

PROGRAMAÇÃO ORIENTADA A OBJETOS

CAPÍTULO 4 – VAMOS MELHORAR A INTERAÇÃO DO USUÁRIO COM O NOSSO SOFTWARE ?

Handerson Bezerra Medeiros

INICIAR

Introdução

Quando pensamos em como o usuário vê e interage com o *software*, precisamos ter uma ideia clara dos pilares da programação orientada a objetos e saber a importância da utilização adequada desses pilares, de forma a construirmos nosso código de maneira mais eficaz. Já sabemos como desenvolver reutilizando código, garantindo abstrações e, consequentemente, criando uma lógica de mais fácil entendimento. Mas como o usuário final irá interagir com nosso programa?

Até o momento, nos preocupamos em desenvolver uma lógica de negócio coerente, utilizando os mecanismos de desenvolvimento disponíveis na programação orientada a objetos. Agora, precisamos analisar e construir uma interface que servirá de apoio ao nosso usuário final, para interagir com nossos programas.

Entendendo o lado do usuário final, vamos poder lapidar nosso código, construindo novas regras de negócio, que se tornam responsáveis pela detecção de erros no nosso programa. E como construir a interface do nosso programa?

Neste capítulo, vamos aprender a criar a interação do usuário final com nosso programa. Veremos como é a criação de interfaces, conhecendo as bibliotecas gráficas disponíveis na linguagem Java. Vamos conhecer e analisar como devem ser criadas as regras de negócio, responsáveis por tratamento de exceções e erros do nosso programa. Por fim, vamos implementar um programa, de forma a colocar em prática todos os conhecimentos adquiridos sobre programação orientada a objetos, relembrando conceitos importantes.

Vamos estudar esse conteúdo a partir de agora, acompanhe!

4.1 Interagindo com o usuário

Para melhorar a interação do nosso programa com o usuário, precisamos realizar algumas mecânicas que informam às classes, ações que deverão ser obrigatoriamente executadas por cada uma delas. Porém, cada classe realiza as ações de maneira diferentes. Não entendeu? Resumindo, muitas classes podem ter acesso a um mesmo método, porém, cada classe executa esse método de maneira diferente. E, para garantir que tudo isso ocorra sem problemas, fazemos o uso de interfaces, para definir as ações obrigatórias de cada classe. Vamos entender como isso funciona.

4.1.1 Construção de Interfaces

Segundo Gomes (2012), podemos definir interface, como sendo um recurso que define um determinado grupo de classes para que tenha métodos, ou propriedades iguais. Porém, esses métodos iguais podem ser implementados de maneira distinta em cada classe desse grupo. Podemos afirmar, então, que interface é um contrato, ao qual as classes devem obedecer.

Para informar que uma classe implementará uma interface, ou seja, assinará um contrato dos métodos que deverá implementar, é só utilizar a palavra reservada *implements*, na criação desta classe (MANZANO; COSTA, 2014).

Por que utilizar interface em nosso programa? Você lembra que, utilizando a linguagem de programa Java não é possível criar herança múltipla? Então, quando utilizamos de interfaces em nosso programa, conseguimos fazer com que uma classe possua várias interfaces.

VOCÊ O CONHECE?

Para aperfeiçoarmos nossos conhecimentos, é necessário obter o auxílio de livros dos renomados autores existentes. Uma das autoras sobre o desenvolvimento de programação utilizando Java, é a Kathy Sierra. Após dez anos trabalhando na indústria *fitness*, Kathy decidiu mudar de área e assim iniciou uma renomada carreira na indústria da computação. Teve oportunidades de trabalhar com a UCLA e Macintosh, e também na área de desenvolvimento de *games*, se tornando uma das principais programadoras da equipe no *game* Terratopia (PEDRAZZI, 2016).

Um outro motivo de utilizarmos interfaces em nosso programa é que o desenvolvedor acaba seguindo um padrão de projeto. Facilitando assim, que ele siga um padrão, durante a implementação. Vamos aplicar esse conceito de interface em problemas reais para melhorar nosso entendimento.

4.1.2 Realização de Interfaces

Lembra do nosso programa de calcular a área de uma figura geométrica? Como você já sabe, esse cálculo vai variar, de acordo com a figura geométrica. Por exemplo, no quadrado, basta multiplicar por dois, a medida de um lado ($\text{área} = L^2$), já em um retângulo seria a base multiplicada pela altura ($\text{área} = B \times A$). Inicialmente, criamos uma interface chamada *CalcularArea*, a qual as classes devem obedecer. Ela contém uma assinatura de um método que é implementado pelas classes que efetivarem esta interface.

```
public interface Area{  
    public double calcular();  
}
```

Podemos observar que a classe do tipo interface não possui uma estrutura que informe detalhes lógicos dos métodos existentes dentro da interface. Ou seja, é apenas uma assinatura de quais métodos devem ser obrigatoriamente executados pelas classes que forem implementar essa interface. Para entender melhor, vamos analisar um código de uma classe que implementa essa interface acima.

```
public class Quadrado implements Area{  
    private double lado;  
    public Quadrado(double lado) {  
        this.lado = lado;  
    }  
    @Override  
    public double calcular(){  
        return this.lado * this.lado;  
    }  
}  
  
public class Retangulo implements Area{  
    private double base;  
    private double altura;  
    public Retangulo(double base, double altura) {  
        this.base = base;  
        this.altura = altura;  
    }  
    @Override  
    public double calcular(){  
        return this.base * this.altura;  
    }  
}
```

Como podemos notar, as classes *Quadrado* e *Retângulo* obedeceram ao contrato realizado pela interface, a partir do qual, elas estão implementando. Porém, cada classe implementou o método de maneira diferente.

VOCÊ QUER VER?

Aprendemos sobre interface, mas você sabia que interface é um dos recursos mais utilizados na linguagem de programação Java? Como vimos, interface garante um contrato de como as classes que assinam esse contrato, devem ter disponível aqueles métodos. Aprenda um pouco mais sobre interface, acessando o vídeo (GRONER, 2016): <<https://www.youtube.com/watch?v=6uLLfRNgRA4>>. Você pode aplicar todos os conceitos aprendidos em outra problemática.

Vamos analisar interfaces em outro cenário. Iremos criar uma classe mãe que represente a estrutura de caixa de som. Ela conterá todos os atributos e métodos necessários, que serão compartilhados para qualquer objeto que estende a classe mãe.

```
public class CaixaDeSom{  
    private boolean status;  
    private int tamanho;  
    private double potencia;  
    // Método construtor  
    public CaixaDeSom(int tamanho, double potencia){  
        this.status = false;  
        this.tamanho = tamanho;  
        this.potencia = potencia;  
    }  
}
```

```
//Metodos get e set de todos os atributos
```

.....

```
public boolean ligar(boolean b) {  
    status = b;  
    return status;  
}
```

```
public boolean desligar(boolean b) {  
    status = b;  
    return status;  
}  
}
```

Após feita a classe mãe, iremos criar a interface que será o contrato, no qual todos os objetos que estendem da classe *CaixaDeSom* deverá obedecer obrigatoriamente a todos os métodos contidos na interface. Vamos desenvolver a interface Controle.

```
public interface Controle{  
    //Metodos  
    boolean ligar();  
    boolean desligar();  
}
```

Agora que já temos nossa interface definida, vamos desenvolver duas classes filhas que irão utilizar todos os métodos definidos na interface Controle, porém, com implementações diferentes. Para ilustrar esse cenário criaremos as classes filhas *MiniSystem* e *SomCarro*.

```
public class MiniSystem extends CaixaDeSom implements Controle{
```

```
//Construtor

public MiniSystem (int tamanho, double potencia){

    super(tamanho, potencia);

}

//Metodos

public boolean ligar(){

    setStatus(true);

    return getStatus();

}

public boolean desligar(){

    setStatus(false);

    return getStatus();

}

}

public class SomCarro extends CaixaDeSom implements Controle{

//Construtor

public SomCarro (int tamanho, double potencia){

    super(tamanho, potencia);

}

//Metodos

public boolean ligar(){

    System.out.println("Você está ligando o Som!");

    return super.ligar(true);

}

public boolean desligar(){
```

```
System.out.println("Você está desligando o Som!");
return super.desligar(false);
}

}
```

Notamos que as classes filhas, apesar de estenderem a mesma classe mãe, executaram todos os métodos definidos pela interface, de maneiras distintas.

Trabalhar com interface é simples e acrescenta muito valor no nosso desenvolvimento. Garantindo que as classes implementem uma determinada estrutura, porém sendo executadas de maneiras diferentes. Proporcionando assim, uma alternativa para implementar o conceito de herança múltiplas.

4.2 Melhorando a interação com o usuário

Sabemos construir programas que interagem com o usuário textualmente. Mas você concorda que utilizar de uma interface gráfica, na qual o usuário pode interagir com o programa por meio de janelas e botões, proporciona uma interação mais agradável, né? É justamente sobre criação de interfaces gráficas, que vamos aprender agora. Vamos entender!

4.2.1 Visão geral das bibliotecas gráficas da linguagem Java

A linguagem de programação Java nos oferece uma variedade de bibliotecas que são utilizadas para a interação do usuário com o programa, de forma gráfica. São nossas interfaces gráficas de usuários (GUI). Em Java, as funcionalidades básicas podem ser encontradas em dois pacotes GUI: o `java.awt` e `javax.swing`.

