Programação de GUI, tratamento de erros, empacotamento e publicação

Tiago T. Wirtti

24 de setembro de 2014

Resumo

Neste texto veremos alguns conceitos fundamentais para a programação de aplicativos Java. Entre eles destacamos a correta utilização de construtores, criação de GUI, tratamento de erros, organização em pacotes, criação de uma arquivo "executável"do formato .jar, entre outras coisas.

1 Construtores

Nesta seção vamos conhecer com mais detalhes um importante conceito de construtor [1], que se aplica ao Java e outras linguagem de programação. Observando mais atentamente a classe hipotética `AreaDesenho` (Figura 1), percebemos a existência de algo que parece (mais não é) um “método”. Ele possui o próprio nome da classe e não tem tipo de retorno. Na verdade não se trata de um método, mas de um construtor da classe `AreaDesenho`! Então, temos a primeira dica: construtores não são métodos, são construtores ... Sobre o construtor em questão, vamos fazer algumas considerações:

- É publico (poderia ser privado, protegido ou padrão).
- Não possui parâmetros.
- Invoca o construtor sem argumentos da sua superclasse diretamente através da chamada `super();`
- Chama um método de classe denominado `inicialize()`.

Uma classe pode ter qualquer quantidade de construtores. Então, podemos fornecer um novo construtor para a classe `AreaDesenho` (Figura 1), desta vez com os parâmetros [2] `largura` e `altura`.

Algumas regras básicas para a criação de construtores:

- São chamados de forma encadeada (da classe sendo instanciada para as superclasses), assim, o primeiro construtor a ser executado é sempre o de Object!

- Não são métodos, logo não podem ser herdados.
- Podem usar qualquer modificador de acesso e não possuem um tipo de retorno.
- Devem ter o mesmo nome da classe.
- Um construtor padrão é fornecido pelo compilador se, e somente se, não for fornecido pelo programador.
- O construtor padrão será sempre um construtor sem argumentos.
- A primeira instrução em um construtor deve ser (1) a chamada a um construtor da mesma classe, usando `this(parâmetros)`; ou (2) a chamada ao construtor da superclasse, usando `super(parâmetros)`.

A seguir (Figuras 2 e 3) temos, respectivamente, as classes `Pessoa` e `Funcionario`. Observe que entre elas há uma relação de herança. Essas duas classes exemplificam o uso de construtores encadeados. Use o programa da Figura 4 para observar o encadeamento de construtores.

```
public AreaDesenho() {
      super();
      initialize();
}

public AreaDesenho(int largura, int altura){
      super();
      this.setSize(largura, altura);
      this.setLayout(null);
      this.addMouseListener(this);
}
```

Figura 1: Exemplo de construtor com e sem argumentos.

2 GUI em Java

O pacotes mais básico para criação de interfaces gráficas em Java é o `java.awt`, onde *awt* significa *abstract windowing toolkit*. O pacote o `java.awt` é um conjunto de classes que oferece componentes para a construção de GUIs em Java. Toda a informação necessária pode ser encontrada na documentação padrão da API Java [3]. A seguir (Figura 5) temos um esquema simplificado da hierarquia de classes/interfaces do pacote `java.awt`.

Uma desvantagem o pacote `java.awt` é a sua dependência do sistema gráfico do sistema operacional. Desta forma não é possível alterar a aparência *(look-and-feel)* [4] do sistema de janelas. Como Java é multiplataforma, surgiu a necessidade de se criar um novo pacote gráfico sobre `java.awt`, que pudesse entregar gráficos de melhor qualidade e independentes do sistema gráfico nativo

```
class Pessoa{
	String nome;
	Pessoa(){
		// chama o construtor Object() implicitamente
		System.out.println("Executou Pessoa()");
	}
	Pessoa(String nome){
		// chama o construtor Individuo()
		this();
		this.nome = nome;
		System.out.println("Executou Pessoa(String nome)");
	}
}
```

Figura 2: Exemplo de construtor com e sem argumentos.

```
class Funcionario extends Pessoa{
	String empresa, cargo, matricula;
	Funcionario(){
		// chama o construtor Individuo() implicitamente
		System.out.println("Executou Funcionario()");
	}
	Funcionario(String nome, String empre){
		// chama o construtor Individuo(com 1 argumento)
		// explicitamente
		super(nome);
		this.empresa = empre;
		System.out.println("Executou Funcionario(String nome,
				String empre)");
	}
	Funcionario(String nome, String empre, String cargo){
		// chama o construtor Funcionario(com 2 argumentos)
		// explicitamente
		this(nome, empre);
		this.cargo = cargo;
		System.out.println("Executou Funcionario(String nome,
				String empre, String cargo)");
	}
	Funcionario(String nome, String empre, String cargo,
			String matricula){
		// chama o construtor Funcionario(com 3 argumentos)
		// explicitamente
		this(nome, empre, cargo);
		this.matricula = matricula;
		System.out.println("Executou Funcionario(String nome,
				String empre, String cargo,
				String matricula)");
	}
}
```

Figura 3: Exemplo de construtor com e sem argumentos.

```
public class TesteConstrutores{
      public static void main(String args[]){

            System.out.println("======================");
            System.out.println("Instancia objeto Pessoa");
            Pessoa p2 = new Pessoa("Joaquim");

            System.out.println("\n======================");
            System.out.println("Instancia objeto Funcionario");
            Funcionario f2 = new Funcionario("Roberto", "Sun",
                                             "Programmer", "123456");
      }
}
```

Figura 4: Exemplo de construtor com e sem argumentos.

