

6

# GRAPHICAL USER INTERFACES

## Paradigmas de Programação

### LEI - ISEP

Luiz Faria, adaptado de Donald W. Smith (TechNeTrain.com)

## Objetivos

- ❑ Implementar interfaces gráficas simples com o utilizador
- ❑ Adicionar botões, campos de texto e outros componentes a uma *frame window*
- ❑ Tratar eventos que são gerados pelos componentes da GUI
- ❑ Utilizar gestores de posicionamento para organizar componentes num contentor
- ❑ Escrever programas que desenham gráficos simples

2

## Conteúdos

- ❑ *Frame Windows*
- ❑ Eventos e Tratamento de Eventos
- ❑ Processamento de Entradas de Texto
- ❑ Criação de Desenhos
- ❑ Gestores de Posicionamento
- ❑ Componentes de seleção (*radio buttons*, *check* e *combo boxes*)
- ❑ *Timer Events*
- ❑ *Mouse Events*

3

## *Frame Windows*

- ❑ O Java dispõe de classes para criar aplicações gráficas que podem correr em qualquer interface gráfica com o utilizador
  - Uma aplicação gráfica mostra informação no interior de uma *frame*: uma janela com uma barra de título
- ❑ A classe `JFrame` permite apresentar uma *frame*
  - Contida no package `javax.swing`



4

## Classe JFrame

- ❑ Cinco passos para mostrar uma *frame*:

1) Construir um objeto da classe JFrame

```
JFrame frame = new JFrame();
```

2) Definir as dimensões da *frame*

```
frame.setSize(300,400);
```

3) Definir o título da *frame*

```
frame.setTitle("An Empty Frame");
```

4) Definir o código para a “default close operation”

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

5) Tornar a *frame* visível

```
frame.setVisible(true);
```



5

## EmptyFrameViewer.java

- ❑ A JVM (Java Virtual Machine) encarrega-se de todo o trabalho para mostrar a *frame* na GUI

```
1 import javax.swing.JFrame;           Utilização da biblioteca java Swing
2 /**
3  * This program displays an empty frame.
4  */
5 public class EmptyFrameViewer
6 {
7     public static void main(String[] args)
8     {
9         JFrame frame = new JFrame();
10
11         final int FRAME_WIDTH = 300;
12         final int FRAME_HEIGHT = 400;
13         frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
14         frame.setTitle("An empty frame");
15         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16
17         frame.setVisible(true);
18     }
19 }
20 }
```

6

## Adicionando Componentes à GUI

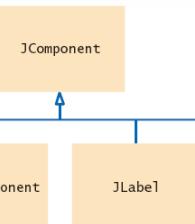
- ❑ Não é possível desenhar diretamente sobre um objeto `JFrame`

- ❑ Constrói-se um objeto e adiciona-se à *frame*

- Alguns exemplos de objetos que podem ser adicionados:
  - `JComponent`
  - `Jpanel`
  - `JTextComponent`
  - `JLabel`

```
public class RectangleComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        // Instruções de desenho
    }
}
```

Estende a classe `Jcomponent` e reescreve o método `paintComponent`



7

## Adicionando *Panels*

- ❑ Se temos mais do que um componente, podemos colocá-los num *panel* (um contentor de outros componentes da user-interface) e depois adicionar o *panel* à *frame*:

- 1: Criar os componentes

```
JButton button = new JButton("Click me!");
JLabel label = new JLabel("Hello, World!");
```

- 2: Adicioná-los ao *panel*

```
JPanel panel = new JPanel();
panel.add(button);
panel.add(label);
frame.add(panel);
```

Usar um `JPanel` para agrupar vários componentes gráficos

Adicionar o *panel* à *frame*

8

## FilledFrameViewer.java

```
1 import javax.swing.JButton;
2 import javax.swing.JFrame;
3 import javax.swing.JLabel;
4 import javax.swing.JPanel;
5
6 /**
7  * This program shows a frame that is filled with two components.
8 */
9 public class FilledFrameViewer
10 {
11     public static void main(String[] args)
12     {
13         JFrame frame = new JFrame();
14
15         JButton button = new JButton("Click me!");
16         JLabel label = new JLabel("Hello, World!");
17
18         JPanel panel = new JPanel();
19         panel.add(button);
20         panel.add(label);
21         frame.add(panel);
22
23         final int FRAME_WIDTH = 300;
24         final int FRAME_HEIGHT = 100;
25         frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
26         frame.setTitle("A frame with two components");
27         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28
29         frame.setVisible(true);
30     }
31 }
```

9

## Usar Herança para Personalizar Frames

### □ Para frames complexos :

- Definir uma subclasse de JFrame
- Armazenar os componentes como variáveis de instância
- Inicializá-los no construtor da subclasse

```
public class FilledFrame extends JFrame
{
    private JButton button; Os componentes são variáveis de instância
    private JLabel label;
    private static final int FRAME_WIDTH = 300;
    private static final int FRAME_HEIGHT = 100;

    public FilledFrame()
    {
        createComponents();
        setSize(FRAME_WIDTH, FRAME_HEIGHT);
    }
}
```

No construtor da subclasse,  
inicializar e adicioná-los com a  
ajuda de um método auxiliar

10

## Usar Herança para Personalizar *Frames*

- Depois instanciar a *frame* personalizada a partir do método `main`

```
public class FilledFrameViewer2
{
    public static void main(String[] args)
    {
        JFrame frame = new FilledFrame();
        frame.setTitle("A frame with two components");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

11

## Adicionar o método `main` à classe *Frame*

- Alguns programadores preferem esta técnica

```
public class FilledFrame extends JFrame
{
    ...
    public static void main(String[] args)
    {
        JFrame frame = new FilledFrame();
        frame.setTitle("A frame with two components");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
    public FilledFrame()
    {
        createComponents();
        setSize(FRAME_WIDTH, FRAME_HEIGHT);
    }
    ...
}
```

Uma vez que `main` instanciou a `FilledFrame`, temos acesso às variáveis e métodos de instância

12

## Eventos e Tratamento de Eventos

- ❑ Nas aplicações atuais o utilizador interage com as respectivas interfaces através do rato e teclado
- ❑ O utilizador pode introduzir texto em campo de texto, selecionar *pull down menus*, pressionar botões, ... -> eventos
  - O programa deve reagir às ações do utilizador
  - O programa pode optar por receber e tratar eventos tais como “mouse move” ou botão pressionado “action event”

13

## Eventos e Tratamento de Eventos

- ❑ Os programas devem indicar que eventos pretendem receber
- ❑ Através da instalação de *event listener objects*
  - Um *event listener object* pertence a uma classe declarada por nós
  - Os métodos das nossas *event listener classes* contêm as instruções que devem ser executadas perante a ocorrência do evento
- ❑ Para instalar um *listener*, é necessário conhecer a fonte do evento (*event source*)
- ❑ Podemos adicionar um *event listener object* a um conjunto selecionado de *event sources*:
  - Exemplos: *OK Button clicked*, *Cancel Button clicked*, *Menu Choice* ...
- ❑ Sempre que o evento ocorre, o *event source* chama os métodos apropriados de todos os *event listeners* instalados

14

## Exemplo: ActionListener

- A interface ActionListener tem um método:

```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```

- A classe ClickListener implementa a interface ActionListener

```
1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3
4 /**
5  * An action listener that prints a message.
6 */
7 public class ClickListener implements ActionListener
8 {
9     public void actionPerformed(ActionEvent event)
10    {
11        System.out.println("I was clicked.");
12    }
13 }
```

Podemos ignorar o parâmetro evento – este contém informação tal como quando ocorreu o evento

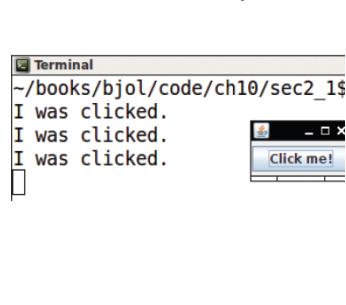
15

## Registo do ActionListener

- Criar um objeto ClickListener e ‘registar’ (adicionar) numa event source específica

```
ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

- Após o registo, sempre que o objeto button é pressionado, este irá chamar `listener.ActionPerformed`, passando o evento como parâmetro



```
1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3
4 /**
5  * An action listener that prints a message.
6 */
7 public class ClickListener implements ActionListener
8 {
9     public void actionPerformed(ActionEvent event)
10    {
11        System.out.println("I was clicked.");
12    }
13 }
```

16

## ButtonFrame1.java

```
6  /**
7   * This frame demonstrates how to install an action listener.
8  */
9  public class ButtonFrame1 extends JFrame
{
10 {
11     private static final int FRAME_WIDTH = 100;
12     private static final int FRAME_HEIGHT = 60;
13
14     public ButtonFrame1()
15     {
16         createComponents();
17         setSize(FRAME_WIDTH, FRAME_HEIGHT);
18     }
19
20     private void createComponents()
21     {
22         JButton button = new JButton("Click me!");
23         JPanel panel = new JPanel();
24         panel.add(button);
25         add(panel);
26
27         ActionListener listener = new ClickListener();
28         button.addActionListener(listener);
29     }
30 }
```

Cria e adiciona um JButton ao frame

Instrui o botão para invocar o objeto *listener* sempre que o botão é pressionado

17

## ButtonViewer1.java

- ❑ Não é necessário efetuar qualquer alteração em main para implementar um *event handler*