A biblioteca AWT (*Abstract Window Toolkit*) foi a primeira API disponibilizada pelo Java que implementava as interfaces gráficas. Segundo Pereira (2017), a biblioteca AWT nos fornece um conceito de “*Write once, run anywhere*” (Escreva uma vez, execute em qualquer lugar), ou seja, se desenvolvemos uma caixa de texto para um programa que execute no Windows, ou Linux, ele deverá seguir o estilo daquela plataforma.

Após surgimento da biblioteca *Swing*, a AWT acabou sendo superada. É importante saber que a biblioteca *Swing* não é uma alternativa ao AWT, na verdade é uma complementação para a AWT (PEREIRA, 2017).

A *Swing* possui funcionalidades gráficas chamadas JVC (*Java Foundation Classes*), dando suporte na implementação de botões, menus, com funcionalidades *drag-and-drop* (arrastar e soltar), dentre outras coisas. A biblioteca que iremos aprender neste capítulo, será a *Swing*. Por se tratar de uma API implementada toda em JAVA, a *Swing* se torna mais flexível que a AWT.

Para garantir que ocorra portabilidade da plataforma, o *swing* apresenta o *look-and-feel* (traduzindo para o português, seria o conceito de olha e sinta) que garante que, independente de qual plataforma estiver executando aquele programa, em todas elas, o programa será apresentado com a mesma interface, por exemplo, de cores e tamanhos. Ou seja, conseguimos garantir que nosso programa se comporte da mesma maneira em todos os ambientes.

Mas só existe essas bibliotecas disponíveis? *Swing* e AWT são consideradas bibliotecas gráficas oficiais da linguagem Java, porém, podemos optar por utilizar outras bibliotecas disponíveis no mercado. Como, por exemplo, a biblioteca desenvolvida pela IBM, a SWT.

4.2.2 Divisão de responsabilidades: interface com o usuário e regras de negócio

Quando estamos desenvolvendo, utilizando interfaces, temos que ter em mente a divisão do nosso sistema em camadas. Essas camadas são de tipos diferentes.

A primeira é a **camada de apresentação, ou interface**, que interage diretamente com o usuário, e é por meio dela que teremos classes responsáveis por implementar a interface do sistema e capturar interações do usuário.

A segunda é a **camada de negócio**, onde ficam as funcionalidades e regras de todo o sistema. Por exemplo, em um sistema para controle de contas de um banco, o código nesta camada recebe as informações da camada de interface, realiza os débitos, créditos e o saldo da conta é calculado, bem como, regras da conta, como a regra que fará com que não seja permitido um débito maior do que o saldo disponível.

É na interface que o usuário informa os dados da conta e a operação desejada, por exemplo um saque. Esta informação será recebida por um código da camada de negócio, que, por sua vez, é responsável por processar a ordem de saque. Depois de

realizar a ação, a camada de negócio devolve para a camada de interface, informando o sucesso da operação, ou uma mensagem de erro.

É importante que essa divisão esteja clara em seus sistemas para que ele fique modular, todo o código referente à interface deve possuir como responsabilidade capturar e apresentar a informação para o usuário. Já o código da camada de negócio, processará essa informação, devolvendo para a interface o resultado desse processamento. É pensando nessa divisão de responsabilidade que vamos codificar.

4.2.3 Implementação de interfaces gráficas

A partir de agora, vamos estudar e implementar mais detalhadamente, a biblioteca *Swing*. Vamos conhecer alguns dos principais componentes utilizados pelo *swing*. Segundo Palmeira (2012), podemos citar os componentes:

- *JLabel* – exibe texto não editável ou ícones (controle *swing*, componente *label*);
- *JTextField* – insere dados do teclado e serve também para exibição do texto editável, ou não editável (controle *swing*, componente área de texto);
- *JCheckBox* – especifica uma opção que pode ser, ou não, selecionada (controle *swing*, componente caixa de seleção);
- *JComboBox* – fornece uma lista de itens, que possibilita ao usuário selecionar um item, ou digitar para procurar (controle *swing*, componente caixa de combinação);
- *JList* – lista para se selecionar vários itens (controle *swing*, componente listar).

Vamos entender melhor cada um desses componentes, desenvolvendo o programa de cadastro de produtos. Como meio de implementação desse exemplo, vamos utilizar o *Framework NetBeans*. Após criarmos um projeto de aplicativo Java, é necessário criar uma classe que será a interface gráfica do nosso programa. Para isso, basta clicar com o botão direito no projeto, novo e escolher *Form JFrame*, nomeando de *AddComida.java*.

Finalizado esse processo, nosso programa já possuirá (inclusive se for executado também) uma janela, conforme vemos na figura abaixo.

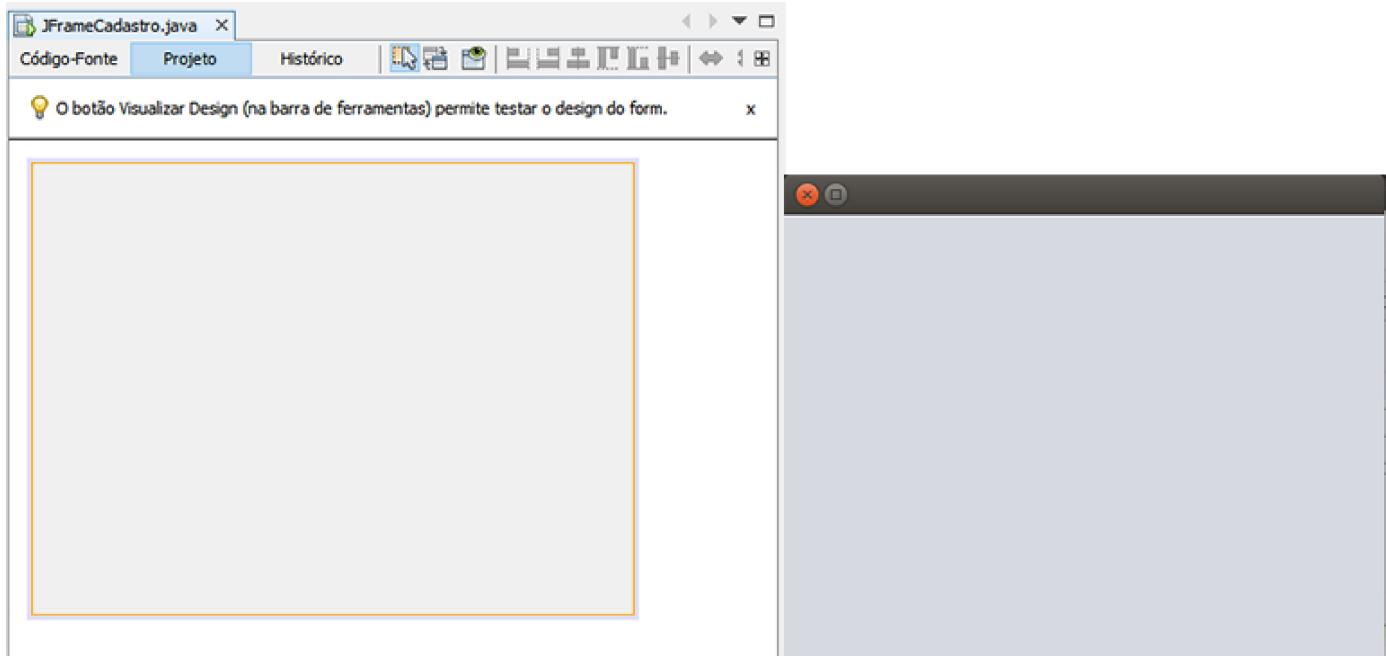


Figura 1 - Modo visual de programação da classe JFrame e janela em execução. Fonte: Elaborada pelo autor, 2018.

Podemos observar que, na área da programação, temos as opções de visualizar Projeto graficamente, como também, podemos visualizar as instruções de código na opção código-fonte ao clicar nesse botão, vemos o código gerado para a criação dessa tela. Apesar de não termos programado nenhuma linha de instrução, observamos, também, que já é possível executar nosso programa.

VOCÊ SABIA?

Temos no mercado vários *frameworks* disponíveis para nos auxiliares no desenvolvimento do nosso programa. Um deles é o *NetBeans*, ambiente de desenvolvimento integrado (IDE) Java desenvolvido pela empresa Sun Microsystems. Um dos diferenciais deste *framework* se dá pela facilidade de trabalhar com a interface gráfica, pois ele permite que o usuário “desenhe” a interface e apenas programe os eventos (TOSIN, 2011).

Como já sabemos, após criar o *JFrame*, podemos fazer a inclusão dos componentes visuais disponíveis pela biblioteca *swing* para modelarmos nossa interface gráfica. Para isso, utilizamos a paleta (vista na figura abaixo) apresentada pelo *NetBeans*, que

contém os componentes, propriedades e eventos, que podemos utilizar no nosso programa.