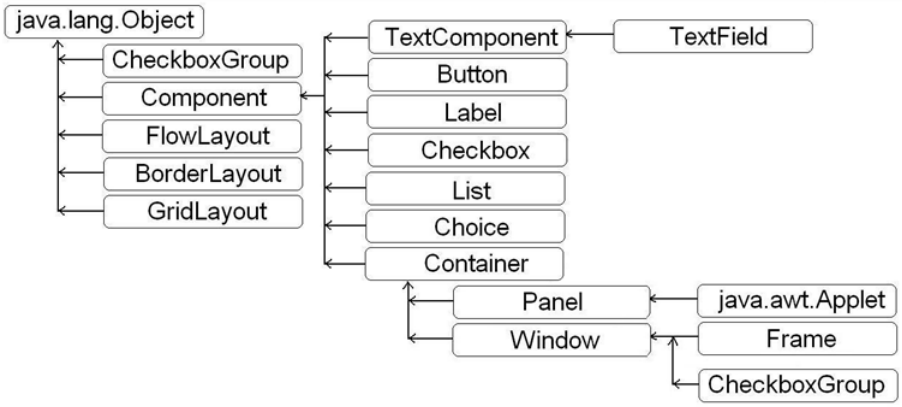

Figura 5: Esquema simplificado do pacote `java.awt`.

do sistema operacional. Daí surgiu o pacote `javax.swing`, que é um conjunto de classes que oferecem componentes para a construção de GUIs sobre o pacote `java.awt` (por exemplo, `javax.swing.JFrame` estende de `java.awt.Frame`, conforme ilustra a Figura 6), permitindo maior flexibilidade e configuração da aparência *(look-and-feel)* conforme a necessidade do aplicativo gráfico, além de maior riqueza gráfica [3].

Um conceito muito importante na criação de GUIs em Java é o *layout* [5]. O *layout* é uma forma de você distribuir os componentes gráficos na tela. Há vários tipos de *layouts*. Para conhecê-los em detalhes, visite o tutorial *A Visual Guide to Layout Managers* [6].

Com os conceitos adquiridos até aqui (espero que você tenha visitado as referências), faremos, a seguir, um exemplo de criação de GUI em Java.

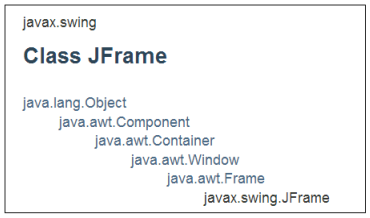

Figura 6: Hierarquia de `JFrame` no pacote `javax.swing`[3].

3 Exemplo de criação de uma GUI em Java

Ao criar uma GUI, o primeiro passo é, certamente, idealizá-la (especificá-la, desenhá-la e etc) em papel ou em outro meio. Assumiremos que esse trabalho já foi feito e o resultado deste "desenho"é mostrado na Figura

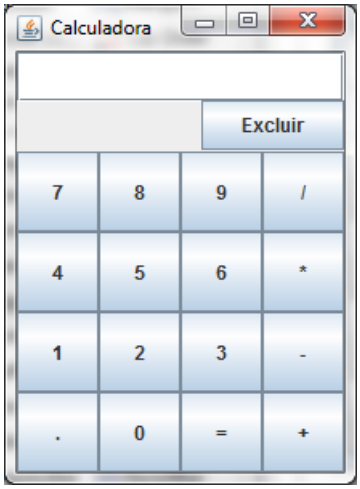

Figura 7: Hierarquia de `JFrame` no pacote `javax.swing`[3].

Utilizaremos o IDE Eclipse, versão Kepler, com o plug-in WindowBuilder, que pode ser obtido no site do `www.eclipse.org`, em downloads, selecionando o *hiperlink Eclipse IDE for Java Developers*.

Veja a seguir os passos para a criação da GUI da Figura 7:

- Criar um projeto Java comum (`File > New > Java Project`).
- Criar uma classe de interface gráfica. Como o projeto selecionado, pressionar o botão direito do mouse e escolher `New > Other ...` (ou pressionar `<CTRL + N>`), e, na janela seguinte, selecionar `WindowBuilder > Swing Designer > Application Window`. Esse comando cria uma classe que é uma janela gráfica e, ao mesmo tempo, o ponto de entrada de uma aplicação.