```
1  import javax.swing.JFrame;
2
3  /**
4   * This program demonstrates how to install an action listener.
5  */
6  public class ButtonViewer1
{
7
8     public static void main(String[] args)
9     {
10        JFrame frame = new ButtonFrame1();
11        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12        frame.setVisible(true);
13    }
14 }
```

18

## Listeners como Inner Classes

- ❑ Anteriormente, vimos que o código a executar perante um evento *button clicked* é colocado num objeto de uma classe *listener*
- ❑ As *Inner Classes* são muitas vezes usadas como *ActionListeners*
- ❑ Uma *Inner class* é uma classe declarada no interior de outra classe
  - Pode ser declarada dentro ou fora de um método da classe
- ❑ Porquê usar *inner classes*? Duas razões:
  - 1) Coloca a *listener class* exatamente onde é necessária
  - 2) Os seus métodos têm acesso a variáveis que são declaradas à sua volta
    - Neste aspeto, as *inner classes* declaradas dentro de métodos comportam-se de forma semelhante a blocos embutidos (*nested blocks*)

19

## Exemplo: Inner Class Listener

- ❑ A *inner class ClickListener* declarada no interior da classe *ButtonFrame2* pode aceder a variáveis locais declaradas no bloco que contém a classe

```
public class ButtonFrame2 extends JFrame
{
    private JButton button;
    private JLabel label;
    ...
    class ClickListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            label.setText("I was clicked");
        }
    ...
}
```

Outer Block      Inner Block

É possível aceder a métodos da instância privada de um objeto *label*

20

## ButtonFrame2.java (1)

```
1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3 import javax.swing.JButton;
4 import javax.swing.JFrame;
5 import javax.swing.JLabel;
6 import javax.swing.JPanel;
7
8 public class ButtonFrame2 extends JFrame
9 {
10     private JButton button;
11     private JLabel label;
12
13     private static final int FRAME_WIDTH = 300;
14     private static final int FRAME_HEIGHT = 100;
15
16     public ButtonFrame2()
17     {
18         createComponents();
19         setSize(FRAME_WIDTH, FRAME_HEIGHT);
20     }
21 }
```

21

## ButtonFrame2.java (2)

- Altera *label* de “Hello World!” para “I was clicked.”:

```
22 /**
23  * An action listener that changes the label text.
24 */
25 class ClickListener implements ActionListener
26 {
27     public void actionPerformed(ActionEvent event)
28     {
29         label.setText("I was clicked.");
30     }
31 }
32
33 private void createComponents()
34 {
35     button = new JButton("Click me!");
36     ActionListener listener = new ClickListener();
37     button.addActionListener(listener);
38
39     label = new JLabel("Hello, World!");
40
41     JPanel panel = new JPanel();
42     panel.add(button);
43     panel.add(label);
44     add(panel);
45 }
46 }
```

22

## InvestmentFrame.java (1)

```
1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3 import javax.swing.JButton;
4 import javax.swing.JFrame;
5 import javax.swing.JLabel;
6 import javax.swing.JPanel;
7
8 public class InvestmentFrame extends JFrame
9 {
10     private JButton button;
11     private JLabel resultLabel;
12     private double balance;
13
14     private static final int FRAME_WIDTH = 300;
15     private static final int FRAME_HEIGHT = 100;
16
17     private static final double INTEREST_RATE = 5;
18     private static final double INITIAL_BALANCE = 1000;
19
20     public InvestmentFrame()
21     {
22         balance = INITIAL_BALANCE;
23
24         createComponents();
25         setSize(FRAME_WIDTH, FRAME_HEIGHT);
26     }
```

23

## InvestmentFrame.java (2)

```
28 /**
29  * Adds interest to the balance and updates the display.
30 */
31 class AddInterestListener implements ActionListener
32 {
33     public void actionPerformed(ActionEvent event)
34     {
35         double interest = balance * INTEREST_RATE / 100;
36         balance = balance + interest;
37         resultLabel.setText("Balance: " + balance);
38     }
39 }
40
41 private void createComponents()
42 {
43     button = new JButton("Add Interest");
44     ActionListener listener = new AddInterestListener();
45     button.addActionListener(listener);
46
47     resultLabel = new JLabel("Balance: " + balance);
48
49     JPanel panel = new JPanel();
50     panel.add(button);
51     panel.add(resultLabel);
52     add(panel);
53 }
54 }
```



24

## Erro Frequentes (1)



- Alteração dos parâmetros na implementação de um método de uma interface

```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```

- Na implementação da interface, é necessário declarar cada método exatamente como especificado na interface
- Exemplo de erro acidental:

```
class MyListener implements ActionListener
{
    public void actionPerformed()
        // Oops . . . Esquecimento do parâmetro
        ActionEvent
    {
        . . .
    }
}
```

25

## Erro Frequentes (2)



- Esquecer de adicionar (registar) um *Listener*
  - Se na execução do programa não houver reação às interações com um botão, verificar se o *listener* foi adicionado ao botão
  - O mesmo se aplica a outros componentes. É frequente criar a classe *listener* e esquecer de adicionar o *listener* ao *event source*

```
...
ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

26

## Processamento de Entrada de Texto

- ❑ A classe JTextField fornece um campo de texto editável
  - Na construção de um campo de texto, definir:
    - Número aproximado de caracteres esperado
    - Se o utilizador excede este número, o texto desloca-se para a esquerda (*scroll*)

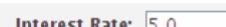


```
final int FIELD_WIDTH = 10;
final JTextField rateField = new JTextField(FIELD_WIDTH);
```

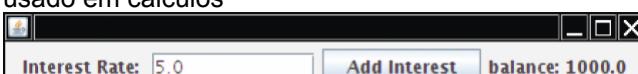
27

## Adicionar um *Label* e um Botão

- ❑ Um *Label* ajuda o utilizador a perceber o que se pretende que este introduza
  - Normalmente à esquerda da *textbox*
- ❑ Um Botão com um método actionPerformed pode ser usado para ler o texto da *textbox* com o método `getText`
  - `getText` devolve uma String e deve ser convertido para um valor numérico se usado em cálculos



```
JLabel rateLabel = new JLabel("Interest Rate: ");
```



```
double rate = Double.parseDouble(rateField.getText());
double interest = account.getBalance() * rate / 100;
account.deposit(interest);
resultLabel.setText("balance: " + account.getBalance());
```

28

## InvestmentFrame2.java

```
9  /**
10   * A frame that shows the growth of an investment with variable interest.
11  */
12 public class InvestmentFrame2 extends JFrame
13 {
14     private static final int FRAME_WIDTH = 450;
15     private static final int FRAME_HEIGHT = 100;
16
17     private static final double DEFAULT_RATE = 5;
18     private static final double INITIAL_BALANCE = 1000;
19
20     private JLabel rateLabel;
21     private JTextField rateField;
22     private JButton button;
23     private JLabel resultLabel;
24     private double balance;
25
26     public InvestmentFrame2()
27     {
28         balance = INITIAL_BALANCE;
29
30         resultLabel = new JLabel("Balance: " + balance);
31
32         createTextField();
33         createButton();
34         createPanel();
35     }
```

Colocar os componentes de entrada no interior da frame

29

## InvestmentFrame2.java (2)

```
39     private void createTextField()
40     {
41         rateLabel = new JLabel("Interest Rate: ");
42
43         final int FIELD_WIDTH = 10;
44         rateField = new JTextField(FIELD_WIDTH);
45         rateField.setText("" + DEFAULT_RATE);
46     }
47
48     /**
49      * Adds interest to the balance and updates the display.
50     */
51     class AddInterestListener implements ActionListener
52     {
53         public void actionPerformed(ActionEvent event)
54         {
55             double rate = Double.parseDouble(rateField.getText());
56             double interest = balance * rate / 100;
57             balance = balance + interest;
58             resultLabel.setText("Balance: " + balance);
59         }
60     }
61
62     private void createButton()
63     {
64         button = new JButton("Add Interest");
65
66         ActionListener listener = new AddInterestListener();
67         button.addActionListener(listener);
68     }
```

Implementar os cálculos no método  
ActionPerformed

Manter o código para  
o *listener* e o objeto  
(Botão) no mesmo  
local

30

## Áreas de Texto

- Áreas de texto com várias linhas: `JTextArea`
  - Definir as dimensões em número de linhas e de colunas

```
final int ROWS = 10;
final int COLUMNS = 30;
JTextArea textArea = new JTextArea(ROWS, COLUMNS);
```
  - Usar o método `setText` para definir o texto de `JTextArea` ou `JTextField`

