



UAST
Unidade Acadêmica
de Serra Talhada - PE
Desde 2006



MPOO

Site: <https://sites.google.com/site/profricodemery/mpoo>

Site: <http://ava.ufrpe.br/>

Disciplina: Modelagem e Programação Orientada a Objetos (MPOO)

Profº: Richarlyson D'Emery

LISTA DE EXERCÍCIOS IX

Leia atentamente as instruções gerais:

- No Eclipse crie um novo projeto chamado **br.edu.mpoo.listaIX.SeuNomeSobrenome**, o qual deverá ter **pastas de pacotes** para cada questão: **questao1**, **questao2**, e assim sucessivamente, contendo todas as respostas da lista.
- Quando a questão envolver uma discussão teórica utilize um arquivo **.txt** (Menu File -> Submenu New -> Opção File), por exemplo, **questao5.1.txt**
- A lista envolve questões práticas e conceituais, então deverão ser entregues no AVA tanto os códigos-fonte (projeto completo) quanto às demais respostas.
- A entrega da lista compõe sua frequência e avaliação na disciplina.

Fique atento!

Prezado aluno, esta é a lista de exercícios é relativa à Semana 10 em que vimos, em especial, o conceito de “Manipulação de Exceções”. Por isso, antes de responder, reveja a vídeo-aula. Também aprenda novas interfaces para “Tratamento de Eventos”!

Você sabia?

Métodos podem usar **Exceções** para tratar eventuais situações não esperadas e pensadas quando utilizados em aplicações. Ou seja: *“Aconteceu algo de errado!”*.

A manipulação de exceções em Java é uma maneira simples de lidar com situações excepcionais que podem surgir em tempo de execução e dar o devido tratamento a essas situações.

Há comandos clássicos que já se sabe que podem ocasionar um erro, como, por exemplo: a transferência de dados que utiliza uma conexão de Internet pode falhar se esta “cair”; ou não conseguir manipular o dado de um objeto por este não ter sido devidamente instanciado; sem querer apaguei os arquivos de imagem do meu *java game* então como os personagens terão aparência? ... e muitas outras situações...

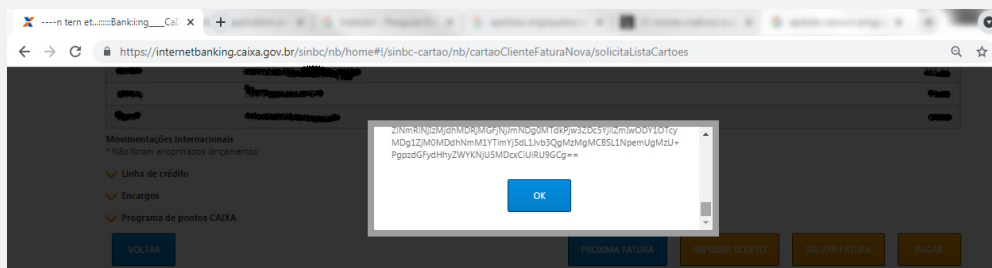
*“Detran-PE, seu contratado esqueceu de aprender **Exception**, rsrsrs”;*

“CEF, obrigado pelo incômodo, rsrsrs”

Extrato de Débitos <http://online4.detran.pe.gov.br/ServicesWeb/Veiculo/frmDetalhamentoDebitos.aspx>

System.NullReferenceException: Object reference not set to an instance of an object. at frmDebito.Page_Load(Object sender, EventArgs e) in c:\inetpub\wwwroot\ServicesWeb/Veiculo/frmDetalhamentoDebitos.aspx.cs:line 47

Último CRLV Emitido - Exercício: Label	
Emissão:	
Entrega:	
Destino:	Label
Retirante:	Label
Devolução:	Label
Motivo:	Label



(Mas, que 🐱 é isso? São exemplos clássicos de não tratamento para situações inesperadas e do que **NÃO** mostrar ao usuário!)