- Redimensionar a janela para as dimensões 220 x 300 *pixels* (propriedade `size`) e renomeie a aplicação para "Calculadora"(propriedade `title`). O redimensionamento da janela pode ser feito também arrastando-se uma das bordas da mesma (preferível). A esta altura, você já tem uma GUI que pode ser executada. Execute o programa pela primeira vez! Observe que a janela não está posicionada no centro da tela (área de trabalho). Se quiser alterar isso para que a posição seja sempre central, altere o método `initialize()` conforme sinalizado na Figura 8.

```
private void initialize() {
        frmCalculadora = new JFrame();
        frmCalculadora.setBounds(new Rectangle(0, 0, 220, 300));
        frmCalculadora.setTitle("Calculadora");
        frmCalculadora.setBounds(100, 100, 220, 300);
        frmCalculadora.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frmCalculadora.setLocationRelativeTo(null); // centraliza janela
}
```

Figura 8: Centralizando a tela no *desktop*.

- O próximo passo deve atingir o resultado da Figura 9. Para isso, você deve (1) inserir um `JPanel` na área central do `contentPane` e mudar o *layout* deste painel (nomeado `panel`) para `Absolute Layout` (Figura 10); (2) Inserir um campo texto na posição A; (3) Inserir um `JPanel` nomeado `panel_Excluir` em B e definir o seu *layout* como `Absolute`; e (4) inserir um `JPanel` nomeado `panel_Teclado` com *layout* do tipo `GridLayout` com 4 linhas por 4 colunas.

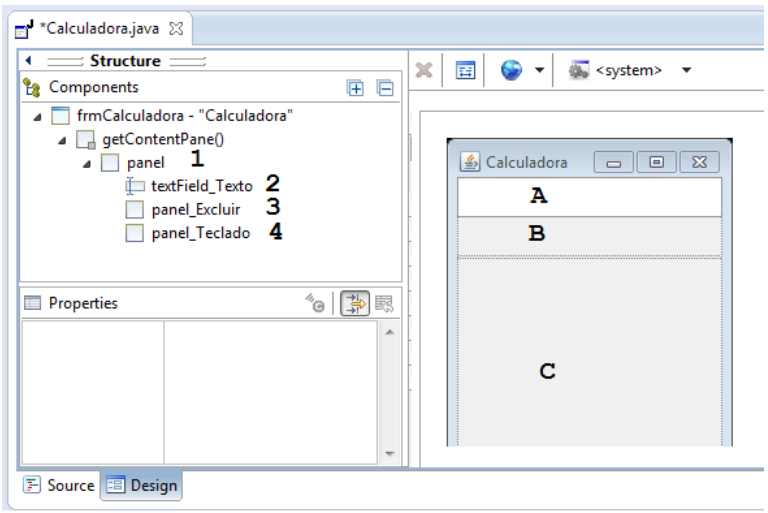

Figura 9: Calculadora após a implementação dos *layouts*.

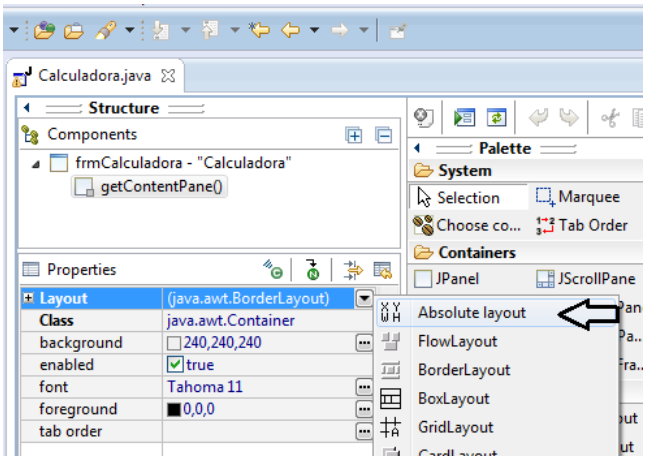

Figura 10: Usando o *layout* absoluto.

- Montar o teclado. O teclado é formado pro vários `JButton` posicionados em grade. O resultado esperado é mostrado na Figura 11, a seguir.

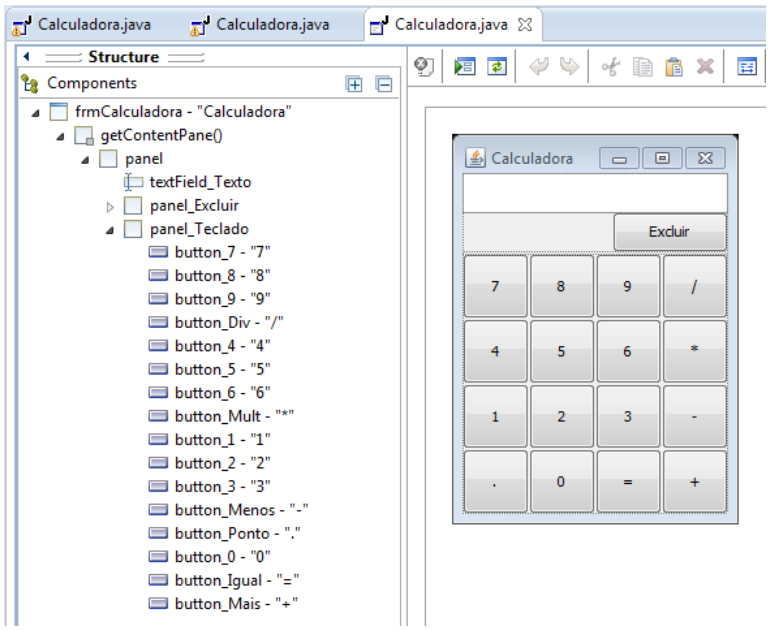

Figura 11: Calculadora com teclado.

