado, e a caixa de abertura abaixo será exibida:



Para sabermos se um arquivo foi selecionado, devemos verificar o resultado da chamada ao método que exibe a janela. Esse resultado foi armazenado na variável result. Existe uma constante pré-definida na classe JFileChooser, que informa que o resultado foi a seleção de um nome de arquivo e não o pressionamento do botão [Cancel]. Essa constante é APPROVE_OPTION. Portanto, se result é igual a JFileChooser.APPROVE_OPTION, podemos abrir o arquivo selecionado.

A obtenção do nome do arquivo selecionado, é feita pelo método getSelectedFile() do FileChooser. Esses passos são demonstrados no código a seguir (no quadro):

```
   public void actionPerformed(ActionEvent e)   // código executado no evento
   {
      JFileChooser arqEscolhido = new JFileChooser ();
      arqEscolhido.setFileSelectionMode(JFileChooser.FILES_ONLY);
      int result = arqEscolhido.showOpenDialog(Editor.this);
```

```
//código de verificação se um arquivo foi selecionado e obter seu nome
      if (result == JFileChooser.APPROVE_OPTION)
      {
         File arquivo = arqEscolhido.getSelectedFile();
         System.out.println("Processando "+arquivo.getName());
      }
   }
```

Pode-se, após isso, declarar-se um BufferedReader para ler os dados do arquivo selecionado, usando arquivo.getName() como argumento do objeto FileReader usado para configurar o BufferedReader. Isso é feito abaixo, em código que seguiria o System.out.println do quadro acima:

```
try {
  BufferedReader arqFiguras = new BufferedReader(
                                        new FileReader(arquivo.getName()));
}
catch (FileNotFoundException e)     {
     System.out.println("Arquivo não pôde ser aberto");
}
```

A abertura de um FileReader deve ser feita dentro de um bloco try-catch, para capturer possíveis erros de arquivo não encontrado (tratadas pela classe FileNotFoundException).

Como se vê no comentário logo após o catch { }, depois que abrirmos o arquivo, devemos processá-lo, lendo cada linha e interpretando o seu conteúdo de acordo com o formato de definição de cada figura geométrica no arquivo, com as características abaixo:

Tipo Figura	Coord. Base		Componentes de cor			Informações específicas	
p	x	y	corR	corG	corB		
l	x1	y1	corR	corG	corB	x2	y2
c	xc	yc	corR	corG	corB	raio	
o	xc	yc	corR	corG	corB	raioA	raioB
colunas 0-5	5-10	10-15	15-20	20-25	25-30	30-35	35-40

Cada campo do arquivo ocupará 5 posições, para facilitar o processamento de cada campo.
Por exemplo, as linhas arquivo abaixo,que poderiam ser lidas de um arquivo de figuras, representam:

```
          1         2         3         4
01234567890123456789012345678901234567890  → "régua" não está no arquivo
o      500  300  192  192  192  200  150
c      200  500  255    0    0  340
p      110  120    0  255    0
```

o – oval, com centro em (500,300), raioA =200, raioB = 150, cor cinza (R=192,G=192,B=192)
c – círculo, com centro em (200,500), raio = 340, cor vermelha (R=255,G= 0,B= 0)
p – ponto, com coordenada (110,120), cor verde (R= 0,G=255,B= 0)

Essas linhas seriam lidas e, de acordo com o primeiro caracter (‘p’,’l’,’c’ ou ’e’), que é o tipo de figura, recuperaríamos os demais campos. Em outras palavras, se o tipo da figura é ‘l’, devemos recuperar x1,y1, corR, corG e corB, que são comuns a todas as figuras geométricas. Em seguida, recuperamos os valores que são específicos de uma linha reta (a coordenada do 2º extremo da reta, dada por (x2,y2)).

Para recuperar partes de uma string, usamos a função substring, como estudamos nas classes Pessoa e Aluno.

Cada figura geométrica que lermos do arquivo deverá ser armazenada na próxima posição não usada de um vetor de figuras geométricas. Cada posição do vetor, portanto, armazenará uma instância de uma figura geométrica. Ao final da leitura, teremos no vetor a representação, por objetos, de todas as figuras cujas propriedades foram gravadas no arquivo e, em seguida, exibimos essas figuras, através dos métodos desenhar() de cada objeto gráfico.