```
textArea.setText("Account Balance");
```

31

## Áreas de Texto

- O método `append` adiciona texto ao fim da *text area*
  - Usar *newline* para quebrar linhas

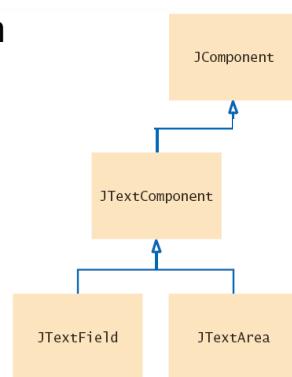
```
textArea.append(account.getBalance() + "\n");
```
- Usar o método `setEditable` para controlar entrada do utilizador

```
textArea.setEditable(false);
```

32

## JTextField e JTextArea

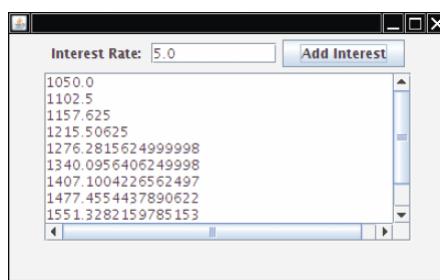
- ❑ JTextField e JTextArea herdam de JTextComponent:
  - setText
  - setEditable



33

## JTextField e JTextArea

- ❑ O método append está declarado na classe JTextArea



- ❑ Para adicionar *scroll bars*, usar JScrollPane:

```
JScrollPane scrollPane = new JScrollPane(textArea);
```

34

## InvestmentFrame3.java (1)

```
11  /**
12   * A frame that shows the growth of an investment with variable interest,
13   * using a text area.
14  */
15 public class InvestmentFrame3 extends JFrame
16 {
17     private static final int FRAME_WIDTH = 400;
18     private static final int FRAME_HEIGHT = 250;
19
20     private static final int AREA_ROWS = 10;
21     private static final int AREA_COLUMNS = 30;
22
23     private static final double DEFAULT_RATE = 5;
24     private static final double INITIAL_BALANCE = 1000;
25
26     private JLabel rateLabel;
27     private JTextField rateField;
28     private JButton button;
29     private JTextArea resultArea;
30     private double balance;
31 }
```

Declarar os componentes a usar

35

## InvestmentFrame.java (2)

```
32     public InvestmentFrame3()
33     {
34         balance = INITIAL_BALANCE;
35         resultArea = new JTextArea(AREA_ROWS, AREA_COLUMNS);
36         resultArea.setText(balance + "\n");
37         resultArea.setEditable(false);
38
39         createTextField();
40         createButton();
41         createPanel();
42
43         setSize(FRAME_WIDTH, FRAME_HEIGHT);
44     }
45
46     private void createTextField()
47     {
48         rateLabel = new JL74         private void createPanel()
49         final int FIELD_WI75         {
50             rateField = new JT76             JPanel = new JPanel();
51             rateField.setT77             panel1.add(rateLabel);
52             rateField.setT78             panel1.add(rateField);
53         }                           79             panel1.add(button);
80             JScrollPane scrollPane = new JScrollPane(resultArea);
81             panel1.add(scrollPane);
82         }                           83             add(panel1);
83     }                           84 }
```

O construtor chama métodos  
que criam os componentes

36

## InvestmentFrame.java (3)

```
54  class AddInterestListener implements ActionListener
55  {
56      public void actionPerformed(ActionEvent event)
57      {
58          double rate = Double.parseDouble(rateField.getText());
59          double interest = balance * rate / 100;
60          balance = balance + interest;
61          resultArea.append(balance + "\n");
62      }
63  }
64
65  private void createButton()
66  {
67      button = new JButton("Add Interest");
68
69      ActionListener listener = new AddInterestListener();
70      button.addActionListener(listener);
71  }
72 }
```

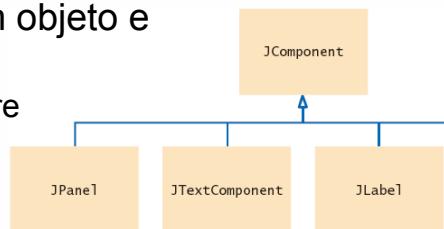
Classe *listener* e o método associado *createButton*

37

## Componentes de Desenho

- ❑ Não é possível desenhar diretamente sobre um objeto JFrame
- ❑ Em vez disso, construir um objeto e adicioná-lo à *frame*

- Alguns tipos de objetos sobre os quais se pode desenhar:
  - JPanel
  - JTextField
  - JLabel



```
public class chartComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        // Instruções de desenho
    }
}
```

Estende a classe JComponent e reescreve o método paintComponent<sup>29</sup>

## Método paintComponent

- ❑ O método **paintComponent** é chamado automaticamente quando:
  - O componente é apresentado pela primeira vez
  - Sempre que a janela é redimensionada ou após ser visualizada depois de ter estado escondida

```
public class chartComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        g.fillRect(0, 10, 200, 10);
        g.fillRect(0, 30, 300, 10);
        g.fillRect(0, 50, 100, 10);
    }
}
```

(0, 0)

(10, 20)

(20, 10)

A bar chart

x

y

39

## ChartComponent.java

```
1 import java.awt.Graphics;
2 import javax.swing.JComponent;
3
4 /**
5  * A component that draws a bar chart.
6 */
7 public class ChartComponent extends JComponent
8 {
9     public void paintComponent(Graphics g)
10    {
11        g.fillRect(0, 10, 200, 10);
12        g.fillRect(0, 30, 300, 10);
13        g.fillRect(0, 50, 100, 10);
14    }
15 }
```

A classe **Graphics** é parte do  
package **java.awt**

- ❑ Temos agora um objeto **Jcomponent** que pode ser adicionado a um **Jframe**

40

## ChartViewer.java

```
1 import javax.swing.JComponent;
2 import javax.swing.JFrame;
3
4 public class ChartViewer
5 {
6     public static void main(String[] args)
7     {
8         JFrame frame = new JFrame();
9
10        frame.setSize(400, 200);
11        frame.setTitle("A bar chart");
12        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14        JComponent component = new ChartComponent();
15        frame.add(component);
16
17        frame.setVisible(true);    Adicionar o componente ao frame
18    }
19}
```

41

## Parâmetro Graphics

- O método `paintComponent` recebe um objeto do tipo **Graphics**
  - O objeto **Graphics** guarda o estado dos gráficos
    - A cor, fonte, etc. atuais, usadas nas operações de desenho
  - A classe **Graphics2D** estende a classe **Graphics**
    - Oferece métodos mais completos para desenhar objetos 2D
    - No contexto do Swing, o parâmetro **Graphics** é na realidade do tipo **Graphics2D**, sendo por isso necessário convertê-lo para **Graphics2D** para que se possa usá-lo

```
public class RectangleComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
    }
}
```

42

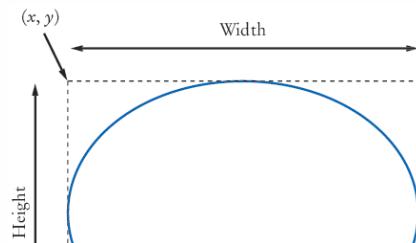
## Ovais, Linhas, Texto e Cor

- ❑ Elipses são definidas a partir de um rectângulo usado para definir os seus limites:
  - Fornecer as coordenadas x e y do canto superior esquerdo
  - Fornecer a largura e altura do rectângulo que define o limite
  - Usar o método `drawOval` da classe `Graphics` criar uma elipse

```
g.drawOval(x, y, width, height);
```

- Usar `drawLine` entre dois pontos:

```
g.drawLine(x1, y1, x1, y2);
```



43

## Desenhar Texto

- ❑ Usar o método `drawString` da classe `Graphics` para desenhar uma string numa janela
  - Especificar a string
  - Especificar o *Basepoint* (coordenadas x e y)
  - A *Baseline* é a coordenada y do *Basepoint*

```
g2.drawString("Message", 50, 100);
```



44

## Cor

- ❑ Por defeito todos os gráficos e strings são desenhados a preto e preenchidos a branco
- ❑ Para alterar a cor, chamar `setColor` com um objeto do tipo `Color`
  - Java usa o modelo de cor RGB
  - É possível usar cores pré-definidas ou criar outras cores