Por enquanto, esta GUI não responde a qualquer ação do usuário. Para que isso ocorra, temos que programar respostas às ações do usuário, ou seja, programar os eventos!

4 Programação orientada a eventos

A programação orientada a eventos é um paradigma de programação. Diferente de programas tradicionais (de antigamente, tais com programas Cobol, Clipper e outros) que seguem um fluxo de controle padronizado, o controle de fluxo de programas orientados a evento é guiado por indicações externas, ou seja, *ações* que são capturadas e associadas a respostas pré-programadas, denominadas *evento*. Sem esse recurso, uma GUI não pode responder a ações aleatórias realizadas pelo usuário. Ao invés de aguardar por um comando completo que processa a informação, o sistema orientado a eventos é programado em um laço de observação de eventos, que recebe repetidamente informação para processar e disparar uma função de resposta de acordo com a ação de entrada.

O método pelo qual a informação é adquirida por camadas mais baixas do sistema é irrelevante. As entradas podem ser enfileiradas ou uma interrupção pode ser registrada para reagir a uma ação específica, ou ainda ambos. Programas orientados a evento geralmente consistem em vários pequenos tratadores (programas que processam os eventos para produzir respostas), e um disparador, que invoca os pequenos tratadores.

Esse método é bastante flexível e permite um sistema assíncrono (que atende a ações em qualquer tempo). Programas com interface com o usuário geralmente utilizam tal paradigma. Sistemas operacionais também são outro exemplo de programas que utilizam programação orientada a eventos, este em dois níveis[1]. Em um ambiente com GUI, alguns exemplos de eventos são:

- Pressionar <ENTER> sobre um botão selecionado;
- Clicar em um item de uma lista;
- Abrir, minimizar, maximizar, restaurar ou fechar uma janela;
- Navegar entre campos de texto editáveis de um formulário e etc.

Na linguagem de programação Java (e em outras linguagens OO), eventos são tratados como objetos. Estes eventos são capturados por objetos conhecidos com *listeners*, ou "observadores de eventos". Tais eventos capturados são tratados por objetos conhecidos como *handlers*, ou "manipuladores de eventos". Os observadores de eventos são representados pela interface `java.util.EventListener` (Figura)12). Os tratadores de eventos são representados pela `java.util.EventObject` (Figura 13).

Antes de programar o tratamento de eventos da calculadora, vale ressaltar que o *framework* WindowBuilder incentiva a utilização de classes anônimas para tratamento de eventos. Essa abordagem, embora muito utilizada, fragmenta fortemente o código de tratamento de eventos. Por isso utilizaremos outra abordagem: agrupar a funcionalidade de tratamento de eventos nos métodos especialis-

[1] No nível mais baixo encontram-se o tratamento de interrupções como tratadores de eventos de hardware, com a CPU realizando o papel de disparador. No nível mais alto encontram-se os processos sendo disparados novamente pelo sistema operacional.

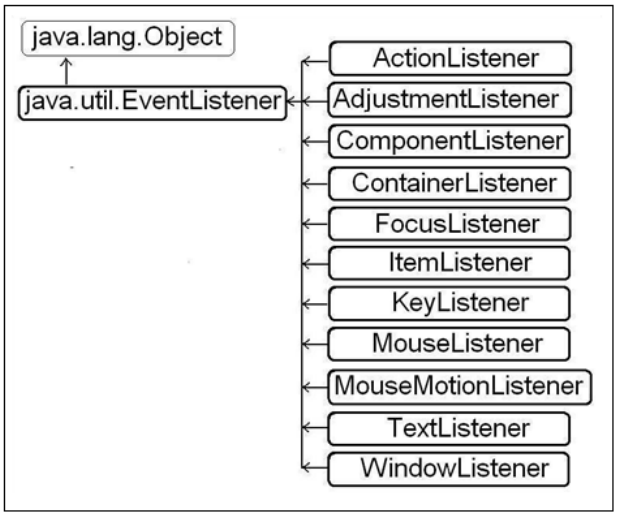

Figura 12: Hierarquia de observadores de eventos.