Ao final da execução do aplicativo, todas as figuras que tenham sido desenhadas e armazenadas no vetor, inclusive as que lemos no início da execução, serão gravadas no arquivo novamente. Para isso, percorre-se o vetor de figuras geométricas e executa-se o método toString() de cada objeto gráfico. O resultado de toString() será gravado no arquivo. Obviamente, toString() deverá gerar uma string com o formato das linhas lidas do arquivo, do exemplo acima, para cada tipo de figura armazenada.

Assim sendo, o método toString() da classe Ponto, por exemplo, deveria ser semelhante a:

```
public class Ponto {
...
  public String transformaString(int valor, int quantasPosicoes)   {
    String cadeia = new String(valor+"");
    while (cadeia.length() < quantasPosicoes)
        cadeia = "0"+cadeia;
    return cadeia.substring(0,quantasPosicoes); // corta, se necessário, para
                                                // tamanho máximo
  }
```

```
  public String transformaString(String valor, int quantasPosicoes)
  {
    String cadeia = new String(valor+"");
    while (cadeia.length() < quantasPosicoes)
       cadeia = cadeia+" ";
    return cadeia.substring(0,quantasPosicoes); // corta, se necessário, para
                                                // tamanho máximo
  }

  public String toString()
  {
      return      transformaString("p",5)+
                  transformaString(getX(),5)+
                  transformaString(getY(),5)+
                  transformaString(getCor().getRed(),5)+
                  transformaString(getCor().getGreen(),5)+
                  transformaString(getCor().getBlue(),5);
  }
```

getRed(), getGreen() e getBlue() são funções da classe Color que devolvem os componentes R, G e B (Red, Green e Blue) de um objeto Color, como é o caso da cor de cada objeto de figura geométrica, cor essa acessada pelo método getCor().

Crie os métodos toString() de cada um dos objetos gráficos derivados de Ponto (Linha, Circulo, Oval), retornando uma string com os valores de definição geométrica do objeto, como fizemos para a classe acima. Lembre-se que, por serem heranças de Ponto, todos esses objetos possuem o método transformaString() com as diversas versões polimórficas.

Logo após abrirmos o arquivo, devemos ler suas linhas e capturar as informações armazenadas nelas. O trecho abaixo, que deve ser colocado logo após a abertura do arquivo e antes do } que termina a parte try (trechos que estão dentro dos quadros), lê linhas do arquivo até que este acabe, processa cada uma, determinando o tipo de figura geométrica que a string lida representa e captura os valores adicionais que cada tipo de figura possui, de acordo com o tipo da figura:

```
try {
   BufferedReader arqFiguras = new BufferedReader(
                                        new FileReader(arquivo.getName()));
try {
  String linha = arqFiguras.readLine();
  while (linha != null)
  {
     String tipo = linha.substring(0,5).trim();
     int xBase = Integer.parseInt(linha.substring(5,10).trim());
     int yBase = Integer.parseInt(linha.substring(10,15).trim());
     int corR  = Integer.parseInt(linha.substring(15,20).trim());
     int corG  = Integer.parseInt(linha.substring(20,25).trim());
     int corB  = Integer.parseInt(linha.substring(25,30).trim());
     Color cor = new Color(corR, corG, corB);
     switch (tipo.charAt(0))
     {
     case 'p' : // figura é um ponto
      figuras.insereAposFim(new ListaSimples.NoLista(
                                    new Ponto(xBase,yBase, cor), null);
      break;
     case 'l' : // figura é uma linha
      int xFinal =Integer.parseInt(linha.substring(30,35).trim());
      int yFinal =Integer.parseInt(linha.substring(35,40).trim());
      figuras.insereAposFim(new ListaSimples.NoLista(
                         new Linha(xBase, yBase, xFinal, yFinal, cor), null);
     break;
     case 'c' : // figura é um círculo
      int raio =Integer.parseInt(linha.substring(30,35).trim());
      figuras.insereAposFim(new ListaSimples.NoLista(
```

```
                    new Circulo(xBase, yBase, raio, cor), null);
        break;
        case 'o' : // figura é uma oval
         int raioA =Integer.parseInt(linha.substring(30,35).trim());
         int raioB =Integer.parseInt(linha.substring(35,40).trim());
         figuras.insereAposFim(new ListaSimples.NoLista(
                                new Oval(xBase, yBase, raioA, raioB, cor), null);
        break;
        }
        linha = arqFiguras.readLine();
    }
    arqFiguras.close();

    frame.setTitle(arquivo.getName());
    desenharObjetos(pnlDesenho.getGraphics());
}
   catch (IOException ioe){
       System.out.println("Erro de leitura no arquivo");
   }
}
catch (FileNotFoundException e)     {
      System.out.println("Arquivo não pôde ser aberto");
}
```