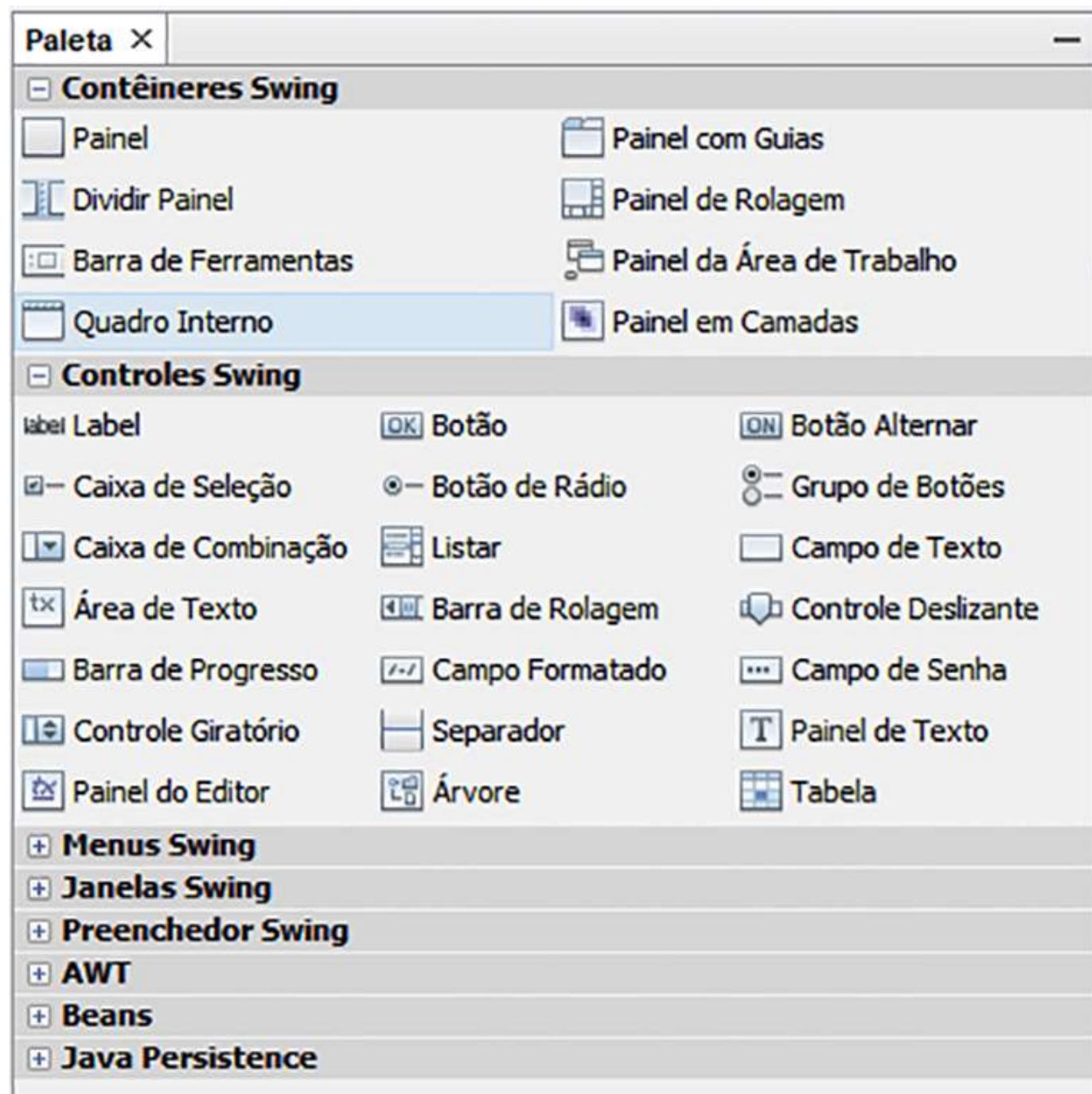


Figura 2 - Modo visual de programação da classe JFrame e janela em execução. Fonte: Elaborada pelo autor, 2018.

Essa paleta trabalha com arrastar e soltar, ou seja, se você deseja colocar um botão na tela, devemos clicar e segurar no componente da paleta e arrastarmos ele para um local no nosso *JFrame*. Ao colocarmos um *label* em nossa tela, podemos acessar logo abaixo da paleta, suas propriedades, como na figura abaixo. Nessas propriedades, podemos modificar sua cor (*background*), tipo da fonte (*font*), o texto

que aparecerá na tela (*text*), tamanho horizontal e vertical, por exemplo. Outro ponto importante, no botão código (figura a seguir), temos a opção *Nome da variável*, que nos informará a referência para esse *JText Field*. Por exemplo, utilizaremos esse nome, se precisarmos pegar o texto digitado pelo usuário no *JText Field* do nome.

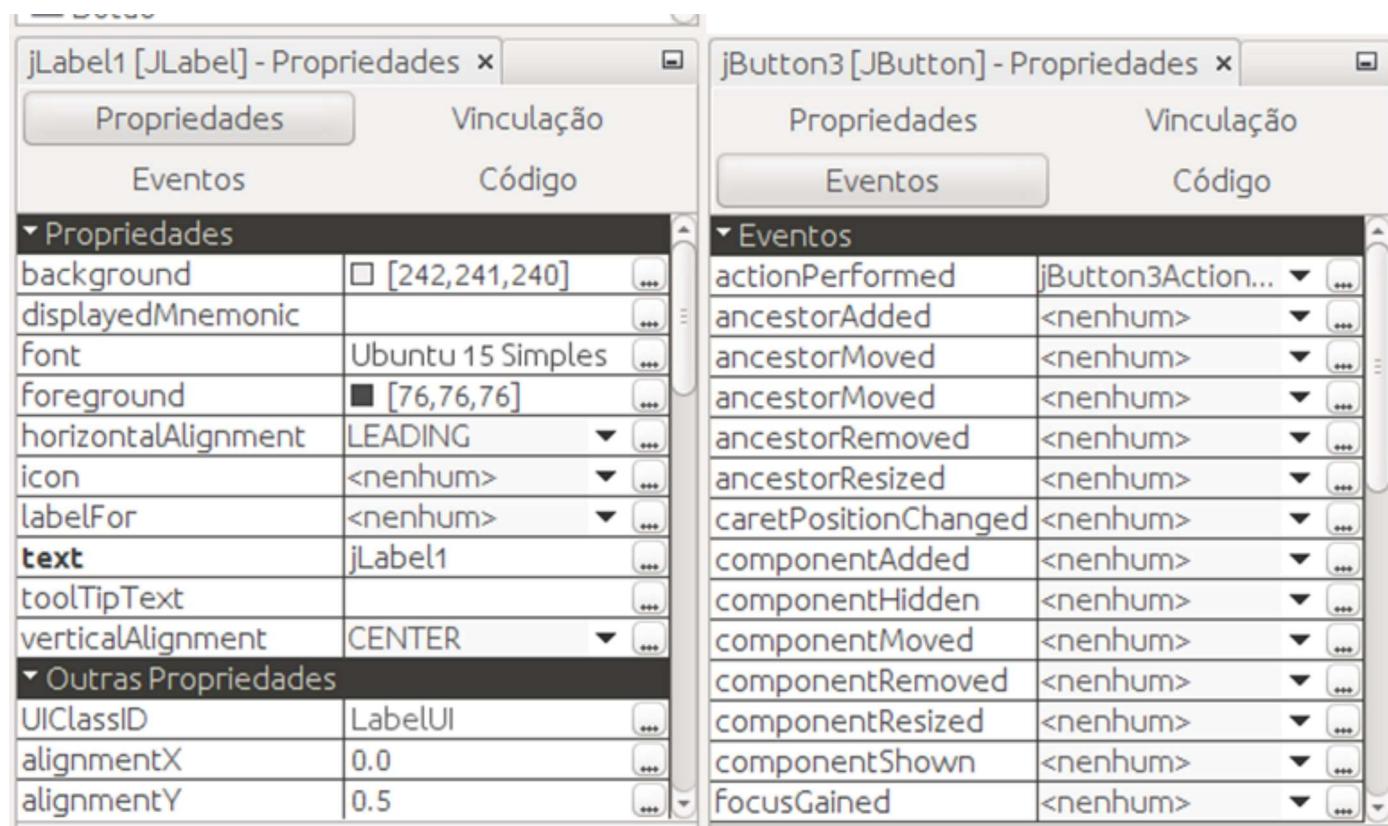


Figura 3 - Propriedades de um JLabel e de um JButton em um JFrame. Fonte: Elaborada pelo autor, 2018.

Outra boa prática é agrupar os componentes que fazem parte da nossa tela. Para isso, utilizamos os *JPanels* (Sessão Contêiner *swing* da paleta de componentes). Tais componentes possuem uma propriedade *Border*. Clicando nessa opção, podemos colocar um texto, indicando que os componentes internos se referem a "Dados da comida", como vemos na figura a seguir. Com isso, arrastando os componentes.

Inicialmente, adicionaremos dois *JPanels*. No primeiro, adicionaremos os *JLabel* e *JText Field*, referentes ao Nome, Valor e Dias Validade. No outro, adicionaremos três botões para Salvar, Cancelar e Limpar. Não se esqueça de alterar a propriedade *text*, para mudar o texto de cada componente e a propriedade *Border* do primeiro *JPanel*. Na figura abaixo, vemos como ficará nossas telas, após adicionar tais componentes.