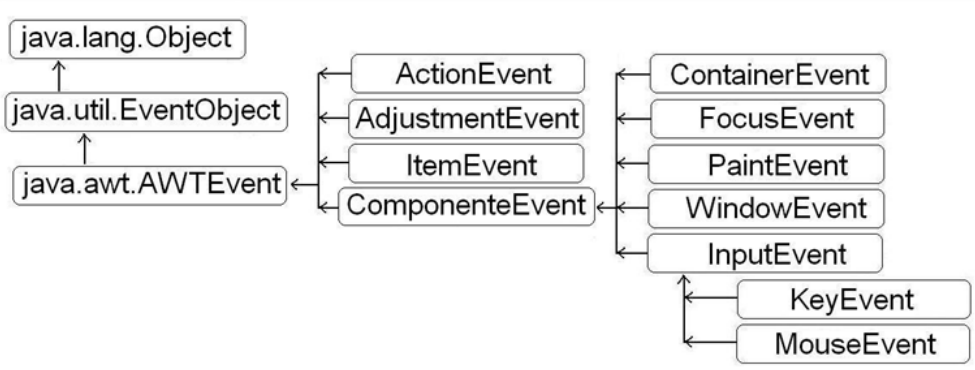

Figura 13: Hierarquia de tratadores de eventos.

tas. Para fazer isso, precisamos adaptar o código gerado automaticamente. Siga os passos:

- Transformar todas as variáveis locais de `initialize()` em atributos privados da classe `Calculadora`. Não se esqueça de eliminar as declarações locais no `initialize()` e manter apenas a atribuições feitas às variáveis.
- Fazer com que `Calculadora` estenda `JFrame`. O resultado é observado na Figura 14.

O tratamento de eventos propriamente dito, consiste em:

```
public class Calculadora extends JFrame {
     private JFrame frmCalculadora;
     private JTextField textField_Texto;
     private JPanel panel;
     private JPanel panel_Excluir;
     private JButton button_Excluir;
     private JPanel panel_Teclado;
     private JButton button_7;
     private JButton button_8;
     private JButton button_9;
     private JButton button_Div;
     private JButton button_4;
     private JButton button_5;
     private JButton button_6;
     private JButton button_Mult;
     private JButton button_1;
     private JButton button_2;
     private JButton button_3;
     private JButton button_Menos;
     private JButton button_Ponto;
     private JButton button_0;
     private JButton button_Igual;
     private JButton button_Mais;
     // continua ...
```

Figura 14: Adaptação no código da calculadora para usar tratamento de eventos centralizado.

- Implementar a interface de observação apropriada, por exemplo, `ActionListener` (para eventos de ação), `KeyListener` (para eventos de captura de teclado), e etc (Figura 15).

```
public class Calculadora extends JFrame
                         implements ActionListener,
                                    KeyListener{
      // atributos ...
```

Figura 15: Implementação das interfaces `ActionListener` e `KeyListener`.

- Adicionar o observador de eventos ao componente que se pretende observar. No caso de um evento de ação, deve-se fazer a chamada do método `addActionListener()` sobre o botão (Figura 16). No caso do campo texto `textField_Texto`, deve-se chamar `addKeyListener()`.
- Implementar os métodos tratadores de eventos, no caso, os métodos das interfaces `ActionListener` e `KeyListener`. Isso pode ser feito automaticamente no Eclipse. Basta pressionar `<CTRL + 1>` sobre o erro e escolher, no menu suspenso, a opção `Add Unimplemented Methods`, conforme Figura

```
button_7 = new JButton("7");
button_7.addActionListener(this);
panel_Teclado.add(button_7);
```

Figura 16: Adicionando um observador de evento de ação.

17. O resultado é o fornecimento de implementação vazia para cada um dos métodos das duas interfaces (Figura 18)

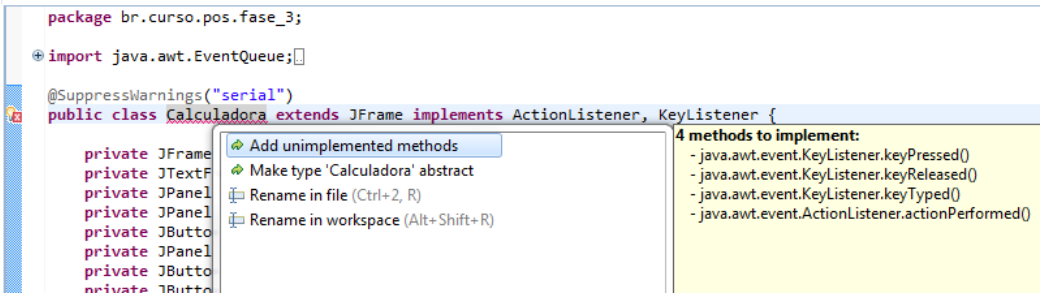

Figura 17: Adicionando tratadores de eventos de ação e de teclado.

```
@Override
public void keyPressed(KeyEvent arg0) {
}
@Override
public void keyReleased(KeyEvent arg0) {
}
@Override
public void keyTyped(KeyEvent arg0) {
}
@Override
public void actionPerformed(ActionEvent arg0) {
}
```

Figura 18: Adicionando tratadores de eventos de ação e de teclado.