Os comandos

```
    frame.setTitle(arquivo.getName());
    desenharObjetos(pnlDesenho.getGraphics());
```

mudam o título da janela-filha, para o nome do arquivo aberto, e chama desenhaObjetos, passando a área gráfica de desenho (Contexto Gráfico) do painel de desenhos como parâmetro.

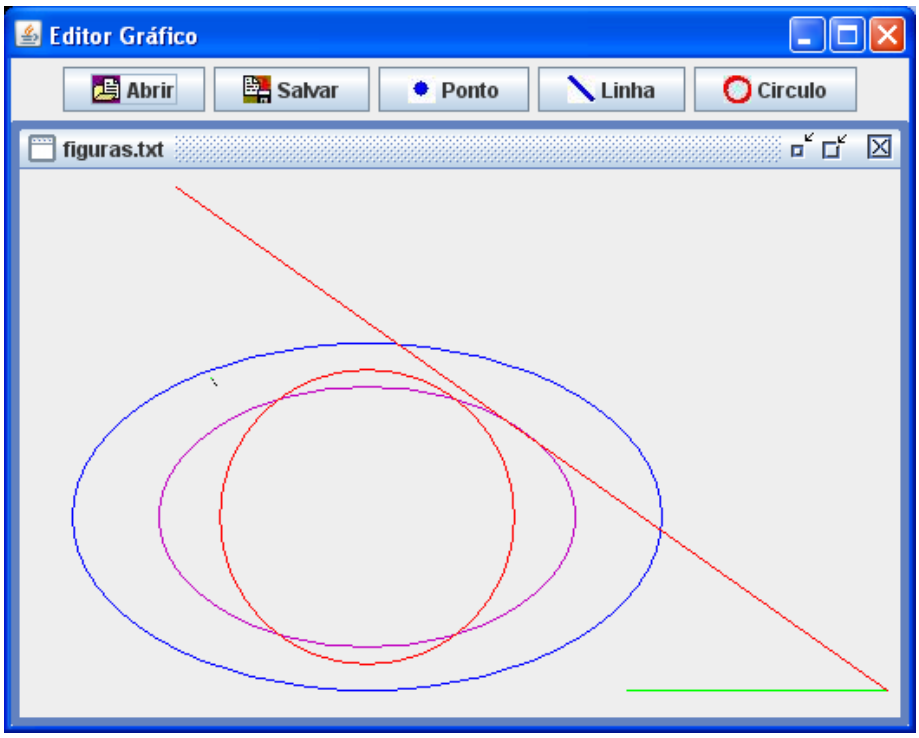
O código desse método chama o método repaint(), que dispara o evento paint() do Contexto Gráfico. Como o contexto passado foi o do painel pnlDesenho, é chamado o evento paintComponent() desse painel, que percorre o vetor de figuras e desenha suas figuras na área do contexto gráfico. O código vem abaixo:

```
public static void desenharObjetos(Graphics g)
{
   pnlDesenho.paintComponent(g);
}
```

O arquivo figuras.txt foi digitado como abaixo, para testarmos a leitura e exibição das figuras:

```
o      200  200    0    0  255  170  100
o      200  200  192    0  192  120   75
l      500  300    0  255    0  350  300
l      500  300  255    0    0   90   10
c      200  200  255    0    0   85
p      110  120    0  255    0
p      111  121    0    0    0
p      112  123    0    0    0
p      113  124    0    0    0
```

O resultado gerado vem abaixo:



Sites para consultas:

Programação gráfica em Java com Swing: http://www.guj.com.br/java.tutorial.artigo.38.1.guj

Sobre JFileChooser: http://java.sun.com/docs/books/tutorial/uiswing/components/filechooser.html
http://www.guj.com.br/posts/list/56458.java

Eventos do Mouse

As interfaces MouseListener e MouseMotionListener definem os métodos ouvintes para tratar eventos de cliques de mouse e de movimentação do cursor do mouse. Os métodos definidos em cada uma e que processam eventos de mouse são:

MouseListener:

- public void mousePressed(MouseEvent e)
  - chamado quando um botão do mouse é pressionado com o cursor do mouse sobre um componente
- public void mouseClicked(MouseEvent e)
  - chamado quando um botão do mouse é pressionado e liberado em componente sem mover o cursor do mouse
- public void mouseReleased(MouseEvent e)
  - chamado quando um botão do mouse é liberado depois de ser pressionado. Este evento sempre é precedido por um evento mousePressed
- public void Entered(MouseEvent e)
  - chamado quando o cursor do mouse entra dentro dos limites de um componente
- public void mouseExited(MouseEvent e)
  - chamado quando o cursor do mouse sai de dentro dos limites de um componente.

MouseMotionListener:

- public void mouseDragged(MouseEvent e)
  - chamado quando o botão do mouse é pressionado e o mouse é movido. Esse evento sempre é precedido por um evento mousePressed
- public void mouseMoved(MouseEvent e)
  - chamado quando o mouse é movido com o cursor em um componente

Todo componente visual do swing que tiver eventos de mouse deverá implementar MouseListener e MouseMotionListener, caso queira utilizar os métodos definidos nessas interfaces. Os componentes que não implementarem essas interfaces não serão afetados pelo mouse, mesmo que clicados. Em outras palavras, implementar essas interfaces para um componente significa programar o comportamento do aplicativo quando o mouse interagir com o componente para o qual um ouvinte de mouse está sendo definido.

Caso não seja necessário um dos eventos acima, deve-se ainda assim declará-lo, mas deixando seu corpo de comandos vazio.

No caso do nosso programa de editor gráfico, há diversos botões, cujos eventos precisam ser ouvidos e ações realizadas em resposta. Isso é feito através da implementação de action listeners para cada botão, e não envolve necessáriamente a criação de ouvintes de mouse. No entanto, quando se pressionar o botão btnLinha, por exemplo, devemos solicitar ao usuário que indique 2 pontos no painel de desenho, e a indicação de cada um dos pontos envolve um clique sobre o painel de desenho. Portanto, a maioria dos eventos de tratamento de mouse que implementaremos estarão associados ao pnlDesenho.

O parâmetro “MouseEvent e” é configurado para cada evento de mouse disparado. A variável e possui dois métodos, getX() e getY(), que devolvem a coordenada (X,Y) do cursor do mouse no momento em que o evento foi disparado. Portanto, quando o usuário pressionar o botão [btnPonto], esperaremos que ele indique um ponto no frame, logo em seguida. Podemos até mesmo colocar uma mensagem para o usuário informando a necessidade de ele indicar um ponto na área de desenho. Essa mensagem pode ser colocada no título da janela-filha frame.

No entanto, isso ainda não acessará o ponto indicado pelo usuário, que será fornecido através de um clique na área de desenho. Portanto, além do tratador de evento do btnPonto, teremos de criar um ouvinte de eventos de mouse e um tratador para cada tipo de evento, para obter a coordenada do ponto. Isso acontece porque é o clique do mouse sobre a área de desenho que informará o local onde o ponto se encontrará, e não o clique do botão. São dois ouvintes, dois tratadores diferentes, para objetos diferentes.

Começaremos o tratamento dos eventos de mouse modificando a classe interna MeuJPanel como se segue no código abaixo. Em primeiro lugar, devemos informar o compilador que essa classe implementará as interfaces de audição de eventos de mouse, ou seja, ela implementa MouseMotionListener e MouseListener:

```
private class MeuJPanel extends JPanel
      implements MouseMotionListener, MouseListener   → interfaces ouvintes de mouse
{
```

Podemos criar labels no fundo da janela filha para fornecer e solicitar informações ao usuário. Esses dois labels serão agrupados em um panel com layout de grid, com uma linha e duas colunas. Na coluna 1 (da esquerda) ficará uma área para mensagens ao usuário e na coluna 2 (da direita) exibiremos as coordenadas pelas quais o mouse passa.