```
g.setColor(Color.YELLOW);
g.fillRect(350, 25, 35, 20);
```

Todas os elementos desenhados após `setColor` irão usar a nova cor

Color	RGB Value
Color.BLACK	0, 0, 0
Color.BLUE	0, 0, 255
Color.CYAN	0, 255, 255
Color.GRAY	128, 128, 128
Color.DARKGRAY	64, 64, 64
Color.LIGHTGRAY	192, 192, 192
Color.GREEN	0, 255, 0
Color.MAGENTA	255, 0, 255
Color.ORANGE	255, 200, 0
Color.PINK	255, 175, 175
Color.RED	255, 0, 0
Color.WHITE	255, 255, 255
Color.YELLOW	255, 255, 0

45

## ChartComponent2.java

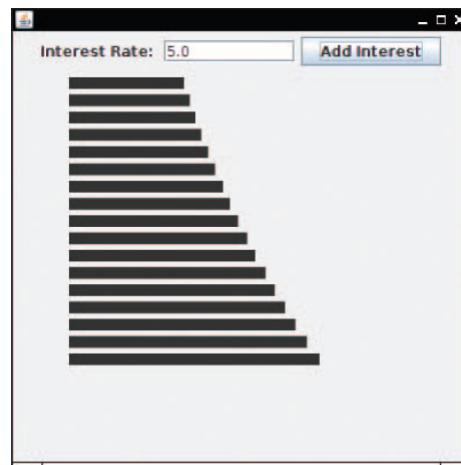
```
1 import java.awt.Color;
2 import java.awt.Graphics;
3 import javax.swing.JComponent;
4 
5 /**
6  * A component that draws a demo chart.
7 */
8 public class ChartComponent2 extends JComponent
{
9     public void paintComponent(Graphics g)
10    {
11        // Draw the bars
12        g.fillRect(0, 10, 200, 10);
13        g.fillRect(0, 30, 300, 10);
14        g.fillRect(0, 50, 100, 10);
15
16        // Draw the arrow
17        g.drawLine(350, 35, 305, 35);
18        g.drawLine(305, 35, 310, 30);
19        g.drawLine(305, 35, 310, 40);
20
21        // Draw the highlight and the text
22        g.setColor(Color.YELLOW);
23        g.fillOval(350, 25, 35, 20);
24        g.setColor(Color.BLACK);
25        g.drawString("Best", 355, 40);
26    }
27 }
28 }
```



46

## Aplicação: Evolução de Investimento

- ❑ Introduzir a taxa de juro
- ❑ Pressionar o botão *Add Interest* para adicionar barras ao gráfico
- ❑ Manter uma lista de valores para redesenhar todas as barras



47

## ChartComponent.java

```
6  /**
7   * A component that draws a chart.
8  */
9  public class ChartComponent extends JPanel
10 {
11     private ArrayList<Double> values;
12     private double maxValue;
13
14     public ChartComponent(double max)
15     {
16         values = new ArrayList<Double>();
17         maxValue = max;
18     }
19
20     public void append(double value)
21     {
22         values.add(value);
23         repaint();
24     }
25
26     public void paintComponent(Graphics g)
27     {
28         final int GAP = 5;
29         final int BAR_HEIGHT = 10;
30
31         int y = GAP;
32         for (double value : values)
33         {
34             int barWidth = (int) (getWidth() * value / maxValue);
35             g.fillRect(0, y, barWidth, BAR_HEIGHT);
36             y = y + BAR_HEIGHT + GAP;
37         }
38     }
39 }
```

Adicionar um novo valor ao ArrayList

Usar um ArrayList para guardar os valores das barras

Desenhar as barras num ciclo

48

## InvestmentFrame4.java (1)

```
10  /**
11   * A frame that shows the growth of an investment with variable interest,
12   * using a bar chart.
13  */
14  public class InvestmentFrame4 extends JFrame
15  {
16      ...
17      public InvestmentFrame4()
18      {
19          ...
20          ...
21          ...
22          ...
23          ...
24          ...
25          ...
26          ...
27          ...
28          ...
29          ...
30          ...
31          ...
32          ...
33          ...
34          ...
35          ...
36          ...
37          ...
38          ...
39          ...
40          ...
41          ...
42          ...
43          ...
44          ...
45      private void createTextField()
46      {
47          ...
48          ...
49          ...
50          ...
51          ...
52      }
```

Instância e inicializa o componente ChartComponent

Usa métodos auxiliares para criar os componentes

49

## InvestmentFrame4.java (2)

```
54  class AddInterestListener implements ActionListener
55  {
56      ...
57      public void actionPerformed(ActionEvent event)
58      {
59          ...
60          ...
61          ...
62      }
63  }
64  private void createButton()
65  {
66      ...
67      ...
68      ...
69      ...
70      ...
71  }
72
73  private void createPanel()
74  {
75      ...
76      ...
77      ...
78      ...
79      ...
80      ...
81  }
82 }
```

Definição do Listener e do Botão

50

## Erro Comum (1)



- ❑ Omissão de redesenho (*repaint*)
  - Quando se altera o conteúdo de um componente, este não é redesenhadado automaticamente
  - É necessário chamar o método `repaint` do componente
  - O melhor local para chamar `repaint` é no método cujo conteúdo do componente é alterado:

```
void changeData(. . .)
{
    // Actualizar conteúdo do componente
    repaint();
}
```

51

## Erro Comum (2)



- ❑ Por defeito, os componentes têm largura e altura nulas
  - É necessário ter atenção quando se adiciona um componente desenhado a um painel (*panel*)
  - A dimensão por defeito de um *JComponent* é 0 por 0 pixéis – o componente não será visível
  - A solução é chamar o método `setPreferredSize`:

```
chart.setPreferredSize(new Dimension(CHART_WIDTH, CHART_HEIGHT));
```

52

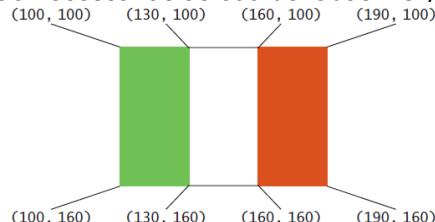
## Passos para Desenhar Formas

### 1) Definir as formas necessárias ao desenho

- Quadrados e retângulos
- Círculos e elipses
- Linhas e texto

### 2) Definir as coordenadas de cada forma

- Para retângulos e elipses é necessário conhecer o canto superior esquerdo, largura e altura
- Para linhas são necessárias as coordenadas x e y dos pontos de início e de fim
- Para texto são necessárias as coordenadas x e y do *basepoint*



53

## Passos para Desenhar Formas

### 3) Escrever as instruções Java para desenhar as formas

```
g.setColor(Color.GREEN);
g.fillRect(100, 100, 30, 60);
g.setColor(Color.RED);
g.fillRect(160, 100, 30, 60);
10.4 Creating Drawings 499
g.setColor(Color.BLACK);
g.drawLine(130, 100, 160, 100);
g.drawLine(130, 160, 160, 160);
```

- Se possível, usar variáveis e 'offsets' para as posições e dimensões

```
g.fillRect(xLeft, yTop, width / 3, width * 2 / 3);
. . .
g.fillRect(xLeft + 2 * width / 3, yTop, width / 3, width * 2 / 3);
. . .
g.drawLine(xLeft + width / 3, yTop, xLeft + width * 2 / 3, yTop);
```

54

## Passos para Desenhar Formas

- 4) Considerar o uso de métodos ou classes para passos repetidos

```
void drawItalianFlag(Graphics g, int xLeft, int yTop, int width)
{
    // Desenhar uma bandeira num dado local e com uma dada dimensão
}
```

- 5) Colocar as instruções de desenho num método  
paintComponent

```
public class ItalianFlagComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        // Instruções de desenho
    }
}
```

Se os desenhos são complexos, optar por chamar os métodos do passo 4

55

## Passos para Desenhar Formas

- 6) Definir a classe de visualização

Fornecer uma classe de visualização, com um método *main* no qual se constrói uma *frame*, adiciona-se o componente e torna-se a *frame* visível

```
public class ItalianFlagViewer
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
        frame.setSize(300, 400);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JComponent component = new ItalianFlagComponent();
        frame.add(component);
        frame.setVisible(true);
    }
}
```

56

## Gestores de Posicionamento

- Distribuem os componentes pelo ecrã
  - Os componentes da GUI são distribuídos através da sua colocação num objeto contentor do package swing:
    - JFrame (*content pane*), JPanel e JApplet
  - Até agora, todos os componentes foram dispostos da esquerda para a direita usando um contentor JPanel



57

## Gestores de Posicionamento

- Cada contentor tem um gestor de posicionamento (*layout manager*) que define o posicionamento dos seus componentes
- Três exemplos de gestores de posicionamento:
  - 1) Border layout
  - 2) Flow layout
  - 3) Grid layout

Os componentes são adicionados a um contentor que usa um gestor de posicionamento para os distribuir no espaço

58

## *Content Pane* da JFrame

- ❑ Uma JFrame tem um *content pane* que é um contendor onde podemos adicionar componentes
  - Usar o método getContentPane para obter a sua referência

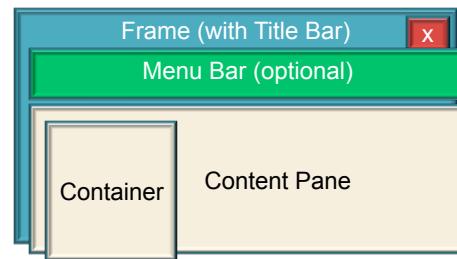
```
Container contentPane = getContentPane();
```

- ❑ Adicionar componentes ao *content pane* ou
  - Adicionar um contendor ao *content pane*, e depois adicionar componentes contendor

59

## *Content Pane* da Jframe

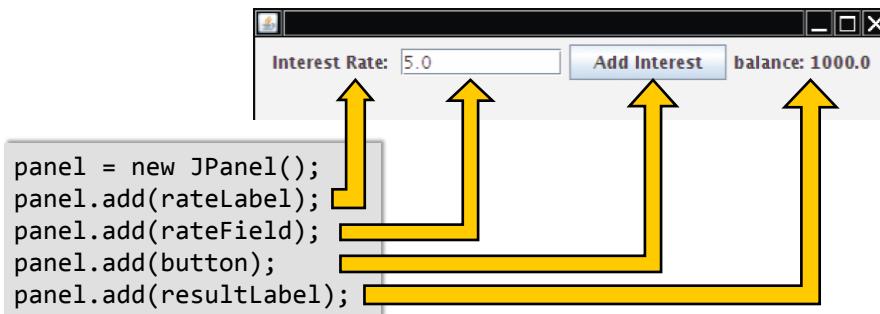
- ❑ Frame
  - Menu Bar
  - Content Pane
    - Componentes
    - Contendor (*container*)
      - Componentes



60

## Flow Layout

- Os componentes são adicionados da esquerda para a direita



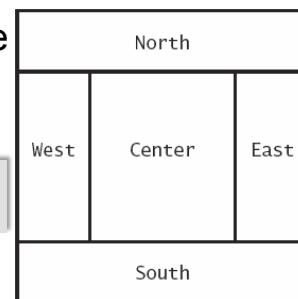
- Um JPanel usa um **flow layout** por defeito

61

## Border Layout

- Os componentes são distribuídos em áreas de um contentor
  - NORTH, EAST, SOUTH, WEST, ou CENTER
  - Especificar uma destas áreas quando se adiciona os componentes
  - O *content pane* de um JFrame usa **border layout** por defeito