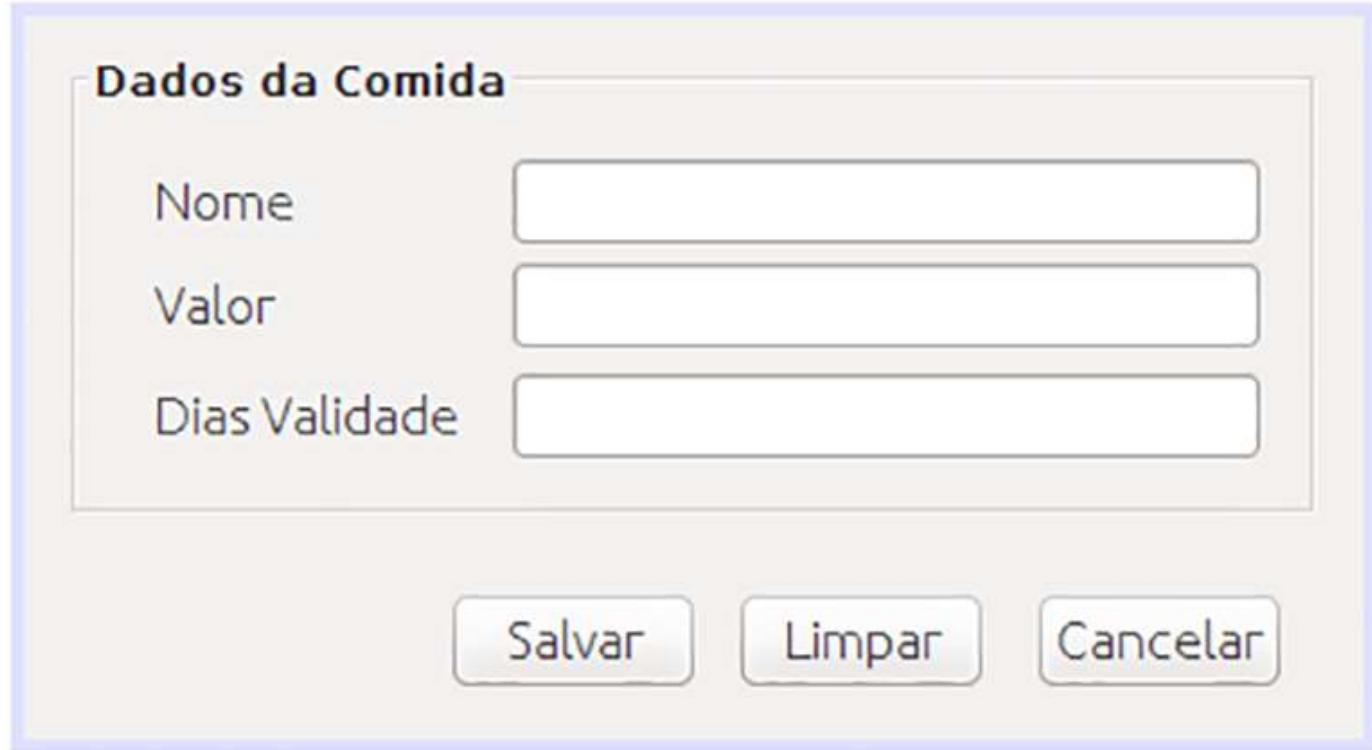


Figura 4 - Modo visual de programação com os componentes labels, campos de texto e botões e sua janela em execução. Fonte: Elaborada pelo autor, 2018.

Como podemos visualizar, agora nosso usuário consegue interagir de maneira mais clara com nosso programa, utilizando interface gráfica. Mas será que podem acontecer erros durante essa interação? E se o usuário informar um dado textual em um campo que deveria ser numérico? Para evitar esse e outros tipos de situações, precisamos criar regras em nosso programa. Vamos lá!

4.3 Criando regras

Ao desenvolver nossos programas, sempre estaremos a mercê de possíveis erros que possam vir a quebrá-los. Erros imprevistos que acontecem em tempo de execução são conhecidos como exceções que ocorrem quando algum erro de lógica não é resolvido pelo desenvolvedor. Realizar o tratamento correto dessa situação é muito importante para o correto funcionamento dos sistemas que você desenvolve.

4.3.1 Tratamento de exceções na abordagem orientada a objetos

Exceções (do inglês *Exception*, acrônimo de *Exceptional events*) são objetos que informam a ocorrência de algum problema em tempo de execução de um programa. Esses problemas podem se originar quando codificamos para acessar um índice de vetor que esteja fora do seu tamanho preestabelecido. Acessar algum membro de uma classe que possui uma referência nula; realizar uma operação matemática ilegal como, por exemplo, dividir um número por zero, dentre outros casos (PALMEIRA, 2013).

Na linguagem de programação Java, os tratamentos desses possíveis erros estão representados em uma hierarquia de classes e subclasses. Todas as classes de exceção herdam, direta, ou indiretamente, da classe *Exception*, formando uma hierarquia demonstrada na figura abaixo. A classe raiz de todas as exceções é *java.lang.Throwable*, objetos que sejam dessa classe, ou de suas classes derivadas, podem ser gerados, lançados e capturados pelo mecanismo de tratamento de exceções.

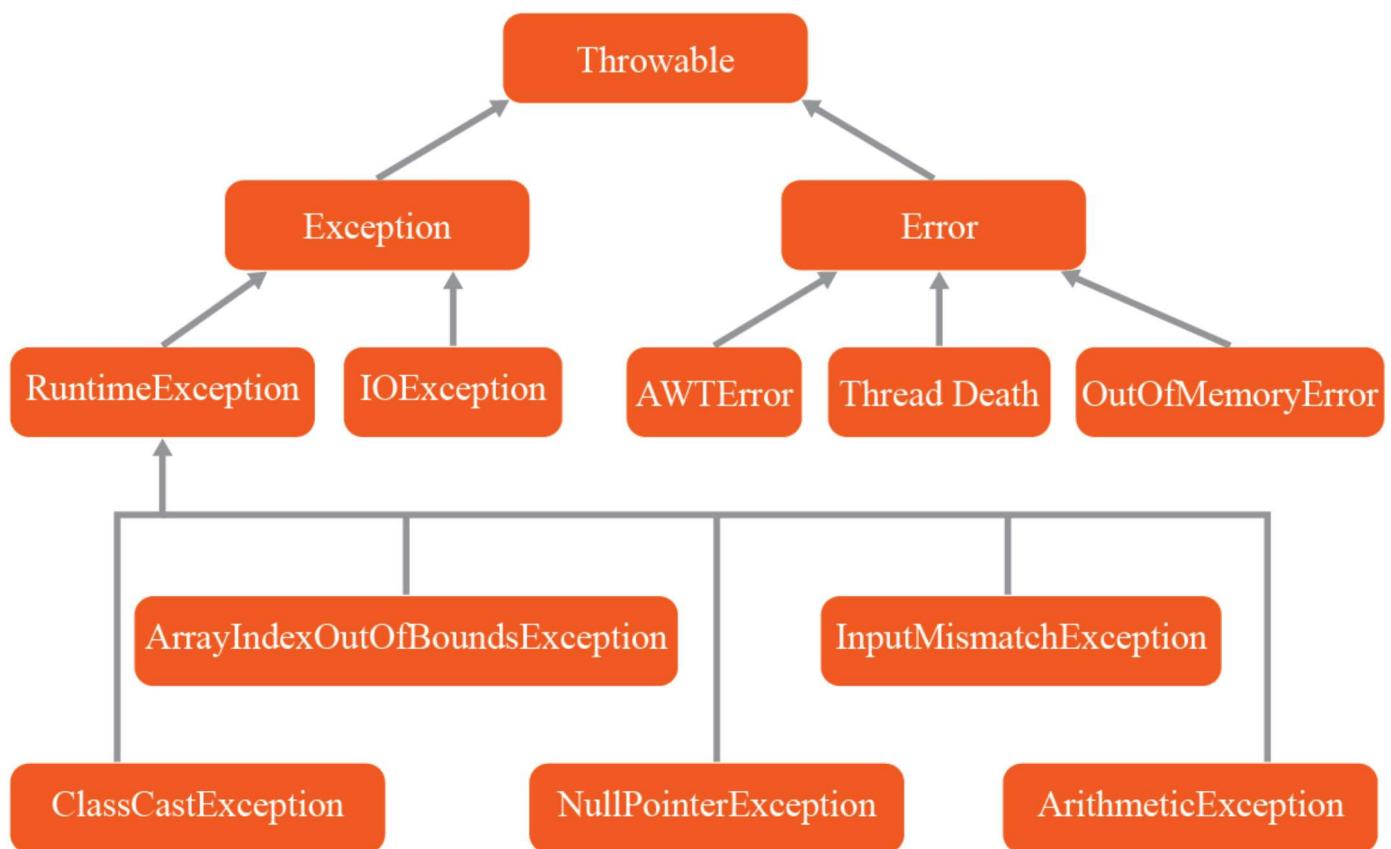


Figura 5 - Hierarquia das classes que implementam e herdam as exceções. Fonte: PALMEIRA, 2013, [s/p].

Explicando um pouco mais sobre essa hierarquia. Percebemos que a classe *Throwable* tem duas subclasses: a *java.lang.Exception* e a *java.lang.Error*. A classe *java.lang.Exception* é a raiz das classes derivadas de *Throwable*, essa classe informa

situações que a aplicação pode querer capturar e realizar um tratamento de exceção que permita prosseguir com o processamento, sem ocorrer interrupção do mesmo. Já a *java.lang.Error* é a raiz das classes derivadas de *Throwable*, que indica situações nas quais a aplicação não deve tentar tratar. Ela, geralmente, indica situações anormais, que não deveriam ocorrer (RICARTE, 2018).

Exemplos de exceções já definidas em Java incluem:

- *java.lang.ArithmetricException* – situações de erros em processamento aritmético, tal como uma divisão inteira por 0;
- *java.lang.InputMismatchException* – quando é tentada uma conversão de valor em formato *String* para um formato numérico, mas o conteúdo dessa *string* não representa, de fato, um número;
- *java.lang.ArrayIndexOutOfBoundsException* – quando a tentativa de acesso a um elemento, acarreta em um arranjo fora de seus limites – ou o índice era negativo, ou era maior, ou igual ao tamanho do arranjo;
- *java.lang.NullPointerException* – indica que o programa tenta usar um objeto, mas ele possui sua referência *null*, no qual uma referência a um objeto era necessária, como, por exemplo, invocar um método, ou acessar um atributo;
- *java.lang.ClassNotFoundException* – indica que a aplicação tentou carregar uma classe, mas não foi possível encontrá-la;
- *java.io.IOException* – indica a ocorrência de algum tipo de erro em operações de entrada e saída. É a raiz das classes que lançam exceções, quando estamos manipulando arquivos (*java.io.FileNotFoundException* e *java.io.InterruptedIOException*), dentre outras.