Após o { da classe interna, declaramos a variável pnlStatus e a instanciamos, para que ela funcione como uma barra de status e armazene os dois labels que citamos acima. Esses labels devem ser declarados como variáveis globais da classe Editor, e serão instanciados no construtor da classe interna, que também vem modificado abaixo:

```
JPanel pnlStatus = new JPanel();                    → barra de status

public MeuJPanel() {
  super();
  pnlStatus.setLayout(new GridLayout(1,2));         → painel com 2 colunas
  statusBar1 = new JLabel("Mensagem");
  statusBar2 = new JLabel("Coordenada");
  pnlStatus.add(statusBar1);                        → label na coluna da esquerda
  pnlStatus.add(statusBar2);                        → label na coluna da direita
  getContentPane().add(pnlStatus,BorderLayout.SOUTH); → status no fundo do formulário
```

Os comandos acima formatam o painel de status usando o formato de Grade com uma linha e duas colunas.

Instancia os labels statusBar1 e statusBar2 e os adiciona ao conteúdo do painel de status. Em seguida, usa o método getContenPane() para obter a área de "desenho" desse MeuJPanel (que em nosso programa será o pnlDesenho), e adiciona o painel de status ao fundo dele, usando, para isso, o formato de layout BorderLayout.SOUTH).

Após isso, devemos indicar que o objeto sendo instanciado (MeuJPanel = this) ouve eventos de mouse e de movimentação de mouse, com os dois comandos abaixo:

```
            addMouseListener(this);              → esta classe "ouve" mouse
            addMouseMotionListener(this);        → e "ouve" também seus movimentos
      }
```

Com isso encerramos o construtor de MeuJPanel. Toda instância de MeuJPanel, como o pnlDesenho, terá um painel de status associado a ele e também ouvirá eventos de mouse e dos seus movimentos.

Quando uma classe implementa métodos de interfaces, todos os métodos declarados na interface devem ser implementados. Como a classe MeuJPanel implementa as interfaces MouseListener e MouseMotionListener, os sete métodos dessas duas interfaces devem ser implementados, como vemos abaixo. Note que o método mouseMoved, da interface MouseMotionListener, já possui código java e que este informa a coordenada por onde o mouse está passando, quando este está sobre o pnlDesenho (que é uma instância de MeuJPanel e, portanto, ouve e dispara os eventos de mouse):

```
      public void mouseMoved(MouseEvent e)      {
            statusBar2.setText("Coordenada: "+e.getX()+","+e.getY());
      }
      public void mouseDragged(MouseEvent e)    {
            // não faz nada por enquanto
      }
      public void mouseClicked (MouseEvent e)   {
            // não faz nada por enquanto
      }
      public void mousePressed (MouseEvent e)   {
            // não faz nada por enquanto
      }
      public void mouseEntered (MouseEvent e)   {
            // não faz nada por enquanto
      }
      public void mouseExited (MouseEvent e)    {
            // não faz nada por enquanto
      }
      public void mouseReleased (MouseEvent e)  {
            // não faz nada por enquanto
      }
```

```
      public void paintComponent(Graphics g)    {
        for (int atual = 0; atual < figuras.getTamanho(); atual++)
        {
            Ponto figuraAtual = figuras.valorDe(atual);
            figuraAtual.desenhar(figuraAtual.getCor(), g);
        }
      }
}
```

Como citado anteriormente, o parâmetro “MouseEvent e” possui métodos para obter informações sobre o uso do mouse, como coordenada do cursor (e.getX() e e.getY()), estado de teclas (tecla shift pressionada no clique do mouse, por exemplo).

O código acima, que está no quadro, é o evento paintComponent(), que já tínhamos escrito e que não será alterado.

Salve e execute seu programa. Note a barra de status no fundo da janela e como as coordenadas do mouse aparecerão nessa barra conforme ele é movido sobre a janela filha:

Vamos agora criar o tratamento de evento do botão btnPonto. Quando o usuário clicar nesse botão, o programa deverá informar que está esperando que o usuário indique um ponto sobre a área de desenho. Portanto, quando ocorrer um clique do mouse sobre esse componente (pnlDesenho), devemos capturar esse ponto, criar um objeto gráfico da classe Ponto com as coordenadas e armazená-lo no vetor de figuras.

O programa, portanto, deve saber, de alguma maneira, que naquele momento ele deve esperar por um ponto. Faremos isso com uma variável lógica esperaPonto que, quando valer true, significará que estamos no momento de esperar por um ponto.