```
panel.setLayout(new BorderLayout());
panel.add(component, BorderLayout.NORTH);
```



62

## Grid Layout

- Os componentes são colocados em caixas dispostas sob a forma de uma tabela

- Especificando as dimensões da tabela (linhas e colunas)

```
JPanel buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(4, 3));
```

- Em seguida adicionar os componentes, os quais serão colocados do canto superior esquerdo até ao canto inferior direito

```
buttonPanel.add(button7);
buttonPanel.add(button8);
buttonPanel.add(button9);
buttonPanel.add(button4);
. . .
```

7	8	9
4	5	6
1	2	3
0	.	CE

63

## Utilização de *Nested Panels*

- Criação de *layouts* complexos usando *nested panels*
  - Atribuir a cada painel um *layout manager* apropriado
  - Os painéis têm contornos invisíveis, pelo que podemos usar os painéis necessários para organizar os componentes

```
JTextField in NORTH of keypadPanel
JPanel GridLayout in CENTER of keypadPanel
JPanel keypadPanel = new JPanel();
keypadPanel.setLayout(new BorderLayout());
buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(4, 3));
buttonPanel.add(button7);
buttonPanel.add(button8);
// ...
keypadPanel.add(buttonPanel, BorderLayout.CENTER);
JTextField display = new JTextField();
keypadPanel.add(display, BorderLayout.NORTH);
```

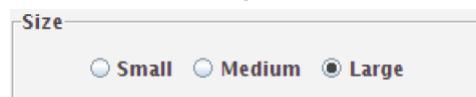
7	8	9
4	5	6
1	2	3
0	.	CE

64

## Componentes de Seleção

- Existem diferentes tipos de componentes para permitir diferentes tipos de seleção:

### Radio Buttons



- Para um conjunto de opções mutuamente exclusivas

### Check Boxes



- Para seleções binárias

### Combo Boxes



- Para um conjunto maior de opções

65

## Painéis de *Radio Buttons*

- Usar um painel para cada conjunto de *radio buttons*

- Por defeito, um JPanel não tem contorno
- É possível adicionar um contorno contendo um texto descriptivo:

```
JPanel panel = new JPanel();
panel.add(smallButton);
panel.add(mediumButton);
panel.add(largeButton);
panel.setBorder(new TitledBorder(new EtchedBorder(),"Size"));
```



- Existem muitos outros estilos de contornos (*borders*)
  - Consultar a documentação do swing

66

## Agrupar Radio Buttons

- ❑ Adicionar Radio Buttons a um ButtonGroup para que apenas um botão possa estar selecionado num dado instante
  - Começar por criar os botões JRadioButtons e depois adicioná-los ao ButtonGroup



```
JRadioButton smallButton = new JRadioButton("Small");
JRadioButton mediumButton = new JRadioButton("Medium");
JRadioButton largeButton = new JRadioButton("Large");
ButtonGroup group = new ButtonGroup();
group.add(smallButton);
group.add(mediumButton);
group.add(largeButton);
```

67

## Selecionar Radio Buttons

- ❑ Quando se usam *radio buttons* é frequente selecionar um dos botões por defeito
  - Usar o método `setSelected`
  - Selecionar o botão antes de tornar a *frame* visível



```
JRadioButton largeButton = new JRadioButton("Large");
largeButton.setSelected(true);
```

- ❑ Chamar o método `isSelected` de cada botão para descobrir qual deles está selecionado

```
if (largeButton.isSelected())
{ size = LARGE_SIZE; }
```

68

## Check Boxes versus Radio Buttons

- Os *radio buttons* e *check boxes* têm aparências distintas

- Os *radio buttons* são redondos e apresentam um ponto preto quando selecionados



- As *check boxes* são quadradas e apresentam uma marca quando selecionadas



69

## Check Boxes

- Uma *check box* é um componente com dois estados: selecionado ou não selecionado
  - Para escolhas que não são mutuamente exclusivas
    - Por exemplo, o estilo do texto pode ser *Italic*, *Bold*, ambos ou nenhum deles
  - Uma vez que o estado de uma *check box* não depende do estado das restantes, não é necessário colocar uma conjunto de *check boxes* no interior de um *button group*



70

## Selecionar Check Boxes

- ❑ Para definir uma *Check Box*, usar a classe `JCheckBox`
  - Passar ao construtor o texto associado à *check box*
- ❑ Chamar o método `isSelected` de uma *check box* para identificar se está selecionada

```
if (italicCheckBox.isSelected())
{ style = style + Font.ITALIC }
```

71

## Combo Boxes

- ❑ Uma *combo box* é uma combinação de uma lista e um campo de texto
  - Usar uma *combo box* com um conjunto grande de alternativas
    - Em situações em que *radio buttons* ocupariam demasiado espaço
  - Pode também ser editável
    - Introduzir uma alternativa numa linha vazia
- Quando a seta à direita é selecionada, é apresentada uma lista a partir da qual se pode selecionar um item da lista



72

## Adição e Seleção de Itens

- Para adicionar um item à lista:

```
JComboBox facenameCombo = new JComboBox();
facenameCombo.addItem("Serif");
facenameCombo.addItem("SansSerif");
. . .
```

- Usar o método `getSelectedItem` para devolver o item selecionado (como um Object)

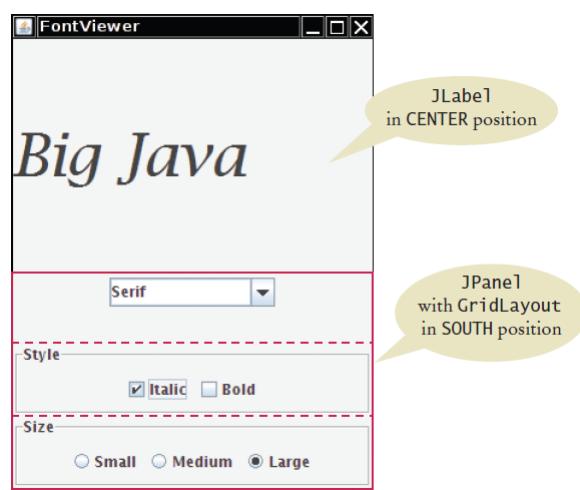
- As *combo boxes* podem armazenar outros objetos para além de strings, sendo por isso necessário efetuar uma conversão:

```
String selectedString = (String)
    facenameCombo.getSelectedItem();
```

73

## Exemplo: FontViewer

- Title Bar
- Label
  - Mostra a fonte selecionada
- Combo Box
- Check Boxes
- Radio Buttons



74

## FontViewer.java (1)

```
1 import javax.swing.JFrame;
2 /**
3  * This program allows the user to view font effects.
4 */
5 public class FontViewer
6 {
7     public static void main(String[] args)
8     {
9         JFrame frame = new FontFrame();
10        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        frame.setTitle("FontViewer");
12        frame.setVisible(true);
13    }
14 }
15 }
```

Instancia um objeto FontFrame

Define o *handler* do evento de encerramento, barra de título e torna a *frame* visível

75

## FontFrame.java (2)

```
35 /**
36  * Constructs the frame.
37 */
38 public FontFrame()
39 {
40     // Construct text sample
41     label = new JLabel("Big Java");
42     add(label, BorderLayout.CENTER);
43
44     // This listener is shared among all components
45     listener = new ChoiceListener();
46
47     createControlPanel();
48     setLabelFont();
49     setSize(FRAME_WIDTH, FRAME_HEIGHT);
50 }
51
52 class ChoiceListener implements ActionListener
53 {
54     public void actionPerformed(ActionEvent event)
55     {
56         setLabelFont();
57     }
58 }
```

Os eventos de todos os componentes da *frame* usam o mesmo *listener*

O método auxiliar *createControlPanel* constrói a GUI

ChoiceListener é uma *inner class* do construtor de FontFrameViewer

76

## FontFrame.java (3)