O tratamento de exceções ajuda a aprimorar a tolerância a falhas de um programa. É uma espécie de prevenção de Erro. Com o tratamento de exceções, um programa pode continuar executando (em vez de encerrar) depois de lidar com um problema. Antigamente, linguagens de programação como C não possuíam mecanismos de tolerância a falha e quando ocorria qualquer problema tínhamos o encerramento do programa.



Por isso **try-catch** usa a premissa: TENTE executar, mas em caso de não dar certo, não se esqueça de PREVER qualquer problema. Um bloco **try-catch** informará ao compilador que você sabe que algo excepcional pode ocorrer no método que está sendo chamado e que você está preparado para manipulá-lo. Toda situação excepcional é um tipo de **Exception**.

Mas atenção: para as coisas que você quiser fazer independente do que ocorra, utilize o **finally**, ou seja, um bloco **finally** é onde você insere um código que deve ser executado independente de uma exceção lançada. Por exemplo:

- durante uma conexão com o banco de dados para realizar determinada ação ocorre uma exceção (NullPointerException ao tentar manipular um atributo inexistente de um objeto), neste caso seria necessário que mesmo sendo lançada uma exceção no meio do processo a conexão fosse fechada;
- durante a escrita de um arquivo é lançada uma exceção por algum motivo (um arquivo não seria fechado), o que resultaria em deixar o arquivo aberto (falha de segurança).

Vejamos a sintaxe:

```
try{
    //tente executar o código:
    //comando1
    //comando2
    //comando...
    //comandoN
} catch(Exception ex){
    //algo deu errado, então vou tratar aqui!
} finally{
    //não quero saber o que pode acontecer: esse bloco será executado!
}
```

Mas atenção:

- se **comando1** falhar então todos comandos subsequentes do bloco **try** não serão executados, pulando para o **catch**.
- **comandos** podem conter diferentes tipos de situações, portanto admite-se vários **catch**'s: aglutinados por | (or) para um tratamento único (multi-catch) ou em diferentes tratamentos:

```
//opção1: multi-catch
try {
    app.execute();
} catch(IOException | NullPointerException | ArithmeticException | IndexOutOfBoundsException ex){
    ex.printStackTrace();
    JOptionPane.showMessageDialog(null, "Bugou, o app será encerrado!");
    System.exit(0);
}

//opção2: catch's distintos:
try {
    app.execute();
} catch(IOException ex){
    //Tratamento específico
} catch(NullPointerException ex){
    //Tratamento específico
} catch(ArithmeticException ex){
    //Tratamento específico
} catch(IndexOutOfBoundsException ex){
    //Tratamento específico
}
```

Colocar inúmeras exceções em multi-catch? Se for para qualquer coisa, então as trate polimorficamente, substituindo-as por apenas **Exception**.

Mas só porque PODE capturar tudo com um superbloco catch polimórfico, não quer dizer que sempre DEVA fazê-lo. Pergunte-se: Tratar o QUE do QUÊ? O bloco **catch (Exception ex){ }** capturar **QUALQUER** exceção, portanto, você não saberá automaticamente o que deu errado!

E não se esqueça de deixar o programador ciente de que uma exceção ocorreu, ou seja, "mostrar aquele erro no console". Para isso utilize o método **printStackTrace()**.

Outra possibilidade é a utilização da cláusula **throws** na declaração de um método. Logo, quando já se sabe na definição de um método que uma situação inesperada possa acontecer então defina uma exceção para ele, garantido ao compilador que foram tomadas as devidas precauções.

```
//FalhaException.java
public class FalhaException extends Exception{
    public FalhaException (String mensagem){
        super(mensagem);
    }
}

//OutraFalhaException.java
public class OutraFalhaException extends Exception{
    public OutraFalhaException (String mensagem){
        super(mensagem);
    }
}

//Classe.java
public class Classe {
    public void podeDarFalha(Object object) throws FalhaException, OutraFalhaException{
        if(object==null)
            throw new FalhaException("Deu Falha.");
        else
            throw new OutraFalhaException("Falha sem solução.");
    }
}

//App.java
public class App {
    public static void main(String[] args) {
        try {
            new Classe().podeDarFalha(null); //Ex. parâmetro: null or new Object()
        } catch (FalhaException e) {
            e.printStackTrace();
            //correção para falha
        } catch (OutraFalhaException e) {
            e.printStackTrace();
            JOptionPane.showMessageDialog(null, e.getMessage() + "O sistema será
finalizado");
        }
        System.exit(0);
    }
}
```

Diferentes exceções podem ser definidas, desde que separadas por vírgula.