Declare essa variável entre os atributos (variáveis globais), no início da classe Editor. Em seguida, dentro do construtor de Editor, associe o ouvinte DesenhaPonto ao botão btnPonto, com o comando

```
btnPonto.addActionListener(new DesenhaPonto());
```

Portanto, DesenhaPonto deverá ser uma classe interna que implementa a interface ActionListener e que possua um método actionPerformed, como fizemos com a classe FazAbertura, associada, anteriormente, ao btnAbrir.

Na classe DesenhaPonto, devemos mudar o texto exibido no statusBar1, exibindo uma mensagem ao usuário para que ele indique um ponto na área de desenho. Em seguida, devemos fazer a variável esperaPonto valer true, para indicar ao programa que neste momento o próximo clique do mouse corresponde ao ponto que se espera para criar um ponto no vetor de figuras e exibi-lo na tela. Isso será feito com os comandos abaixo:

```
private class DesenhaPonto implements ActionListener {
  public void actionPerformed(ActionEvent e)
  {
      statusBar1.setText("Mensagem: clique o local do ponto desejado:");
      limparEsperas();
      esperaPonto = true;
  }
}
```

limpaEsperas() é um método que deve colocar false em todas as variáveis lógicas que indicarão o que o programa está esperando no momento. Por exemplo, quando se pressiona o botão btnLinha, deve-se esperar o primeiro ponto e, em seguida, o segundo ponto. Portanto, devemos também declarar as variáveis lógicas globais esperaInicioLinha e esperaFimLinha. O código de limparEsperas(), portanto, será inicialmente o código abaixo. Quando você estiver desenvolvendo o restante do projeto, deverá agregar novas atribuições, para os parâmetros de círculos, ovais e quaisquer outros objetos geométricos que for incluir no projeto.

```
private void limpaEsperas()
{
  esperaPonto = false;
  esperaInicioLinha = false;
  esperaFimLinha = false;
}
```

É importante que todas as variáveis lógicas de “espera” sejam “limpas” com o valor false, de modo que apenas aquela que for tornada true no clique de um botão de desenho de figuras indique o que o programa está esperando no próximo clique.

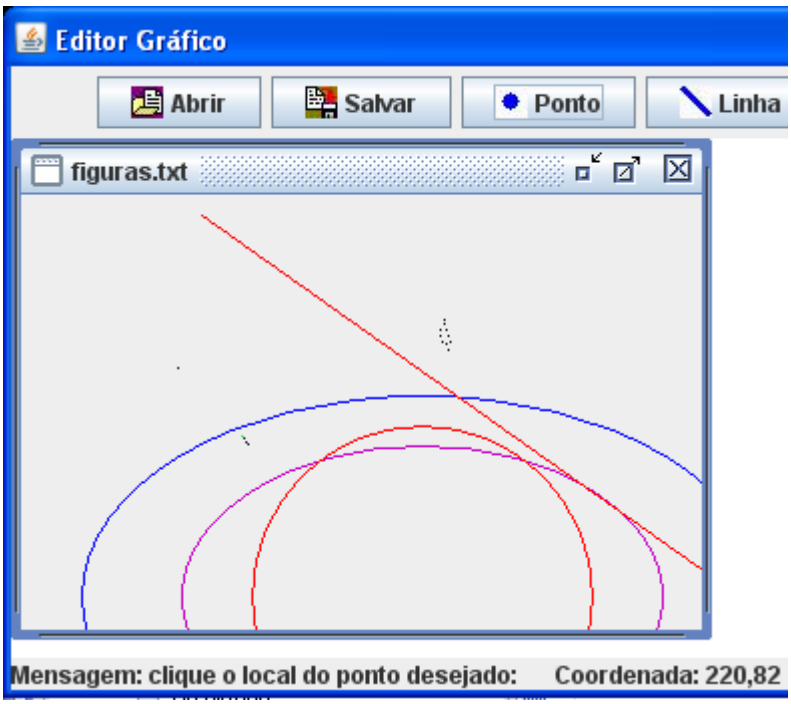
Temos agora que criar o código do tratador de evento mousePressed referente ao pnlDesenho. Nesse evento, precisamos descobrir o que o programa está esperando no momento. Como estamos criando o tratamento de ponto, devemos perguntar se o programa “esperaPonto” e, caso isso seja verdade, capturamos o ponto em que o mouse estava quando o evento foi disparado. Criamos uma instância da classe Ponto com essas coordenadas e com a cor atual e chamamos o método incluirNoFinal() do objeto de manutenção de figuras geométricas, que incluirá o novo Ponto, após a última posição usadae incrementao contador de figuras (quantosDados) e colocamos false na variável esperaPonto, uma vez que o ponto já foi fornecido. Também devemos limpar a mensagem que está no statusBar1, pois ela já foi atendida.

```
public void mousePressed (MouseEvent e)
{
  if (esperaPonto)
  {
      Ponto novoPonto = new Ponto(e.getX(), e.getY(), corAtual);
      figuras.incluirNoFinal(novoPonto);
      novoPonto.desenhar(novoPonto.getCor(), pnlDesenho.getGraphics());
      esperaPonto = false;
  }
}

public void mouseClicked (MouseEvent e)
{
  statusBar1.setText("Mensagem:");
}
```