```
60  /**
61   * Creates the control panel to change the font.
62  */
63  public void createControlPanel()
64  {
65      JPanel facenamePanel = createComboBox();
66      JPanel sizeGroupPanel = createCheckboxes();
67      JPanel styleGroupPanel = createRadioButtons();
68
69      // Line up component panels
70
71      JPanel controlPanel = new JPanel();
72      controlPanel.setLayout(new GridLayout(3, 1));
73      controlPanel.add(facenamePanel);
74      controlPanel.add(sizeGroupPanel);
75      controlPanel.add(styleGroupPanel);
76
77      // Add panels to content pane
78
79      add(controlPanel, BorderLayout.SOUTH);
80  }
```

Utiliza métodos auxiliares para construir cada um dos painéis

Adiciona cada um dos painéis ao controlPanel

77

## Método setLabelFont (1)

```
152 /**
153  * Gets user choice for font name, style, and size
154  * and sets the font of the text sample.
155 */
156 public void setLabelFont()
157 {
158     // Get font name
159     String facename = (String) facenameCombo.getSelectedItem();
160
161     // Get font style    getSelectedItem para a combo box
162
163     int style = 0;
164     if (italicCheckBox.isSelected())
165     {
166         style = style + Font.ITALIC;
167     }
168     if (boldCheckBox.isSelected())
169     {
170         style = style + Font.BOLD;
171     }
172 }
```

Interroga os componentes da frame e define o nome da fonte e o estilo

isSelected para as checkboxes

78

## Method setLabelFont (2)

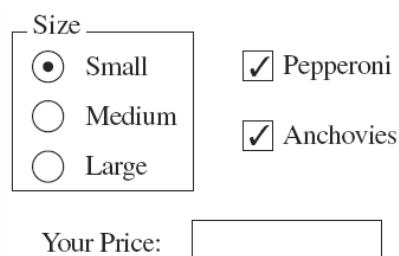
```
173 // Get font size  
174  
175 int size = 0;  
176  
177 final int SMALL_SIZE = 24;  
178 final int MEDIUM_SIZE = 36;  
179 final int LARGE_SIZE = 48; | isSelectedItem para os radio buttons  
180  
181 if (smallButton.isSelected()) { size = SMALL_SIZE; }  
182 else if (mediumButton.isSelected()) { size = MEDIUM_SIZE; }  
183 else if (largeButton.isSelected()) { size = LARGE_SIZE; }  
184  
185 // Set font of text field  
186  
187 label.setFont(new Font(facename, style, size));  
188 label.repaint();  
189 } | Chama setFont com face,  
190 } estilo e tamanho
```

79

## Concepção de uma *User Interface*

### 1) Desenhar um esboço com a disposição dos componentes

- Desenhar todos os botões, *labels*, campos de texto e contornos

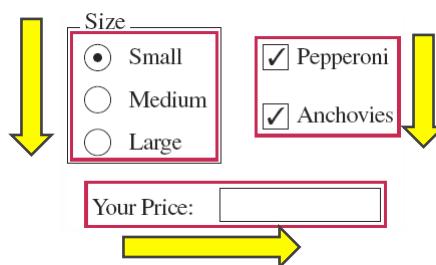


80

## Concepção de uma *User Interface*

2) Identificar agrupamentos de componentes adjacentes que tenham a mesma disposição

- Começar por procurar componentes adjacentes que estejam organizados de baixo para cima ou da esquerda para a direita



81

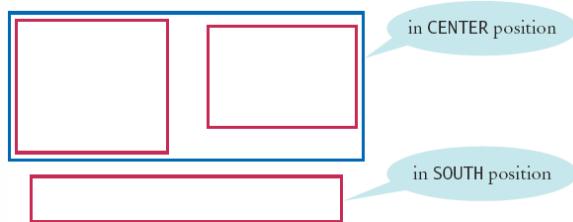
## Concepção de uma *User Interface*

3) Identificar *layouts* para cada um dos grupos

- Para componentes horizontais, usar *Flow Layout*
- Para componentes verticais, usar *Grid Layout* com uma coluna

4) Agrupar os diferentes grupos

- Olhar para cada grupo como uma entidade e agrupar estas entidades em grupos, tal como se agruparam os componentes no passo anterior



82

## Concepção de uma User Interface

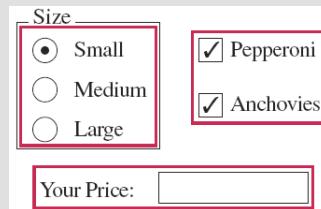
### 5) Escrever o código para gerar o layout

```
JPanel radioButtonPanel = new JPanel();
radioButtonPanel.setLayout(new GridLayout(3, 1));
radioButton.setBorder(new TitledBorder(new EtchedBorder(), "Size"));
radioButtonPanel.add(smallButton);
radioButtonPanel.add(mediumButton);
radioButtonPanel.add(largeButton);

JPanel checkBoxPanel = new JPanel();
checkBoxPanel.setLayout(new GridLayout(2, 1));
checkBoxPanel.add(pepperoniButton());
checkBoxPanel.add(anchoviesButton());

JPanel pricePanel = new JPanel(); // Usa o FlowLayout por defeito
pricePanel.add(new JLabel("Your Price:"));
pricePanel.add(priceTextField);

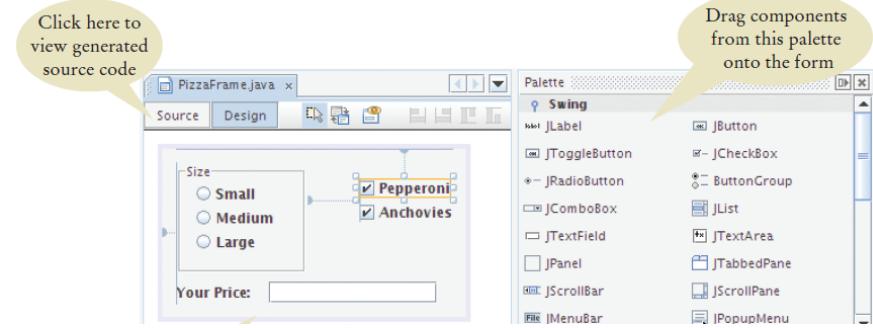
JPanel centerPanel = new JPanel(); // Usa FlowLayout
centerPanel.add(radioButtonPanel);
centerPanel.add(checkBoxPanel); // Frame usa o BorderLayout por defeito
add(centerPanel, BorderLayout.CENTER);
add(pricePanel, BorderLayout.SOUTH);
```



83

## Utilização de um Editor de GUI

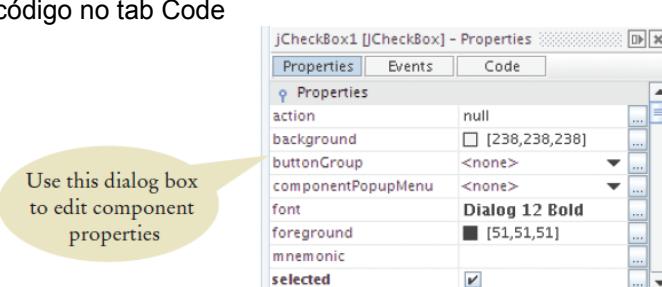
- Um editor de GUI permite arrastar componentes sobre um painel e gera o código automaticamente
- O NetBeans disponibiliza uma destas ferramentas (denominada Matisse)
  - Usa o gestor GroupLayout



84

## Propriedades do editor de GUI

- ❑ É possível configurar as propriedades de cada componente
  - Selecionar um componente no ecrã (Neste exemplo JCheckBox1)
  - Selecionar as propriedades: *color, font, default state ...*
- ❑ É possível definir *event handlers* escolhendo o evento a processar e fornecendo o respectivo código
  - Selecionar um evento no tab Events
  - Escrever o código no tab Code

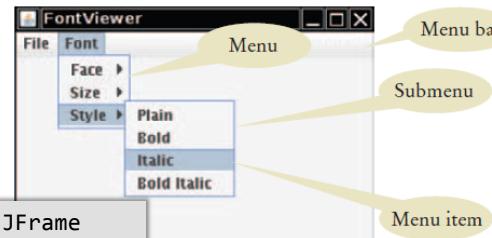


85

## Menus

- ❑ Uma *frame* pode conter uma barra de menu
  - Os itens do menu podem ser adicionados a cada menu ou submenu

```
public class MyFrame extends JFrame
{
    public MyFrame()
    {
        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);
        ...
    }
    ...
}
```



Instancia uma barra de menu e depois adiciona-a à *frame* com o método *setJMenuBar*

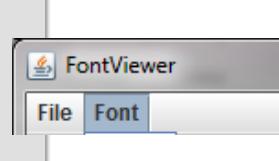
86

## MenuBar e Itens de Menu

- AMenuBar contém Menus

- O contentor dos itens do *top-level menu* é denominado de MenuBar
    - Adiciona objetos JMenu ao MenuBar