Exemplo hipotético!
Pode-se utilizar
`NullPointerException`

Antes de definir uma Exceção, verifique se já há uma exceção definida em Java. Conheça as subclasses da hierarquia de herança da classe `java.lang.Throwable`: **Error** e **Exception**

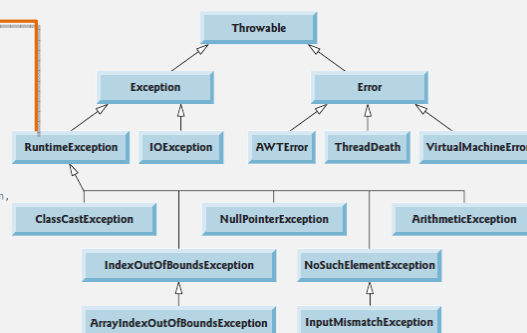
<https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html>

java.lang
Class RuntimeException
java.lang.Object
java.lang.Throwable
java.lang.Exception
java.lang.RuntimeException

All Implemented Interfaces:
Serializable

Direct Known Subclasses:

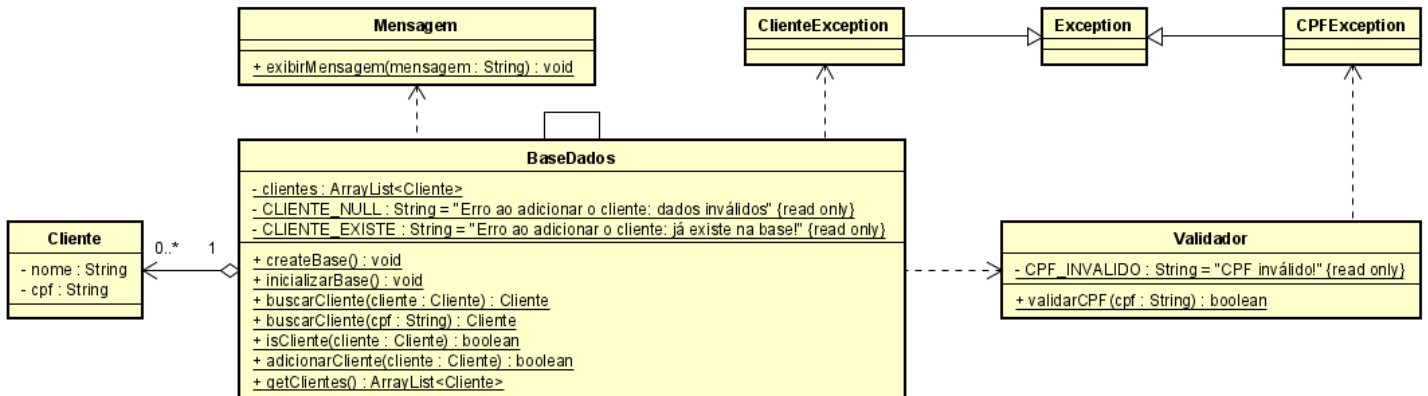
AnnotationTypeMismatchException, ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMMException, CompletionException, ConcurrentModificationException, DataBindingException, DateTimeException, DOMException, EmptyStackException, EnumConstantNotPresentException, EventException, FileSystemAlreadyExistsException, FileSystemNotFoundException, IllegalArgumentException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, IllformedLocaleException, ImagingOpException, IncompleteAnnotationException, IndexOutOfBoundsException, JMSRuntimeException, LSEException, MalformedURLException, MalformedParametersException, MirroredTypesException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, NoSuchMechanismException, NullPointerException, ProfileDataException, ProviderNotFoundException, RasterFormatException, RejectedExecutionException, SecurityException, SystemException, TypeConstraintException, TypeNotPresentException, UncheckedIOException, UndeclaredThrowableException, UnknownEntityException, UnmodifiableSetException, UnsupportedOperationException, WebServiceException, WrongMethodTypeException