Na linguagem Java, existem dois tipos de exceções, que são: as exceções implícitas não precisam de tratamento e demonstram serem contornáveis, originados pela subclasse *Error*, ou *RunTimeException*. Já as explícitas necessitam que o desenvolvedor realize um tratamento, geralmente com o uso do comando *throw*, no qual deve ser declarado pelos métodos, essas exceções são originadas pela *IOException*.

De acordo com Sensolo (2014), outra tipificação é com relação aos erros dos tipos *throwables* pois, representam condições inválidas em locais que estão fora do controle do programa. Um método que possui código nessa situação, por exemplo,

erro na conexão com o banco de dados, falhas de rede, arquivos ausentes, é obrigado a estabelecer um tratamento das exceções lançadas.

Essas exceções são definidas como *Checked Exception*, presentes nos erros das subclasses de *Exception*. Também não requerem tratamento de exceção, porém podem vir a ocasionar o lançamento de exceções. Tais exceções são definidas como *Unchecked Exception* (subclasses de *RuntimeException*). Caso não haja tal tratamento, o programa acaba parando em tempo de execução (*Runtime*). E quando uma exceção não pode ser tratada pela aplicação, a JVM indica que existe algum problema, para o qual não é possível uma correção a partir de um tratamento realizado pelo próprio programa, fazendo com que a execução seja impossibilitada de continuar, *Error Exception* (MARINHO, 2016).

4.3.2 Implementação

Para exemplificar algumas dessas situações, vamos pegar como exemplo, um sistema de lanchonete Hora do Lanche. O sistema fornece uma funcionalidade de fechar a conta da mesa. Com essa funcionalidade, o sistema dá a possibilidade de ver a conta total e o valor de quanto ficaria se os clientes quiserem dividir igualmente, como mostra o método abaixo.

```
class Mesa {
    double valor;
    int numero;
    Mesa(double d, int n) {
        this.valor = d;
        this.numero = n;
    }
    public class Lanchonete {
        public List<Comida> comidas = new ArrayList<Comida>();
        //A classe Comida será implementada mais à frente nessa nossa aula
        public double calcularConta(int pessoas, double valor){
            return valor/pessoas;
        }
    }
}
```

}

```
public void adicionarComida(Comida comida){
```

```
try {
```

```
    if(!comida.estaVencido()){
```

```
        comedas.add(comida);
```

```
}
```

} catch (ComidaVencidaException ex) { //A classe Comida será implementada mais à frente nessa nossa aula

```
        System.out.println(comida.getNome() + " " + ex.getMessage());
```

```
}
```

```
}
```

```
public void fecharConta(Mesa mesa){
```

```
    Scanner leitor = new Scanner(System.in);
```

```
    System.out.print("Informe a quantidade de pessoas: ");
```

```
    int qtdPessoas = leitor.nextInt();
```

```
    double total = calcularConta(qtdPessoas, mesa.valor);
```

System.out.println("Mesa: " + mesa.numero + "Total a pagar cada pessoa: R\$" + total);

```
}
```

```
}
```

O método *fechar conta* recebe um objeto *mesa*, nele contém os produtos consumidos (comida, bebida) e o total da conta que deverá ser paga pelos clientes. Uma possível execução é mostrada abaixo, o total da mesa 1 é 67 reais e a divisão para cada um ficará R\$13.4 para cada.

Mesa: 1 Total a pagar: R\$67.0

Informe a quantidade de pessoas: 5

Total a pagar cada pessoa: R\$13.4

O método *calcularConta* recebe como argumento a quantidade de pessoas que pagam a conta e o valor da conta que será paga. Em seu corpo, é realizada a divisão desses dois valores e retornado seu resultado.

Até esse momento, tudo bem, porém um usuário desatento pode, sem querer, inserir valores que não foram previstos pelo desenvolvedor, as coisas podem não sair como planejado, como por exemplo, se o usuário digitar 0, na quantidade de pessoas.

Mesa: 1 Total a pagar: R\$67

Informe a quantidade de pessoas: 0

Exception in thread "main" java.lang.ArithmetricException: / by zero

at lanchonete.Lanchonete.calcularConta(Lanchonete.java:30)

at lanchonete.Lanchonete.fecharConta(Lanchonete.java:38)

at lanchonete.Lanchonete.main(Lanchonete.java:24)

O que foi que ocorreu? Quando o usuário digitou 0 na quantidade de pessoas, esse valor foi enviado para a função que tentou realizar a divisão do valor da conta (R\$67) e da quantidade de pessoas (0), porém, não existe divisão por zero. Por conta dessa limitação existente na matemática, a exceção *java.lang.ArithmetricException* foi gerada.

No momento em que a exceção é gerada, a JVM cria uma instância de um objeto da classe *Exception*, ou de uma de suas subclasses. Essa criação é chamada de lançamento de exceção, e no momento que ela ocorre, a aplicação é encerrada automaticamente. Isso pode causar um grande transtorno aos usuários do sistema.

O encerramento abrupto do programa mediante o lançamento de uma exceção pode ser evitado com o tratamento da mesma, o que é chamado de captura de exceção. Para realizar esse tratamento das exceções em Java são utilizados os

comandos *try* e *catch*.

```
try{
    //trecho de código que pode vir a lançar uma exceção
}

catch(tipo_excecao_1 e){
    //ação a ser tomada
}

catch(tipo_excecao_2 e){
    //ação a ser tomada
}

catch(tipo_excecao_n e){
    //ação a ser tomada
}
```

No bloco *try* são colocadas todas as linhas de código que possam vir a lançar uma exceção, essas exceções dos códigos colocados no *try* são tratadas pelo comando *catch*. A instrução *catch* lida com as exceções permitindo ao programa a capacidade de manter sua execução consistente, mesmo que exceções ocorram, sendo colocada nele, as linhas de código que serão executadas, quando a exceção for capturada.

Agora, realizando o tratamento de exceção em nossa função *fecharConta()* da classe *Lanchonete*, temos o seguinte código:

```
public void fecharConta(Mesa mesa){

    Scanner leitor = new Scanner(System.in);

    try{
        System.out.print("Informe a quantidade de pessoas: ");
        int qtdPessoas = leitor.nextInt();

        double total = calcularConta(qtdPessoas, mesa.valor);
    }
}
```

```

        System.out.println("Total a pagar cada pessoa: R$" + total);

    } catch (ArithmetricException e){

        System.out.println("Quantidade de pessoas não pode ser 0 ");

    } catch (InputMismatchException e){

        System.out.println("Verifique os valores digitados");

    } catch (Exception e){

        System.out.println("Ocorreu algum problema na função");

    } finally{

        System.out.println("Mesa: " + mesa.numero +" Total a pagar: R$" + mesa.valor);

    }

}

```

Para realizar o tratamento das exceções, é necessário em cada instrução *catch* discriminar qual exceção ele tratará. No código acima, o primeiro *catch* ficará responsável pela exceção *ArithmetricException*, qualquer erro referente a essa exceção, esse *catch* captura e realiza a execução do código presente nele, no caso, vai mostrar a mensagem: “Quantidade de pessoas não pode ser 0”.

Já o segundo *catch* captura as exceções referentes ao *InputMismatchException*, e assim como o primeiro, executará o código presente nele. Caso ocorra alguma exceção que não foi prevista, o terceiro *catch* captura as lançadas pelo tipo *Exception* e seus subtipos. Dessa forma, utilizando o conceito de herança e polimorfismo, conseguimos tratar as exceções que foram identificadas pelo desenvolvedor e casos que ainda não puderam ser previstos.

Agora, imagine que, independente da ocorrência da exceção, ou não, se deseja mostrar o total a pagar da mesa. A bloco *finally* permite que um bloco de comandos seja sempre executado após a execução de um bloco de *catch* e/ou de *try*, finalizando essa sequência de comandos do sistema, independente de ocasionar algum erro no sistema. O uso dele é opcional, não sendo obrigatório sua inserção após o uso do *try/catch*.

Ao executar novamente o código acima, informando a quantidade de pessoas zero, temos a saída:

Informe a quantidade de pessoas: 0

Quantidade de pessoas não pode ser 0

Mesa: 1 Total a pagar: R\$67

Já se o usuário digitar um valor que não seja um número, teremos a execução abaixo:

Informe a quantidade de pessoas: n

Verifique os valores digitados

Mesa: 1 Total a pagar: R\$67

Observe que, mesmo informando valores que deveriam dar errado, por conta do tratamento realizado, o programa não parou e ainda foi possível informar qual erro foi inserido no sistema.

Imagine uma situação na qual nenhuma exceção existente no java faça sentido para um erro em seu sistema, como por exemplo, verificar se uma comida está vencida, ou não. Das classes disponíveis na hierarquia de classes, não temos uma que nos auxilie neste trabalho. Assim como qualquer objeto criado em Java, também é possível criar suas próprias exceções.

Para criar uma exceção, devemos criar uma classe e a mesma estender de *Exception*, como podemos ver na classe abaixo. Por herdar de *Exception*, podemos usar o conceito de polimorfismo para sobrestrar seus métodos, inclusive o método que retorna uma mensagem, quando a exceção é gerada, *getMessage()*.

```
public class ComidaVencidaException extends Exception{  
    @Override  
    public String getMessage(){  
        return "esta vencida";  
    }  
}
```

}

Essa exceção é lançada em um método da classe comida, definida logo abaixo, possuindo os atributos: nome; valor que representará o quanto o cliente deve pagar pela comida, data que representa a data de fabricação daquela comida; e data de vencimento que indica, quantos dias aquele alimento dispõe até o vencimento.