```
JMenuBar menuBar = new JMenuBar();
JMenu fileMenu = new JMenu("File");
JMenu fontMenu = new JMenu("Font");
menuBar.add(fileMenu);
menuBar.add(fontMenu);
```



- Um Menu contém SubMenus e itens de Menu

- Um item de Menu não contém SubMenus
    - Os itens de Menu e os SubMenus são adicionados com o método add

87

## Eventos dos Itens de Menu

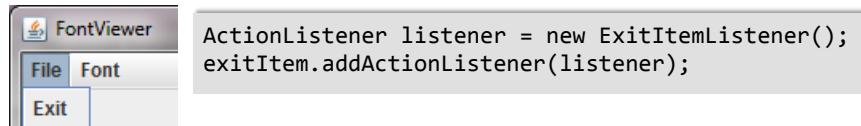
- Os itens de Menu geram eventos quando selecionados

- Adicionar *action listeners* apenas aos itens de menu
    - E não aos menus ou à barra de menu
    - Quando o utilizador clica sobre o nome de um menu e um submenu abre, nenhum evento é gerado

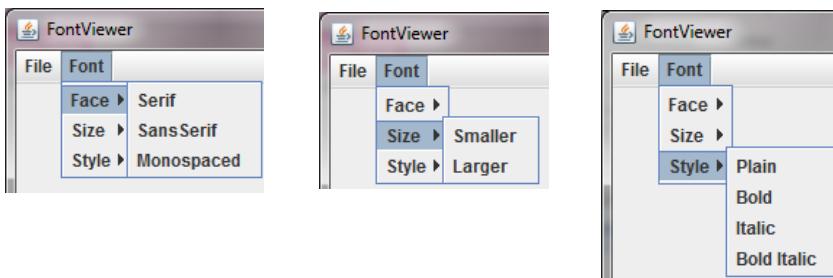
88

## Eventos dos Itens de Menu

- Adicionar *action listeners* a cada item de menu



- O *listener* pode ser personalizado para cada item de menu



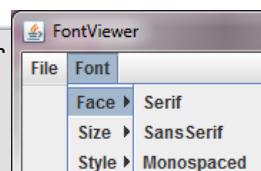
89

## Exemplo de *Listener* para um Item de Menu

- Tarefa do ActionListener

- Toma um parâmetro do tipo *String* (nome da *font face*)
  1. Usa o texto no menu item para definir o novo valor do *face name*
  2. Define uma nova *font* a partir dos valores atuais de *face*, *size*, e *style*, e aplica-los ao *label*

```
class FaceItemListener implements ActionListener
{
    private String name;
    public FaceItemListener(String newName)
    { name = newName; }
    public void actionPerformed(ActionEvent event)
    {
        faceName = name; // Variável de instância da classe frame
        setLabelFont();
    }
}
```



90

## ActionListener (1)

- ❑ Regista o *listener object* com o nome apropriado:

```
public JMenuItem createFaceItem(String name)
{
    JMenuItem item = new JMenuItem(name);
    ActionListener listener = new FaceItemListener(name);
    item.addActionListener(listener);
    return item;
}
```

- ❑ Pode ser melhorado através do uso de uma *inner class* local

- Se deslocarmos a declaração da *inner class* para o interior do método `createFaceItem`, o método `actionPerformed` pode aceder diretamente à variável `name` (em vez de passá-la por parâmetro)

91

## ActionListener (2)

- ❑ Versão *Inner Class* do *Listener*

```
public JMenuItem createFaceItem(final String name)
// Variáveis final podem ser acedidas a partir de um método de
// uma inner class
{
    class FaceItemListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            facename = name; // Accesses the local variable name
            setLabelFont();
        }
    }
    JMenuItem item = new JMenuItem(name);
    ActionListener listener = new FaceItemListener();
    item.addActionListener(listener);
    return item;
}
```

92

## FontViewer2.java - Menu

```
25  /**
26   * Constructs the frame.
27  */
28  public FontFrame2()
29  {
30      // Construct text sample
31      label = new JLabel("Big Java");
32      add(label, BorderLayout.CENTER);
33
34      // Construct menu
35      JMenuBar menuBar = new JMenuBar();
36      setJMenuBar(menuBar);
37      menuBar.add(createFileMenu());
38      menuBar.add(createFontMenu());
39
40      facename = "Serif";
41      fontsize = 24;
42      fontstyle = Font.PLAIN;
43
44      setLabelFont();
45      setSize(FRAME_WIDTH, FRAME_HEIGHT);
46 }
```



Cria a barra de menu top level e adiciona-la à frame usando o método setJMenuBar

Cria os menus File e Font usando métodos auxiliares

93

## FontViewer2.java – File Menu

```
48  class ExitItemListener implements ActionListener
49  {
50      public void actionPerformed(ActionEvent event)
51      {
52          System.exit(0);
53      }
54 }
```

A inner class da frame trata do evento File Menu Exit

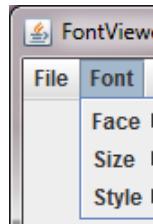
```
60  public JMenu createFileMenu()
61  {
62      JMenu menu = new JMenu("File");
63      JMenuItem exitItem = new JMenuItem("Exit");
64      ActionListener listener = new ExitItemListener();
65      exitItem.addActionListener(listener);
66      menu.add(exitItem);
67      return menu;
68 }
```

Cria o menu File, adiciona o item de menu Exit, instancia a inner class ExitItemListener, regista o objeto e adiciona exitItem ao menu

94

## FontViewer2.java – Submenus

```
74     public JMenu createFontMenu()
75     {
76         JMenu menu = new JMenu("Font");
77         menu.add(createFaceMenu());
78         menu.add(createSizeMenu());
79         menu.add(createStyleMenu());
80         return menu;
81     }
```



```
87     public JMenu createFaceMenu()
88     {
89         JMenu menu = new JMenu("Face");
90         menu.add(createFaceItem("Serif"));
91         menu.add(createFaceItem("SansSerif"));
92     }
93     public JMenu createSizeMenu()
94     {
95         JMenu menu = new JMenu("Size");
96         menu.add(createSizeItem("Smaller", -1));
97         menu.add(createSizeItem("Larger", 1));
98     }
99     public JMenu createStyleMenu()
100    {
101        JMenu menu = new JMenu("Style");
102        menu.add(createStyleItem("Plain", Font.PLAIN));
103        menu.add(createStyleItem("Bold", Font.BOLD));
104    }
105}
106
107
108
109
110
111
112
113
114
115
116
```

Cria o menu Font e adiciona os submenus usando métodos auxiliares

Submenu Font Face

Submenu Font Size

Submenu Font Style

95

## FontViewer2.java – Listeners

```
128     public JMenuItem createFaceItem(final String name)
129     {
130         class FaceItemListener implements ActionListener
131         {
132             public void actionPerformed(ActionEvent event)
133             {
134                 facename = name;
135                 setLabelFont();
136             }
137         }
138         public JMenuItem createSizeItem(String name, final int increment)
139         {
140             class SizeItemListener implements ActionListener
141             {
142                 public void actionPerformed(ActionEvent event)
143                 {
144                     fontsize = fontsize + increment;
145                     setLabelFont();
146                 }
147             }
148             JMenuItem item = new JMenuItem(name);
149             ActionListener listener = new SizeItemListener();
150             item.addActionListener(listener);
151             return item;
152         }
153     }
154 }
```

Inner class listener

Cada item de menu trata dos seus próprios eventos

96

## Timer Events para Animação

- Os *timer events* podem ser usados para implementar animações simples
- O package `javax.swing` disponibiliza a classe `Timer`

Um *timer* notifica um listener a intervalos contantes



97

## Timer Events para Animação

- Classe `javax.swing.Timer`
  - Pode gerar uma série de eventos de forma periódica
  - Especificar a frequência dos eventos e um objeto de uma classe que implementa a interface `ActionListener` interface

```
class MyListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        // Listener action (executada a cada timer event)
    }
}
MyListener listener = new MyListener();
Timer t = new Timer(interval, listener);
t.start();
```

Os *Timer events* começarão a ser gerados após a chamada do método `start`

98

## Exemplo: Retângulo Animado

```
8  public class RectangleComponent extends JComponent
9  {
10     private static final int RECTANGLE_WIDTH = 20;
11     private static final int RECTANGLE_HEIGHT = 30;
12
13     private int xLeft;
14     private int yTop;
15
16     public RectangleComponent()
17     {
18         xLeft = 0;
19         yTop = 0;
20     }
21
22     public void paintComponent(Graphics g)
23     {
24         g.fillRect(xLeft, yTop, RECTANGLE_WIDTH, RECTANGLE_HEIGHT);
25     }
26
27     public void moveRectangleBy(int dx, int dy)
28     {
29         xLeft = xLeft + dx;
30         yTop = yTop + dy;
31         repaint();
32     }
33 }
```

O método repaint chama o método paintComponent

99

## RectangleFrame.java