Observe o uso da variável corAtual na criação da instância de Ponto, no método mousePressed. Essa variável deve ser declarada como atributo da classe Editor, do tipo Color, com o valor inicial de Color.black. Quando você programar o evento clique do btnCores, usará uma caixa de diálogo **JColorChooser** para definir o valor da cor atual escolhida pelo usuário, e essa cor será

armazenada em corAtual. Neste momento, usaremos a cor preta como padrão, mas já deixaremos a variável corAtual declarada e iniciada com Color.black.

Note também que desenhamos no pnlDesenho o ponto recém-criado e armazenado após a última posição usada do vetor de figuras. Após isso limpamos esperaPonto, para que qualquer outro clique que o usuário disparar não crie outro ponto.



Na figura ao lado podemos observar a execução do programa, no momento em que um ponto está sendo solicitado (veja o statusBar1 com a mensagem), e a coordenada atual também é exibida. Note que vários pontos foram já desenhados, sobre a reta vermelha inclinada.

Para terminar nosso estudo, devemos agora criar o tratador de evento para o btnLinha e, em seguida, tratar os cliques de mouse que correspondam à criação de uma linha reta.

No caso da linha reta, teremos de capturar duas posições clicadas sobre a área de desenho. Serão dois momentos diferentes, portanto teremos a necessidade de declarar as variáveis lógicas esperaInicioLinha e esperaFimLinha, que controlarão o fluxo de execução do tratador de evento mousePressed, de maneira semelhante ao que esperaPonto já faz.

Quando obtermos o primeiro ponto de uma linha, devemos guardá-lo para uso posterior, pois ainda não conhecemos o segundo ponto, que será definido apenas com o próximo lugar onde o usuário clicar o mouse na área de desenho. Para isso, usaremos uma nova variável p1, da classe Ponto. Essa variável guardará as coordenadas do primeiro ponto da linha, clicado logo após termos configurado esperaInicioLinha com true. Portanto, as declarações de variáveis de nosso programa devem estar semelhantes ao que vemos no código abaixo:

```
public class Editor extends JFrame
{
    static Color corAtual = Color.black;
    static boolean esperaPonto, esperaInicioLinha, esperaFimLinha;
    static private MeuJPanel pnlDesenho;
    private static ManterFiguras figuras; // objeto de manutenção de vetor
    private static JLabel statusBar1, statusBar2;
    private static Ponto p1 = new Ponto();
```

```
public Editor()   // construtor de Editor que criará o JFrame, ...
```

O código abaixo descreve o actionListener DesenhaReta, que deve ser associado ao btnLinha:

```
private class DesenhaReta implements ActionListener {
  public void actionPerformed(ActionEvent e)      {
    statusBar1.setText("Mensagem: clique o ponto inicial da reta");
```