A classe ainda possui o método *estaVencido*, que verifica se a comida está vencida, realizando

a comparação da data de fabricação, mais data validade, com a data atual. Em seguida, os métodos *gets*.

```
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
public class Comida {
    private String nome;
    private Double valor;
    private Date dataFabricacao;
    private int diasValidade;
    Calendar calendar = Calendar.getInstance();
    //Construtor de comida
    public Comida(String nome, Double valor, Date data, int dias) {
        this.nome = nome;
        this.valor = valor;
        this.dataFabricacao = data;
        this.diasValidade = dias;
    }
    //Método para verificar se a comida está vencida
    public boolean estaVencido() throws ComidaVencidaException{
```

```
Date dataAtual = calendar.getTime();  
calendar.setTime(this.dataFabricacao );  
calendar.add(Calendar.DAY_OF_MONTH, diasValidade);  
Date dataProdutoComValidade = calendar.getTime();  
if(dataProdutoComValidade.after(dataAtual)){  
    return false;  
} else {  
    throw new ComidaVencidaException();  
}  
}  
  
//Métodos Gets  
  
public String getNome() {  
    return nome;  
}  
  
public Double getValor() {  
    return valor;  
}  
  
public Date getData() {  
    return dataFabricacao;  
}  
  
public int getDiasValidade() {  
    return diasValidade;  
}
```

{

Nas quatro primeiras linhas desse método *está vencido*, é feito o cálculo da diferença de dias, no qual, na primeira linha, é pega a data atual, em seguida, pega-se a data atual e adiciona a ela, a quantidade de dias de validade do produto e cria-se uma nova data com essa soma. No *if* seguinte, é verificado se essa data do produto com a validade, está depois da data atual, se estiver, o produto está na validade. Senão a exceção *comida vencida* é lançada com o uso da cláusula *throw*.

Essa situação não é desejada que uma exceção seja tratada no método, mas sim em outro que venha lhe chamar. Nesses casos, é necessário que o tratamento da exceção fique a cargo de quem chamou esta classe, ou método. Esse lançamento da exceção de uma função para outra é feito usando a cláusula *throws* na assinatura do método e o comando *throw* cria, de fato, essa exceção.

VOCÊ QUER LER?

Quando estamos trabalhando com tratamento de exceções podemos utilizar *exceção checada*, ou *não checada*. Optando por uma *exceção checada*, fará com que o programador, obrigatoriamente, trate esse erro de alguma maneira, seja ela por captura, ou tratamento. Caso optarmos por *não checada*, fica a critério do desenvolvedor tratar, ou não (SOUZA, 2013a). Quer entender isso melhor? Leia o artigo completo: [<https://www.devmedia.com.br/checked-exceptions-versus-unchecked-exceptions-trabalhando-com-excecoes/29626>](https://www.devmedia.com.br/checked-exceptions-versus-unchecked-exceptions-trabalhando-com-excecoes/29626) (<https://www.devmedia.com.br/checked-exceptions-versus-unchecked-exceptions-trabalhando-com-excecoes/29626>)>.

No momento que vamos adicionar a comida na lanchonete, deve-se verificar se a comida está vencida, ou não. Para isso, a classe *Lanchonete* deve possuir uma função *adicionarComida*. Esta função recebe um objeto do tipo *comida* e o adiciona na lista de comida na lanchonete, porém, antes de adicionar, é verificado se o mesmo está dentro do prazo da validade, chamando a função *estaVencido*. O fato dessa função poder lançar a exceção *ComidaVencida*, faz com que o uso do *try* nele seja obrigatório, como podemos ver no código do método *adicionarComida* presente de uma classe *Lanchonete*, por exemplo,

```
public void adicionarComida(Comida comida){  
    try {  
        if(!comida.estaVencido()){  
            comedas.add(comida);  
        }  
    } catch (ComidaVencidaException ex) {  
        System.out.print(comida.getNome() + " " +ex.getMessage());  
    }  
}
```

VOCÊ QUER LER?

Percebemos o quanto podemos tornar um programa crítico se não resolvemos e tratarmos os erros e exceções existentes no programa. Para garantir isso, podemos sempre utilizar os sete hábitos das exceções que nos auxilia no desenvolvimento. Esses hábitos podem ser identificados como: use exceções para erros, declare erros em uma granularidade adequada, mantenha o *stack trace*, utilize exceções existentes, tenha exceções bem informadas, não ignore erros e, por fim, separe níveis de abstração (GUERRA, 2018). Vamos entender esses hábitos com mais detalhes?

Agora, podemos desenvolver nosso programa de maneira que ele saiba lidar com erros e exceções durante a sua execução com o usuário. Vamos aprender mais um pouco? Veremos agora, uma outra aplicação prática que vai nos ajudar a entender melhor a interação de interface gráfica e tratamento de erros.

4.4 Colocando em prática

Bem, vimos o quanto é importante trabalharmos com o tratamento de regras e exceções em nosso código, como também, desenvolvermos uma interface interativa e de fácil manipulação, para que nossos usuários utilizem de maneira mais confortável os nossos programas. Agora, vamos colocar em prática os conceitos

vistos nesse capítulo. Para isso iremos criar o projeto da lanchonete com interface gráfica. Neste projeto, nós teremos a opção *ver* adicionar novas comidas a lanchonete, e podemos ver as que já estão cadastradas.

4.4.1 Integrar os conceitos com interface gráfica e tratamento de exceções

Inicialmente iremos usar a classe *Comida* e a exceção *ComidaVencidaException*, que já utilizamos anteriormente. Criando, em seguida, interfaces para manipular essas classes, nossa interface terá duas telas, a principal que lista as comidas do restaurante e nos fornece a opção de adicionar mais comidas. E a segunda tela será a de cadastro de comidas, na qual o usuário fornece as informações e salva a comida.

CASO

Em 1996, um dos mais novos foguetes não-tripulados da Europa, o Ariane 5, se desligou sozinho, após segundos de seu lançamento inaugural. Também foram destruídos quatro satélites científicos usados para estudar como o campo magnético da Terra interage com os ventos solares, ambos colidiram com o chão e explodiram. A causa desses desligamentos ocorreu quando o sistema de orientação tentou transformar a velocidade do foguete de 64-bits para um formato de 16 bits, ou seja, o número era muito grande para realizar essa conversão, o que resultou em erro de estouro. O custo desse problema foi orçado em \$500 milhões (HINKEL, 2018).

Classe AddComida.java

Se lembra do nosso *JFrame* criado na sessão 4.2.3? Vamos retomá-lo, criando ações de clicar nos botões. Vale lembrar que os nomes dos componentes da nossa interface são gerados pelo *NetBeans*, e em alguns projetos eles podem variar. Se tiver dúvida, selecione o componente desejado, na paleta lateral das propriedades, clique no botão código e você verá a opção *Nome da Variável*.

Os métodos dos eventos a seguir, serão na classe *AddComida.java*. O primeiro evento é o de adicionar comida utilizando o botão *salvar* (*JButton3*). Para adicionar esses eventos, podemos fazer de duas maneiras: dar dois cliques com o botão direito do mouse no botão, ou acessar na caixa de propriedades a aba eventos e procurar a propriedade *actionPerformed*. Ambas as maneiras geram e te redirecionam para a

área de código desse evento. Esse evento acessa o que o usuário digitou nos *TextFields*, pelo método *getText()*. Como é possível que o usuário digite informações que não condizem com os campos, ou não digite nada, os *catches* são necessários para evitar esses possíveis erros, por fim, os métodos deverão ficar como o código abaixo.

```
Comida comida;  
//botão salvar  
  
private void JButton3ActionPerformed(java.awt.event.ActionEvent evt) {  
  
try {  
    String nome = JTextField1.getText(); //Pegando o texto referente ao nome da  
comida  
  
    Double valor = Double.parseDouble(JTextField2.getText()); // Pegando o texto  
referente ao valor  
  
    int data = Integer.parseInt(JTextField3.getText()); // Pegando o texto  
referente ao dias validade  
  
    comida = new Comida(nome, valor, new Date(), data);  
  
    JOptionPane.showMessageDialog(null, "Comida adicionada com sucesso",  
"Adicionar Comida", 1);  
  
} catch (java.lang.NumberFormatException e) {  
  
    JOptionPane.showMessageDialog(null, "Não foi possível adicionar comida,  
verifique os valores digitados", "Adicionar Comida", 0);  
  
}  
  
this.dispose();  
}
```

Para cancelar (*JButton1*), ou seja, fechar a tela, adicionaremos um novo evento que chamaremos a função *dispose*:

```
//botão fechar
```

```
private void JButton1ActionPerformed(java.awt.event.ActionEvent evt) {  
  
    this.dispose();  
}
```

Para limpar os *JTextField*, criaremos um evento para o botão de limpar (*JButton2*), esse evento setará os valores dos *TextFields* para carácter vazio, como visto no código abaixo.

```
//botão limpar
```

```
private void JButton2ActionPerformed(java.awt.event.ActionEvent evt) {  
  
    JTextField1.setText("");  
    JTextField2.setText("");  
    JTextField3.setText("");  
}
```

Classe Principal.java

Criaremos no projeto, um novo *Form JFrame* chamado *Principal* (botão direito do mouse no projeto, opção novo e selecionar *Form JFrame*), que será a primeira tela do nosso sistema. Dentro do *JFrame Principal*, utilizaremos a paleta de componentes para adicionar os componentes nessa tela: menu superior (*JMenuBar*), esse componente vem com um menu de itens (*JMenu*), no qual podemos colocar vários Itens de Menus (*JMenuItem*), eles podem ser usados para criar menus, como, por exemplo, uma opção para adicionar comida. Feito isso, o conteúdo central da nossa tela, colocaremos um *JPanel* e dentro dele uma *JTable*.