Em Java trata-se **Exception**. A classe **Error** e suas subclasses representam situações anormais que acontecem na JVM e não acontecem frequentemente, logo não devem ser capturados pelas aplicações — normalmente não é possível que aplicativos se recuperem de **Error**'s.

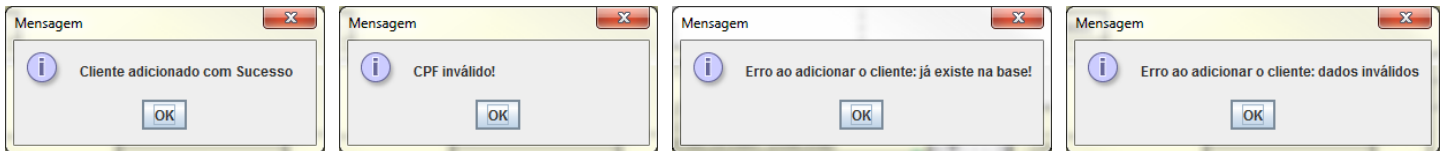
Mão na Massa!

1) Analise o diagrama de classes abaixo e as regras de negócio:



São Regras de negócios:

- RN01 – As operações de gerenciamento de um cliente na base devem verificar se um cliente e seu cpf são válidos.
- RN02 – A codificação deve aproveitar comportamentos já definidos, evitando a duplicidade de programação;
- RN03 – Confirmações devem exibir mensagens para o usuário, por exemplo:



Problema 1: O método abaixo valida um cpf, portanto modifique-o de maneira a levantar a exceção `CPFException` quando um cpf não for válido.

```

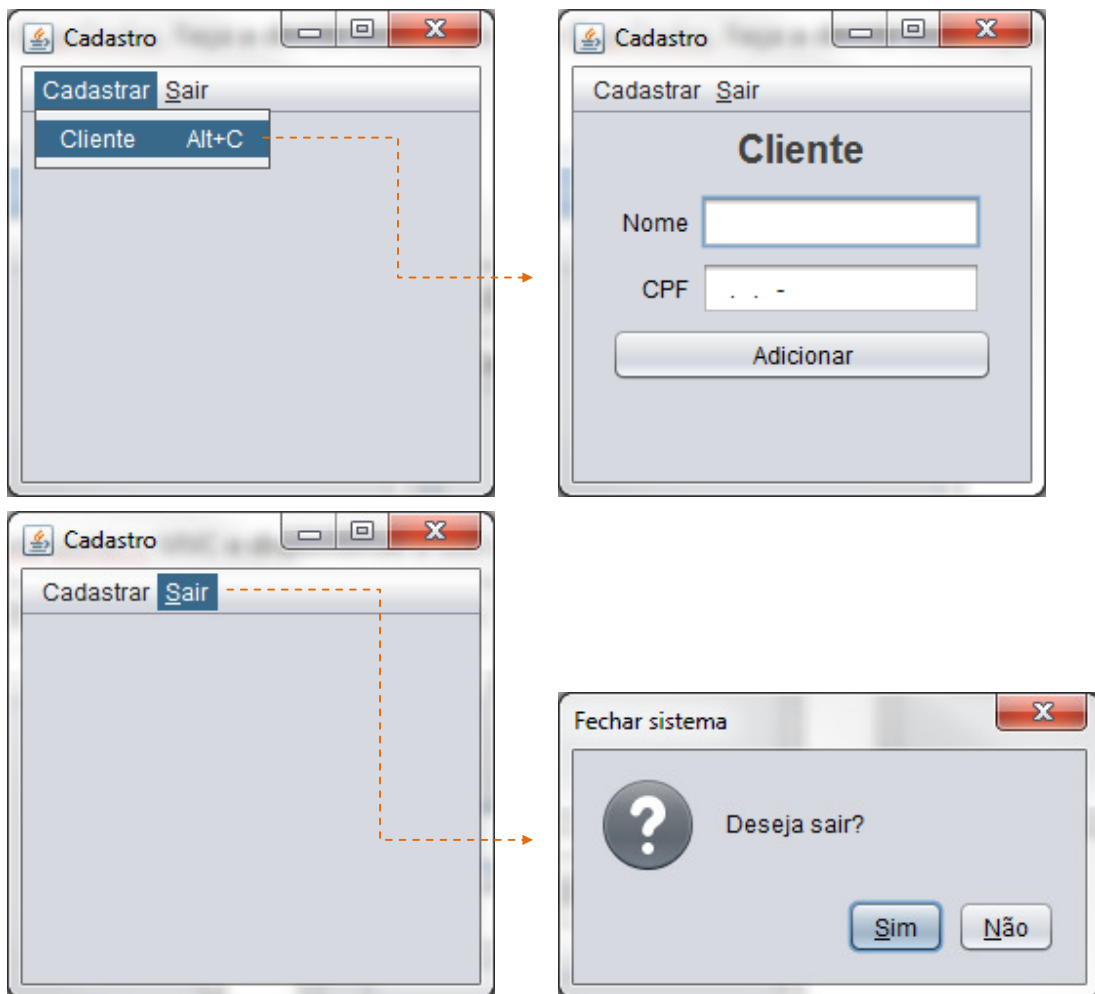
public boolean validarCPF(String CPF) {
    if (CPF.equals("00000000000") || CPF.equals("11111111111") ||
        CPF.equals("22222222222") || CPF.equals("33333333333") ||
        CPF.equals("44444444444") || CPF.equals("55555555555") ||
        CPF.equals("66666666666") || CPF.equals("77777777777") ||
        CPF.equals("88888888888") || CPF.equals("99999999999") ||
        (CPF.length() != 11)) return (false);
    char dig10, dig11;
    int sm, i, r, num, peso;
    try {
        sm = 0;
        peso = 10;
        for (i=0; i<9; i++) {
            num = (int)(CPF.charAt(i) - 48);
            sm = sm + (num * peso);
            peso = peso - 1;
        }
        r = 11 - (sm % 11);
        if ((r == 10) || (r == 11)) dig10 = '0';
        else dig10 = (char)(r + 48);
        sm = 0;
        peso = 11;
        for(i=0; i<10; i++) {
            num = (int)(CPF.charAt(i) - 48);
            sm = sm + (num * peso);
            peso = peso - 1;
        }
        r = 11 - (sm % 11);
        if ((r == 10) || (r == 11)) dig11 = '0';
        else dig11 = (char)(r + 48);
        if ((dig10 == CPF.charAt(9)) && (dig11 == CPF.charAt(10))) return (true);
        else return (false);
    } catch (InputMismatchException erro) {
        return (false);
    }
}
    
```

Problema 2: Codifique em Java o diagrama de classes, respeitando as regras de negócio definidas. Faça o devido uso do bloco **try - catch** na chamada de métodos e **throw** e **throws** para o tratamento das exceções **CPFException** e **ClienteException**.

Problema 3: Crie uma App, na qual:

- Possui três clientes, dos quais:
 - Um tem cpf válido (utilizar o seu cpf);
 - Um não tem cpf válido;
 - Um cliente é null
- Ilustra a utilização dos métodos de BaseDados. Faça a devida associação com a RN03.

2) Para a questão anterior, adote o *architectural pattern* MVC e disponibilize a GUI abaixo para cadastrar clientes.