```
9  public class RectangleFrame extends JFrame
10 {
11     private static final int FRAME_WIDTH = 300;
12     private static final int FRAME_HEIGHT = 400;
13
14     private RectangleComponent scene;
15
16     class TimerListener implements ActionListener
17     {
18         public void actionPerformed(ActionEvent event)
19         {
20             scene.moveRectangleBy(1, 1);
21         }
22     }
23
24     public RectangleFrame()
25     {
26         scene = new RectangleComponent();
27         add(scene);
28
29         setSize(FRAME_WIDTH, FRAME_HEIGHT);
30
31         ActionListener listener = new TimerListener();
32
33         final int DELAY = 100; // Milliseconds between timer ticks
34         Timer t = new Timer(DELAY, listener);
35         t.start();
36     }
37 }
```

A inner class TimerListener move o retângulo

O objeto da classe TimerListener é chamado a cada pelo objeto Timer

100

## repaint e paintComponent

- Quando se altera as propriedades de um componente, este não é desenhado automaticamente de forma a refletir as alterações
- É necessário chamar o método repaint do componente, tanto num *event handler* ou num dos métodos do componente responsável pelas alterações
- O método paintComponent do componente será então invocado com um objeto Graphics apropriado

Não chamar o método paintComponent diretamente

101

## Mouse Events

- Os *mouse events* são mais complexos do que os eventos gerados por botões ou por timers
- Um *mouse listener* deve implementar todos os cinco métodos da interface MouseListener

Método	Origem do evento
mousePressed	Um botão do rato foi pressionado sobre um componente
mouseReleased	Um botão do rato foi libertado sobre um componente
mouseClicked	The mouse has been clicked on a component
mouseEntered	The mouse enters a component
mouseExited	The mouse exits a component

- Os métodos mousePressed e mouseReleased são chamados sempre que o botão do rato é pressionado ou libertado

102

## Métodos MouseListener

- ❑ Se um botão é pressionado e libertado numa sequência rápida, e o rato não se moveu, então o método mouseClicked também é chamado
- ❑ Os métodos mouseEntered e mouseExited podem ser usados para desenhar um componente da GUI sempre que o rato está a apontar para o seu interior

```
public interface MouseListener
{
    void mousePressed(MouseEvent event);
    void mouseReleased(MouseEvent event);
    void mouseClicked(MouseEvent event);
    void mouseEntered(MouseEvent event);
    void mouseExited(MouseEvent event);
}
```

103

## Implementação de um MouseListener

- ❑ Outra diferença: para o adicionar a um componente, utiliza-se o método `addMouseListener` em vez de `addActionListener`:

```
public class MyMouseListener implements MouseListener
{
    // Implements five methods
}
MouseListener listener = new MyMouseListener();
component.addMouseListener(listener);
```

- ❑ No exemplo que se segue, o utilizador clica num componente contendo um retângulo; sempre que o botão do rato é pressionado, o retângulo é deslocado para a posição do rato

104

## RectangleComponent2.java

```
8  public class RectangleComponent2 extends JComponent
9  {
10     private static final int RECTANGLE_WIDTH = 20;
11     private static final int RECTANGLE_HEIGHT = 30;
12
13     private int xLeft;
14     private int yTop;
15
16     public RectangleComponent2()
17     {
18         xLeft = 0;
19         yTop = 0;
20     }
21
22     public void paintComponent(Graphics g)
23     {
24         g.fillRect(xLeft, yTop, RECTANGLE_WIDTH, RECTANGLE_HEIGHT);
25     }
26
27     public void moveRectangleTo(int x, int y)
28     {
29         xLeft = x;
30         yTop = y;
31         repaint();
32     }
33 }
```

❑ O método `moveRectangleTo` modificado recebe a posição do rato  
❑ É necessário um *mouse listener* para chamá-lo

O método `repaint` chama o método `paintComponent`

105

## Métodos de MouseListener

- ❑ `mousePressed` é o único evento que pretendemos tratar (os métodos para tratar os restantes eventos ficam vazios):

```
15    class MousePressListener implements MouseListener
16    {
17        public void mousePressed(MouseEvent event)
18        {
19            int x = event.getX();
20            int y = event.getY();
21            scene.moveRectangleTo(x, y);
22        }
23
24        // Do-nothing methods
25        public void mouseReleased(MouseEvent event) {}
26        public void mouseClicked(MouseEvent event) {}
27        public void mouseEntered(MouseEvent event) {}
28        public void mouseExited(MouseEvent event) {}
29    }
30
```

Alguns métodos úteis do objeto `event`

106

## RectangleFrame2.java (cont.)

```
8  public class RectangleFrame2 extends JFrame
9  {
10     private static final int FRAME_WIDTH = 300;
11     private static final int FRAME_HEIGHT = 400;
12
13     private RectangleComponent2 scene;
14
15     class MousePressListener implements MouseListener
16     {
17         public void mousePressed(MouseEvent event)
18         {
19             int x = event.getX();
20             int y = event.getY();
21             scene.moveRectangleTo(x, y);
22         }
23
24         // Do-nothing
25         public void mouseClicked(MouseEvent event)
26         {
27             public void mouseEntered(MouseEvent event)
28             {
29             }
30         }
31         public RectangleFrame2()
32         {
33             scene = new RectangleComponent2();
34             add(scene);
35
36             MouseListener listener = new MousePressListener();
37             scene.addMouseListener(listener);
38
39             setSize(FRAME_WIDTH, FRAME_HEIGHT);
40         }
41     }
}
```

Scene é a referência do objeto da classe RectangleComponent2

107

## Resumo: *Frames e Componentes*

- ❑ Para mostrar uma *frame*, construir um objeto `JFrame`, definir as suas dimensões e torná-la visível
- ❑ Usar um `JPanel` para agrupar vários componentes da interface com o utilizador
- ❑ Declarar uma subclasse `Jframe`, definindo uma *frame* mais complexa

108

## Resumo: Eventos e *Handlers*

- ❑ Os eventos da GUI incluem atuação de teclas e de botões do rato, movimentos do rato, seleção de menus, ...
- ❑ Um *event listener* pertence a uma classes criada pelo programador
  - Its methods describe the actions to be taken when an event occurs.
  - Event sources report on events. When an event occurs, the event source notifies all event listeners.
- ❑ Adicionar um *ActionListener* a cada botão para que o programa reaja à ativação do rato
- ❑ Os métodos de uma *inner class* pode aceder a variáveis da classe envolvente

109

## Resumo: *TextFields* e *TextAreas*

- ❑ Usar componentes *JTextField* para criar espaços para entradas do utilizador
  - Colocar um *JLabel* próximo de cada campo de texto
- ❑ Usar um *JTextArea* para mostrar várias linhas de texto
- ❑ É possível juntar *scroll bars* a qualquer componente com um *JScrollPane*

110

## Resumo: Formas (gráficos) Simples

- ❑ Para mostrar um desenho, declarar uma classe que estenda a classe `JComponent`
- ❑ Inserir instruções de desenho no método `paintComponent`
  - Este método é chamada sempre que o componente necessita ser redesenrado
- ❑ A classe `Graphics` tem métodos para desenhar primitivas de desenho (retângulos, ...)
  - Usar `drawRect`, `drawOval` e `drawLine` para desenhar formas geométricas
  - O método `drawString` desenha uma string, começando no seu *basepoint*.

111

## Resumo: Cor e repaint

- ❑ Quando se define uma nova cor no contexto gráfico, esta cor é usada em todas as operações de desenho subsequentes
- ❑ Chamar o método `repaint` sempre que o estado dos componentes se altere
- ❑ Quando se coloca um componente desenhado num painel (*panel*), é necessário especificar a sua dimensão

112

## Resumo: Contentores e *Layouts*

- ❑ Os componentes da GUI são organizados no interior de contentores
  - Os contentores podem ser colocados no interior de contentores maiores
  - Cada contentor tem um gestor de *layout* que gere a disposição dos seus componentes
  - Três dos gestores de *layout* são: *border layout*, *flow layout* e *grid layout*
  - Quando se adiciona um componente a um contentor com o *border layout*, especificar a posição NORTH, EAST, SOUTH, WEST ou CENTER
- ❑ O *content pane* de uma *frame* tem por defeito um *border layout*
- ❑ Um JPanel tem por defeito um *flow layout*

113

## Resumo: Componentes Swing

- ❑ Para um pequeno conjunto de escolhas mutuamente exclusivas, usar um grupo de *radio buttons* ou uma *combo box*
- ❑ Adicionar os *radio buttons* a um *ButtonGroup* de forma a que apenas um dos botões possa estar selecionado
- ❑ Normalmente usa-se um contorno para delimitar visualmente o grupo
- ❑ Para uma escolha binário, usar uma *check box*
- ❑ Para um conjunto mais alargado de escolhas, usar uma *combo box*
- ❑ *Radio buttons*, *check boxes* e *combo boxes* geram *action events*, tal como os botões

114

## Resumo: Menus Swing

- ❑ Uma *frame* pode conter uma barra de menu
  - A barra de menu contém menus
    - Um menu contém submenus e itens de menu
- ❑ Itens de menu geram *action events*
- ❑ Consultar a documentação da API para obter informação sobre outros para construção de GUI

115

## Resumo: *Timers, Eventos*

- ❑ *Timers* e Animação
  - Um *timer* gera *action events* a intervalos fixos
  - Para criar uma animação, o *timer listener* deve actualizar e *repaint* um componente várias vezes por segundo
- ❑ *Mouse Events*
  - Utiliza-se um *mouse listener* para capturar *mouse events*

116