VOCÊ SABIA?

Já ouviu falar em DTO? DTO é a sigla para *Data Transfer Objects*. Mas o que isso significa? Usamos DTO quando queremos transferir dados de um local de um objeto para outro objeto durante a aplicação. Ou seja, podemos dizer que são objetos que possuem atributos e apenas seus métodos assessores, por exemplo, *gets* e *sets* (SOUZA, 2013b).

Para adicionar as colunas da tabela, como vemos na figura a seguir, você deve clicar com o botão direito no componente *JTable*, acessar a opção conteúdo da tabela e na aba colunas, você verá o botão *inserir*. Com ele você consegue inserir cada uma das colunas. Ao terminar, teremos uma tela como vemos na imagem abaixo.

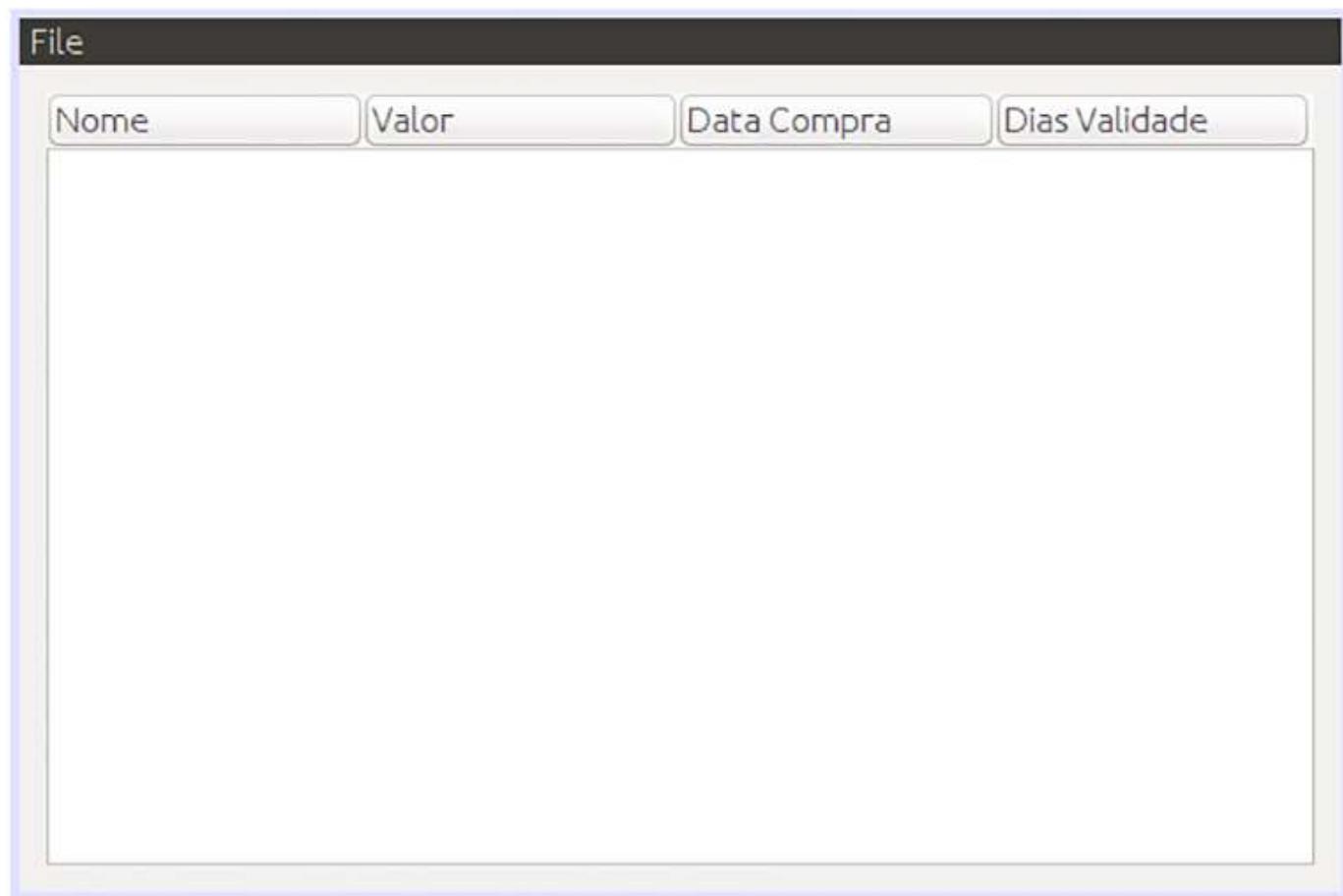


Figura 6 - Exemplo da tela principal desenvolvida com os componentes JMenu, JTable adicionados. Fonte: Elaborada pelo autor, 2018.

Depois de montados os componentes na tela principal, devemos agora criar os eventos de chamar a tela de adição de comida e listar a comida adicionada. A primeira ação que devemos fazer é editar o construtor do *JFrame* na classe *Principal.java*. Para podermos adicionar ou remover elementos da *JTable*, temos que trabalhar com um objeto *DefaultTableModel*, toda a alteração que fizermos nele será

refletida no componente *JTable*, por isso temos que obter sua referência primeiro, para depois usar. Acesse o código do *JFrame Principal.java*, botão código fonte, crie uma variável do tipo *DefaultTableModel* e no constructor da classe *Principal.java*, obtenha a referência do modelo usada no *JTable* da interface gráfica.

```
public DefaultTableModel tableModel;

/*
 * Creates new form Principal
 */
public Principal() {
    initComponents();
    this.setLocationRelativeTo(null);
    tableModel = (DefaultTableModel) JTable1.getModel();
}
```

Para finalizar, no *JFrame Principal*, criaremos o evento para o *JMenuItem* adicionar botão. Esse evento cria o *JDialog*, da tela de Adicionar comida, e coloca sua visibilidade para verdadeiro, fazendo com que seja mostrada a tela. Em seguida, acessa a comida desse *JDialog*, verificando se ela está vencida. Nesse caso, o *try* é requerido, pois caso a comida esteja vencida, a exceção será lançada, sendo tratada no *catch*. Outra exceção que pode ocorrer, é caso o usuário cancele a adição da comida, o evento pegará a comida nula, por isso temos o *catch* para o *NullPointerException*.

```
private void jMenuItem1ActionPerformed(java.awt.event.ActionEvent evt) {
    AddComida adicionarJDialog = new AddComida(this,
    rootPaneCheckingEnabled);
    adicionarJDialog.setVisible(true);
    try {
```

```
Comida comida = adicionarJDialog.comida;
comida.estaVencido();

tableModel.addRow(new Object[] {comida.getNome(), comida.getValor(),
comida.getData(), comida.getDiasValidade()});

} catch (java.lang.NullPointerException e) {
} catch (ComidaVencidaException ex) {
JOptionPane.showMessageDialog(null, ex.getMessage(), "Comida Vencida",
0);

}
```

Como isso, temos a finalização deste projeto, ao executarmos o *JForm* principal, veremos as telas abaixo.