Agora podemos fornecer uma implementação para o método `actionPerformed()`, que dará uma resposta genérica a um clique no botão 7 (Figura 19).

Até agora capturamos apenas o clique sobre o botão "7"! Esse código está muito amador. Podemos melhorá-lo muito ainda. Como já foi mencionado anteriormente, é possível entrar com os valores na calculadora tanto clicando nas teclas (implementado a interface `ActionListener`, chamando o observador de eventos de ação `addActionListener()` sobre o botão e implementando o método `actionPerformed()`) como digitando os valores no campo texto (implementando a interface `KeyListener`, chamando o observador `addKeyListener()` sobre o

```java
@Override
public void actionPerformed(ActionEvent arg0) {
	// TODO Auto-generated method stub
	if (arg0.getSource() == button_7){
		JOptionPane.showMessageDialog(null, "7", "Teste",
				JOptionPane.INFORMATION_MESSAGE);
	}
}
```

Figura 19: Codificação de um tratador de eventos de botão de ação.

campo texto e implementando o método `keyTyped()`). Vamos fazer agora uma implementação para `keyTyped()` que captura todas as teclas da calculadora (Figura 20) usando `switch case`. Observe que a captura do caractere digitado é feita chamando `getKeyChar()`.

```java
public void keyTyped(KeyEvent e) {
   String key = String.valueOf(e.getKeyChar());
   switch( key )
   {
      case "0":
      case "1":
      case "2":
      case "3":
      case "4":
      case "5":
      case "6":
      case "7":
      case "8":
      case "9": System.out.println("Dígito: "+ key); break;
      case ".":
      case "+":
      case "-":
      case "*":
      case "/":
      case "=": System.out.println("Oper.: "+ key); break;
      default:
   }
}
```

Figura 20: Tratamento de eventos de digitação de tecla numérica.

O mesmo pode ser feito para os eventos de ação (clique na tecla), mas, neste caso, a captura do valor teclado é feita obtendo-se o texto (`getText()`) do botão (`e.getSource()`).

Antes de prosseguir, vale observar que o código do `swich case` é idêntico par ambos os tradadores de evento! Por isso podemos propor que o código replicado seja um método. Para fazer isso, (1) selecione o código do `switch case` (incluindo as chaves), (2) tecle a combinação `<ALT + SHIFT + M>`, (3) nomeie o método como

`tratamentoTeclado` (Figura 22) e (4) conclua! O resultado pode ser observado na Figura 23.

```
public void actionPerformed(ActionEvent e) {
    String key = ((JButton)e.getSource()).getText();
    switch( key ) {
        case "0":
        case "1":
        case "2":
        case "3":
        case "4":
        case "5":
        case "6":
        case "7":
        case "8":
        case "9": System.out.println("Dígito: "+ key); break;
        case ".":
        case "+":
        case "-":
        case "*":
        case "/":
        case "=": System.out.println("Oper.: "+ key); break;
        default:
    }
}
```

Figura 21: Tratamento de eventos captura de clique.

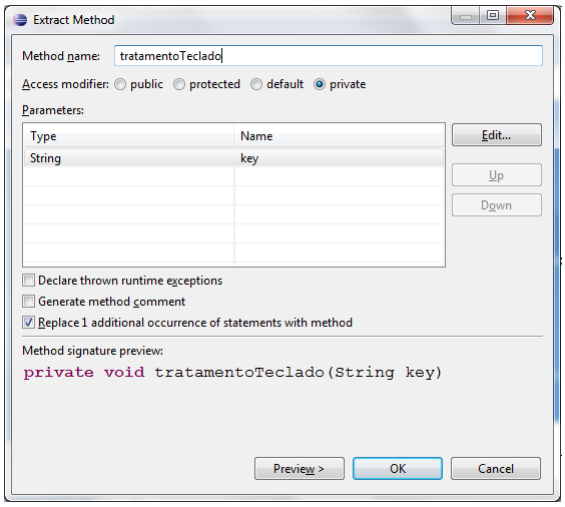

Figura 22: Tratamento de eventos captura de clique.

```
public void keyTyped(KeyEvent e) {
    String key = String.valueOf(e.getKeyChar());
    tratamentoTeclado(key);
}

public void actionPerformed(ActionEvent e) {
    String key = ((JButton)e.getSource()).getText();
    tratamentoTeclado(key);
}
private void tratamentoTeclado(String key) {
    switch( key )
    {
      case "0":
      case "1":
      case "2":
      case "3":
      case "4":
      case "5":
      case "6":
      case "7":
      case "8":
      case "9": System.out.println("Dígito: "+ key); break;
      case ".":
      case "+":
      case "-":
      case "*":
      case "/":
      case "=": System.out.println("Oper.: "+ key); break;
      default:
   }
}
```

Figura 23: Extraindo um método para o `switch case`.

4.1 Realizando operações

Se fizermos um *checklist* do desenvolvimento da calculadora até o momento, teremos:

- Criação do *layout* gráfico: ok!
- Programação dos eventos: ok!
- Execução das operações: nok!