Problema 1: Observe que a GUI utiliza uma máscara para o CPF, enquanto o método `validarCPF` recebe como parâmetro uma `String` de maneira a não conter "." e "-". Sendo assim, proponha uma solução no controlador de maneira a remover os pontos e o hífen para o método de validação de CPF.

Saiba Mais

Para o tratamento de eventos de menus de um `JFrame` utilize a interface `ActionListener` para `JMenuItem` e `MouseListener` para `JMenu`.

Para saber mais como utilizar componentes de menu e para seus respectivos tratamentos e eventos, acesse o tutorial de JavaSE da Oracle disponível em: <https://docs.oracle.com/javase/tutorial/uiswing/components/menu.html>

- 3) A validação do CPF está sendo realizada pela classe BaseDados através de Validador, gerando um processamento desnecessário quando submetido os dados de um cliente após o acionamento do botão “Adicionar”. Modifique o diagrama de classes de maneira que a validação do cpf seja realizada no controlador. Também adicione uma dica para o campo Nome de maneira que o usuário saiba que deve ser informado o “nome Completo”. Ao clicar neste campo a dica desaparece. A liberação do botão “Adicionar” é dada apenas quando o usuário fornecer um Nome e um CPF válido. **Antes de responder a essa questão, estude a próxima seção Saiba Mais!**

Dica: Além de CaretListener e FocusListener também utilize KeyListener.

The figure displays six sequential screenshots of a Java Swing window titled "Cadastro". The window contains a title bar with standard OS controls, a menu bar with "Cadastrar" and "Sair", and a main content area titled "Cliente". The form has two text input fields: "Nome" and "CPF", and a button labeled "Adicionar".

- Screenshot 1 (Top Left):** The "Nome" field contains the placeholder text "Nome Completo". The "CPF" field is empty.
- Screenshot 2 (Top Right):** The "Nome" field is empty, and the "CPF" field contains the placeholder text "... -".
- Screenshot 3 (Middle Left):** The "Nome" field contains the text "Beto Silva". The "CPF" field contains the text "111.111.111-11".
- Screenshot 4 (Middle Right):** The "Nome" field contains the text "Beto Silva". The "CPF" field contains the text "599.518.835-67".
- Screenshot 5 (Bottom Left):** The "Nome" field contains the text "Beto Silva". The "CPF" field contains the text "599.518.835-6".
- Screenshot 6 (Bottom Right):** The "Nome" field is empty, and the "CPF" field contains the text "599.518.835-67".

(*) Não é intenção do exemplo expor ou violar dados pessoais de terceiros. Para o CPF válido do exemplo, utilizou-se o gerador de cpf por <https://www.geradordecpf.org/>

Saiba Mais!

Você sabe como mudar a posição do cursor entre componentes de texto em uma GUI? A resposta é [CaretListener](#)! Com esta interface podemos por, exemplo, saber ou modificar os campos de JTextArea ou JTextField sem que o usuário tenha digitado.

E se a pergunta agora fosse: Você sabe como mudar o foco de uma GUI? Se sua resposta é não, vamos analisar algumas situações:

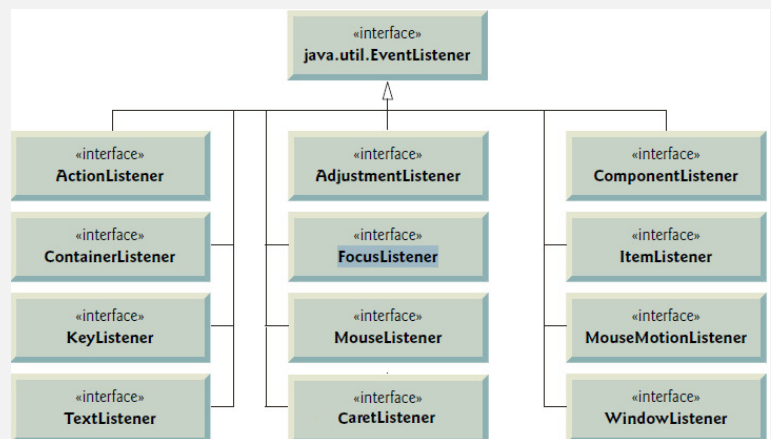
Focus em componentes por método de Component