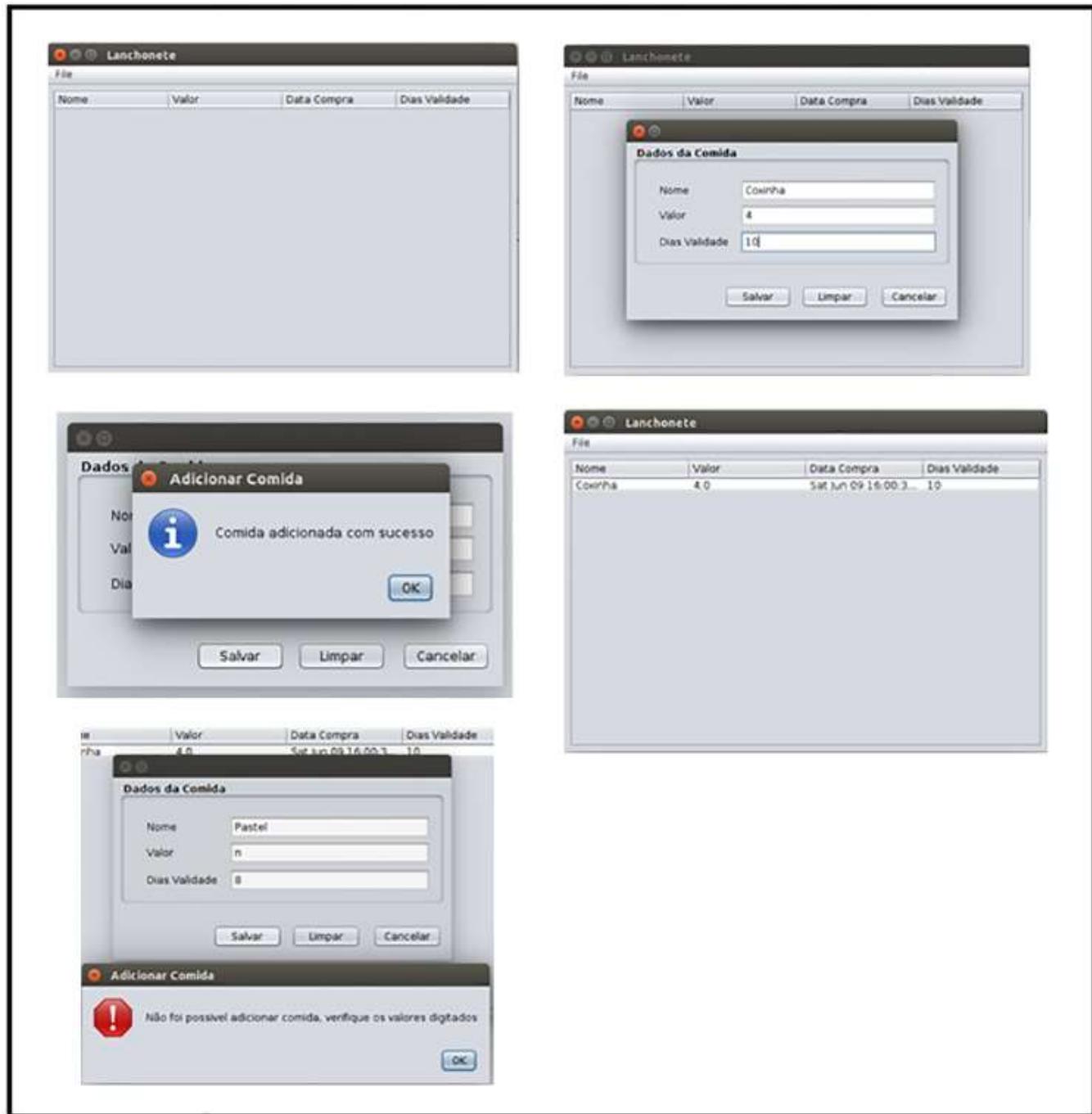


Figura 7 - Telas desenvolvidas para visualizar e adicionar comidas. Fonte: Elaborada pelo autor, 2018.

Nas imagens vemos o fluxo do processo de cadastro de um novo item comida.

Como podemos notar pelo nosso código, conseguimos, agora, realizar um *feedback* ao usuário, após a realização de um cadastro. E, também, conseguimos informá-lo, caso ocorra algum erro de inserção, como, por exemplo, se o usuário informou um valor textual em um campo numérico.

Síntese

Chegamos ao final de mais um capítulo. Aprendemos como podemos melhorar a interação do usuário final com nossos programas. Estudando conceitos de interface gráfica e tratamento de erros, dessa forma estaremos evitando possíveis erros que vierem a ocorrer na regra de negócio desenvolvida para nosso programa. Assim, podemos gerar um programa de mais usabilidade.

Neste capítulo, você teve a oportunidade de:

- compreender o significado de interface, observando as vantagens de utilizar devido à restrição de implementarmos o conceito de herança múltipla;
- aprender sobre interface gráfica, conhecendo e aplicando as bibliotecas disponíveis na linguagem de programação java;
- estudar sobre tratamento de erros e exceções, avaliando e desenvolvendo soluções que são capazes de identificar e tratar erros, durante a execução do programa;
- aplicamos os conceitos de interface gráfica e tratamento de erros em um exemplo completo que integra os dois conceitos.



Bibliografia

GOMES, R. F. Entendendo interfaces Java. **Portal Devmedia**, 2012. Disponível em: <<https://www.devmedia.com.br/entendendo-interfaces-em-java/25502>> <https://www.devmedia.com.br/entendendo-interfaces-em-java/25502>.>. Acesso em: 04/10/2018.

GRONER, L. **Curso de Java 44:** Orientação a Objetos: Interfaces. Canal Loiane Groner, YouTube. Publicado em 17 de fev. de 2016. Disponível em: <<https://www.youtube.com/watch?v=6uLLfRNgRA4>> https://www.youtube.com/watch?v=6uLLfRNgRA4 (https://www.youtube.com/watch?v=6uLLfRNgRA4). Acesso em: 04/10/2018.

GUERRA, E. Os sete hábitos das Exceções altamente eficazes. Revista **MundoJÁgil**, 2018. Disponível em: <<http://www.univale.com.br/unisite/mundo-j/artigos/36Setehabitos.pdf>> (<http://www.univale.com.br/unisite/mundo-j/artigos/36Setehabitos.pdf>). Acesso em: 04/10/2018.

HINKEL, N. Ariane 5: um erro numérico (*overflow*) levou à falha no primeiro lançamento, **Boletim SBMAC**, Divisão de Mecânica e Controle MCT/INPE São José dos Campos – SP, 2018. Disponível em: <<http://www.sbmac.org.br/bol/bol-2/artigos/ariane5.html>> (<http://www.sbmac.org.br/bol/bol-2/artigos/ariane5.html>). Acesso em: 04/10/2018.

MANZANO, J. A. G.; COSTA JR., R. **Programação de Computadores com Java**. Érica, 2014. 127p. [Minha Biblioteca]

MARINHO, A. L. **Programação Orientada a Objetos**. São Paulo: Pearson Education do Brasil, 2016. 167p.

PALMEIRA, T. V. V. Introdução a Interface GUI no Java. **Portal Devmedia**, 2012. Disponível em: <<https://www.devmedia.com.br/introducao-a-interface-gui-no-java/25646>> https://www.devmedia.com.br/introducao-a-interface-gui-no-java/25646 (<https://www.devmedia.com.br/introducao-a-interface-gui-no-java/25646>). Acesso em: 04/10/2018.

PALMEIRA, T. V. V. Trabalhando com Exceções em Java. **Portal Devmedia**, 2013. Disponível em: <<https://www.devmedia.com.br/trabalhando-com-excecoes-em-java/27601>> https://www.devmedia.com.br/trabalhando-com-excecoes-em-java/27601 (<https://www.devmedia.com.br/trabalhando-com-excecoes-em-java/27601>). Acesso em: 04/10/2018.

PEDRAZZI, P. **Kathy Sierra on Designing for Badass**. Portal Building Winning Products, publicado em 10 de outubro de 2016. Disponível em: <<https://medium.com/building-winning-products/kathy-sierra-on-designing-for-badass-ba92cd5fad96>> (<https://medium.com/building-winning-products/kathy-sierra-on-designing-for-badass-ba92cd5fad96>). Acesso em: 04/10/2018.

PEREIRA, V. N. Aula 01 AWT (*Abstract Window Toolkit*). Portal Instituto de Computação. Universidade Estadual de Campinas, 2017. Disponível em: <(https://www.ic.unicamp.br/~ra100621/class/2017.2/ALPOO_files/aula01.html)>
<(https://www.ic.unicamp.br/~ra100621/class/2017.2/ALPOO_files/aula01.html)>
<(https://www.ic.unicamp.br/~ra100621/class/2017.2/ALPOO_files/aula01.html)>. Acesso em: 04/10/2018.

RICARTE, I. L. M. A hierarquia de exceções. Faculdade de Engenharia Elétrica e de Computação. Universidade Estadual de Campinas, 2018. Disponível em: <(http://www.dca.fee.unicamp.br/cursos/PooJava/excecoes/exc_hiercls.html)>
<(http://www.dca.fee.unicamp.br/cursos/PooJava/excecoes/exc_hiercls.html)>
<(http://www.dca.fee.unicamp.br/cursos/PooJava/excecoes/exc_hiercls.html)>. Acesso em: 04/10/2018.

SENSOLO, A. Diferença entre exceção checadas e não checadas? Portal **Stack Overflow**, 2014. Disponível em: <<https://pt.stackoverflow.com/questions/10732/qual-%C3%A9-a-diferen%C3%A7a-entre-exce%C3%A7%C3%A7%C3%85es-checadas-checked-e-n%C3%A3o-checadas-unchecked>> (<https://pt.stackoverflow.com/questions/10732/qual-%C3%A9-a-diferen%C3%A7a-entre-exce%C3%A7%C3%A7%C3%85es-checadas-checked-e-n%C3%A3o-checadas-unchecked>). Acesso em: 04/10/2018.

SOUZA, D. ***Checked Exceptions versus Unchecked Exceptions:*** trabalhando com exceções. Portal **Devmedia**, 2013a. Disponível em: <<https://www.devmedia.com.br/checked-exceptions-versus-unchecked-exceptions-trabalhando-com-excecoes/29626>>
<<https://www.devmedia.com.br/checked-exceptions-versus-unchecked-exceptions-trabalhando-com-excecoes/29626>> (<https://www.devmedia.com.br/checked-exceptions-versus-unchecked-exceptions-trabalhando-com-excecoes/29626>). Acesso em: 04/10/2018.

SOUZA, D. Diferença entre os *patterns* PO, POJO, BO, DTO e VO. **Portal Devmedia**, 2013b. Disponível em: <<https://www.devmedia.com.br/diferenca-entre-os-patterns-po-pojo-bo-dto-e-vo/28162>> (<https://www.devmedia.com.br/diferenca-entre-os-patterns-po-pojo-bo-dto-e-vo/28162>). Acesso em: 04/10/2018.

TOSIN, C. E. G. Explorando o *NetBeans 7.0*. Revista **Java Magazine**, ed. 91. Publicado em Portal **Devmedia**, 2011. Disponível em: <<https://www.devmedia.com.br/explorando-o-netbeans-7-0-artigo-java-magazine->>

91/21121)<https://www.devmedia.com.br/explorando-o-netbeans-7-0-artigo-javamagazine-91/21121> (https://www.devmedia.com.br/explorando-o-netbeans-7-0-artigo-javamagazine-91/21121)>. Acesso em: 04/10/2018.