O próximo passo é criar a lógica da calculadora. A Figura 24 mostra um possível projeto para a realização dos cálculos. Observe que tudo começa com a coleta de dígitos do teclado. A entrada do primeiro operando (`Operando1`) só é concluída quando digitamos o operador. Daí há duas possibilidades: (1) se o operador é unário (negação, ou fatorial, por exemplo), realiza-se o cálculo (`Calculo(=)`); (2) se o operador é binário (soma, produto e etc), coleta o segundo operando (`Operando2`). Após a coleta do segundo operando basta efetuar o cálculo (`Calculo(=)`). Após a

exibição do resultado, há duas possibilidades: (1) aproveitar esse resultado para efetuar um novo cálculo, ou (2) limpar o operando e iniciar a coleta do primeiro operando novamente.

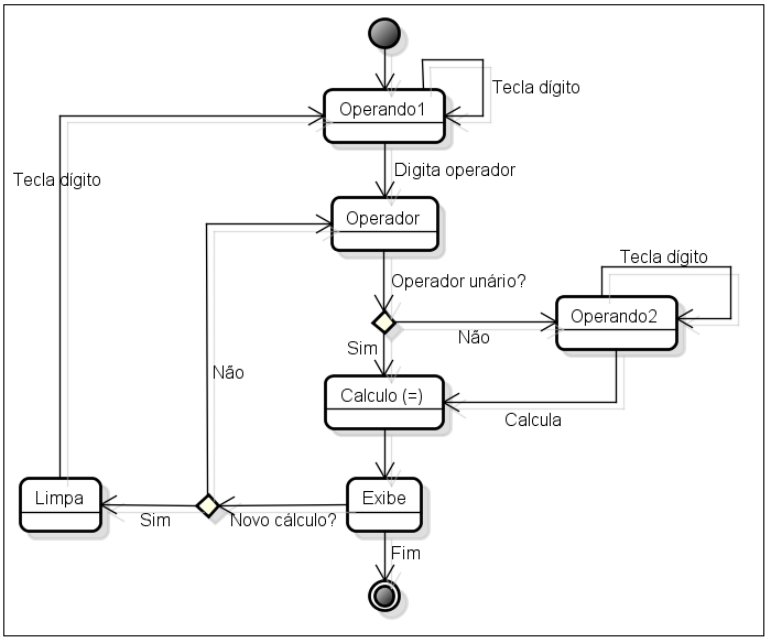

Figura 24: Fluxograma de operação da calculadora.

A programação da lógica pode ser feita em um classe separada, `CalculadoraLog`, por exemplo. Esta classe terá métodos estáticos para realizar as tarefas `obterOperando()`, `montarExpressao()` e `executarCalculo()`, por exemplo. Este métodos serão chamados dentro das opções do comando `switch case`.

5 Tratamento de erros

Em programação, erro (ou exceção) é qualquer evento que causa uma interrupção do fluxo normal de execução de uma sequência de código.

O programador de classes deve implementar a sua classe de tal forma que o seu funcionamento não fique prejudicado caso a mesma seja utilizada incorretamente. Algumas soluções inadequadas para o problema do tratamento de erros:

- Retorno de valor boolean para indicar o erro.
- Retorno de valor fora da faixa de escopo de trabalho do método.
- Exibição de mensagem de erro.
- Parada do processamento.

- Substituir o valor inválido por outro válido.
- Indicar o erro num campo criado somente para este propósito.

Java trata erros de forma mais direta e elegante. Em Java todas as exceções são classes que implementam a interface `Throwable`.

As exceções são classificadas nos seguintes tipos:

- Verificáveis (*checked exceptions*):
  - Devem ser explicitamente tratadas no código.
  - Exception e suas subclasses (exceto RuntimeException e subclasses).
- Não verificáveis (unchecked execptions):
  - Podem ser ignoradas durante o processo de codificação.
  - Error, RuntimeException e suas subclasses.

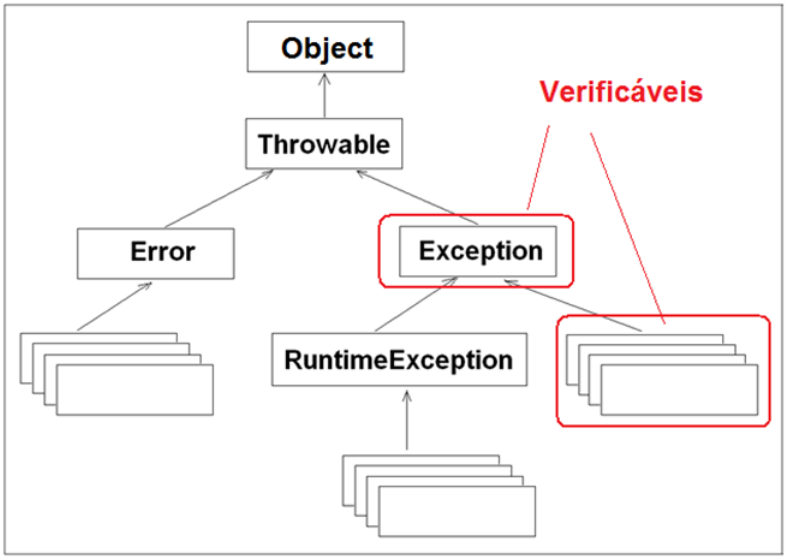

Figura 25: Esquema de tratamento de erros em Java.