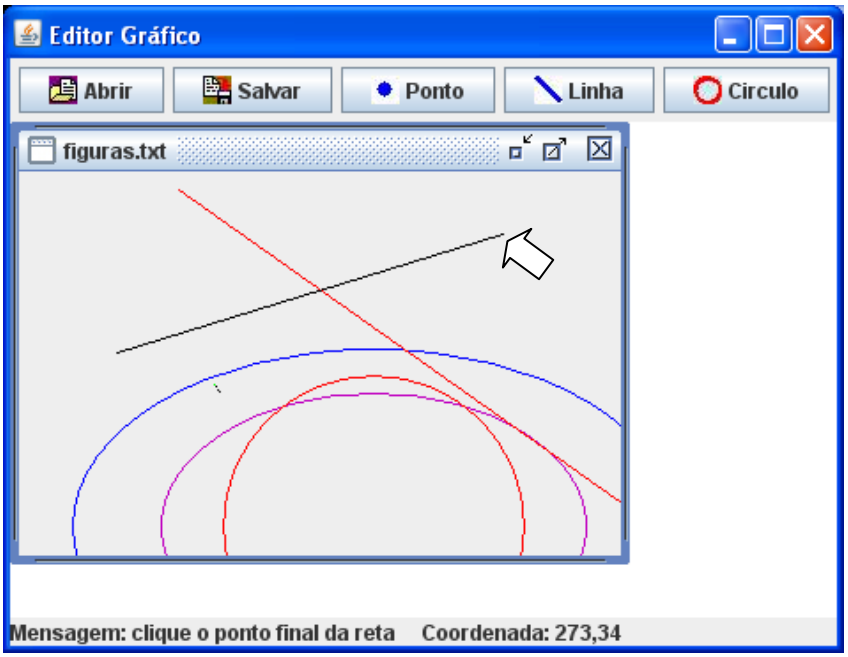
```
    limpaEsperas();
    esperaInicioReta = true;
  }
}
```

Esse código deve ser digitado após o actionListener DesenhaPonto. Observe que as variáveis de espera receberam false no método limpaEsperas() e que DesenhaReta configura apenas esperaInicioReta com true, indicando que, neste momento, o programa espera pelo primeiro ponto da linha reta. O código abaixo é o evento mousePressed como deve ficar para capturar o primeiro ponto da reta:

```
public void mousePressed (MouseEvent e)
{
  if (esperaPonto)
  {
      Ponto pontoInicial = new Ponto(e.getX(), e.getY(), corAtual);
      figuras.InsereAposFim(new ListaSimples.NoLista(pontoInicial, null));
      pontoInicial.desenha(pontoInicial.getCor(), pnlDesenho.getGraphics());
      esperaPonto = false;
  }
  else
      if (esperaInicioReta)
      {
        p1.setCor(corAtual);
        p1.setX(e.getX());
        p1.setY(e.getY());
        esperaInicioReta = false;
        esperaFimReta    = true;
        statusBar1.setText("Mensagem: clique o ponto final da reta");
      }
```

Observe que foi colocado false em esperaInicioReta e true em esperaFimReta. Dessa forma, após termos capturado o primeiro ponto e armazenado-o em p1, colocamos o programa em modo de “espera pelo ponto final da reta”, com a mensagem no statusBar1 indicando isso. Podemos já codificar o tratamento do ponto final da reta, que seria tratado também no evento mousePressed, como todo evento de clique do mouse neste programa:

```
      else
        if (esperaFimReta)
        {
            Ponto pontoFinal = new Ponto(e.getX(), e.getY(), corAtual);
            Linha novaLinha = new Linha(p1, pontoFinal, corAtual);
            figuras.incluirNoFinal(novaLinha);
            novaLinha.desenhar(corAtual, pnlDesenho.getGraphics());
            esperaFimLinha = false;  // linha terminada e incluída
        }
}
```



Projeto para avaliação – em dupla - entrega em 19/10/2025, no Classroom.

Levando em conta o que já desenvolvemos, você deverá implementar as seguintes funcionalidades neste programa:

1. solicitar e desenhar oval
2. solicitar e desenhar círculo
3. derivar a classe Retângulo a partir de Ponto e implementar sua solicitação e desenho
4. derivar a classe Polilinha e implementar sua solicitação e desenho (vetor de n Pontos)
5. escolher a cor atual de desenho, a partir do que as novas figuras serão desenhadas nessa cor
6. Implementar um botão [Selecionar] que, quando clicado, solicite a digitação de um índice do vetor de figuras geométricas, desenhe a figura correspondente com 2 pixels a mais de espessura e também a inclua em um vetor de índices de figuras geométricas selecionadas.
7. Mudar a cor das figuras geométricas selecionadas.
8. Somar um deslocamento (delta X, delta Y) nas figuras geométricas selecionadas, redesenhando-as.

9. Apagar as figuras geométricas selecionadas do desenho principal (as figuras deixam de ser selecionadas)
10. Limpar o vetor de figuras geométricas selecionadas, redesenhando o desenho principal.
11. Limpar o vetor de figuras geométricas do desenho principal e a área de desenho.