O tratamento de erros em Java pode utilizar uma das seguintes abordagens:

- `try-catch-finally`: Execução normal no `try`, erro capturado no `catch` e execução de encerramento no `finally`.
- `throw`: Lança uma exceção, geralmente baseada em alguma condição de erro.
- `throws`: Lança uma exceção para ser tratada por quem chama o método que causa o erro.

A seguir (Figura 26) temos um exemplo simples de código que utiliza as três formas de tratamento de erros.

```
public class TratamentoErros{
      public static void main(String args[]){
            try{
                  int num1 = Integer.parseInt(args[0]);
                  int num2 = Integer.parseInt(args[1]);
                  if (num2 == 0){ // não verificável
                        throw new ArithmeticException("ERRO DE DIVISÃO POR ZERO");
                  }
                  else{
                        System.out.println("num1/num2 = " + div(num1, num2));
                  }
            }
            catch (NullPointerException e){
                  System.out.println(e.getMessage());
            }
            catch (ArrayIndexOutOfBoundsException e) {
                  System.out.println("Número de parâmetros insuficiente");
            }
            catch (NumberFormatException e){
                  System.out.println("Digite apenas numeros inteiros!!!");
            }
            finally {
                  System.out.println("O finally sempre é executado!!!");
            }
      }
      public static int div(int n1, int n2) throws ArithmeticException {
                  return n1/n2;
      }
}
```

Figura 26: Esquema de tratamento de erros em Java.

6 Organização em pacotes

O pacote é um agrupamento de código com a mesma finalidade. Um pacote é praticamente uma pasta onde você põe as classes que tem algo em comum. No Eclipse ele pode ser criado a partir do menu: `File > New > Package`. Uma pasta será criada com o nome do pacote.

Agora, com o conhecimento da abstração de pacote é que podemos apresentar de forma completa os quatro modificadores de acesso:

- Públicos (modificador `public`): Visíveis dentro e fora do pacote
- Protegidos (modificador `protected`): Visíveis dentro do pacote e fora apenas por herança
- Padrão (sem modificador): Visíveis apenas dentro do mesmo pacote
- Privados (modificador `private`): Visíveis apenas dentro da própria classe

A visibilidade dos membros (Figura 27) de uma pacote em relação a outros membros dentro e fora da classe e do próprio pacote é ditada pelas regras de visibilidade estabelecidas pelo uso dos modificadores de acesso.

Visibilidade	Público	Protegido	Padrão	Privado
Da mesma classe	Sim	Sim	Sim	Sim
De qualquer classe do mesmo pacote	Sim	Sim	Sim	Não
De uma subclasse do mesmo pacote	Sim	Sim	Sim	Não
De uma subclasse externa ao pacote	Sim	Sim	Não	Não
De uma classe C externa ao pacote que utiliza uma classe B (tb externa ao pacote), sendo B subclasse de uma classe A do pacote	Sim	Não	Não	Não

Figura 27: Esquema de tratamento de erros em Java.

7 Criação de "executável"Java (.jar)

O arquivo `.jar` é o executável Java. O Eclipse da um bom suporte à criação do arquivo `.jar`. Para criar um `.jar`, faça:

- Selecione (botão direito) o pacote, ou o projeto que pretende exportar.
- No menu suspenso, selecione `Export`.
- Na janela seguinte, selecione a opção `Java > Runnable JAR File`.
- Avance, selecione uma configuração de execução e defina o local de destino de arquivo `.jar`.
- Teste o executável.

8 Exercício: Criando uma GUI para a aplicação Calculadora

Crie a lógica da calculadora cuja GUI foi construída no item 2.

Referências

[1] Providing Constructors for Your Classes. The Java Tutorial. <http://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html>. Acessado em: 13 junho 2014.

[2] Passing Information to a Method or a Constructor. The Java Tutorial. <http://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html>. Acessado em: 13 junho 2014.

[3] API Specification. Java™ Platform, Standard Edition 7 <http://docs.oracle.com/javase/7/docs/api/>. Acessado em: 13 junho 2014.

[4] How to Set the Look and Feel. Java™ Platform, Standard Edition 7 <http://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html>. Acessado em: 13 junho 2014.

[5] How to Use Various Layout Managers. Java™ Platform, Standard Edition 7 <http://docs.oracle.com/javase/tutorial/uiswing/layout/layoutlist.html>. Acessado em: 13 junho 2014.

[6] A Visual Guide to Layout Managers. Java™ Platform, Standard Edition 7 <http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html>. Acessado em: 13 junho 2014.