É possível direcionar o foco para um componente gráfico, como, por exemplo, para um JLabel tem-se:

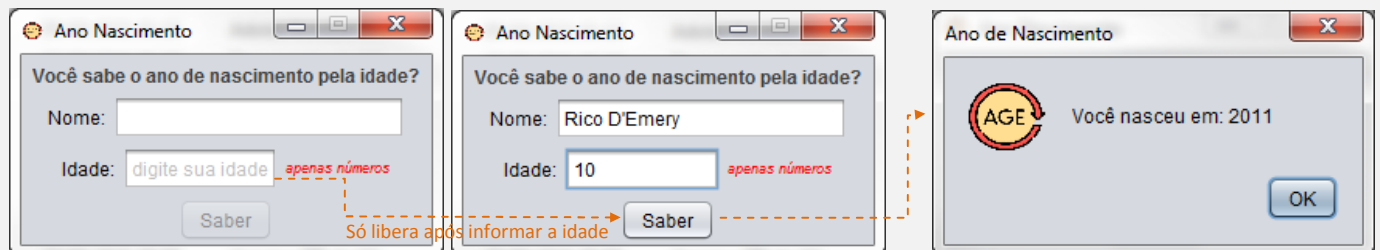
```
label.setFocusable(true);
```

Focus por Interface

Para cada tipo de objeto de evento, há em geral uma [interface listener](#) de eventos correspondentes. Um ouvinte de evento para um evento GUI é um objeto de uma classe que implementa uma ou mais das interfaces ouvintes de evento dos pacotes java.awt.event e javax.swing.event. Muitos dos tipos de ouvinte de evento são comuns aos componentes Swing e AWT. Esses tipos são declarados no pacote java.awt.event Os tipos de ouvinte de evento adicionais que são específicos dos componentes Swing são declarados no pacote javax.swing.event. Conheça outras especializações de EventListener em: <https://docs.oracle.com/javase/10/docs/api/java/util/EventListener.html>



Uma situação corriqueira de GUI's: Um JFrame contendo JLabel e JTextField o foco ficará no JTextField, ou seja, o campo de entrada de dados ficará ativo neste componente! Para que se possa mudar o foco podemos utilizar a interface [FocusListener](#). Vejamos um exemplo:



Continua na próxima página....

```
// Icone.java
package sistema.model;

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.ImageIcon;

public class Icone {
    private File file;
    private BufferedImage bufferImage;
    private ImageIcon icon;
    public static final String URL_ICONE = "img/age.png";

    public Icone(String urlFile) throws IOException{
        try {
            file = new File(urlFile);
            bufferImage = ImageIO.read(file);
            icon = new ImageIcon(bufferImage);
        } catch (IOException e) {
            e.printStackTrace();
            throw e;
        }
    }

    public ImageIcon getIcone() { return icon; }
}

//Mensagem.Java
package sistema.view;

import java.awt.HeadlessException;
import java.io.IOException;
import javax.swing.JOptionPane;
import sistema.model.Icone;

public class Mensagem extends JOptionPane {
    public final static String MENSAGEM_FALHA = "Erro ao carregar imagem. O sistema será encerrado!";

    public static void exibirMensagemAniversario(String mensagem, String urlIcone) throws
HeadlessException, IOException{
        try {
            showMessageDialog(null, mensagem, "Ano de Nascimento", JOptionPane.OK_OPTION, new
Icone(urlIcone).getIcone());
        } catch (HeadlessException | IOException e) {
            e.printStackTrace();
            throw e;
        }
    }

    public static void exibirMensagemFalha(String mensagem){
        showMessageDialog(null, mensagem, "Erro sistema", JOptionPane.ERROR_MESSAGE);
    }
}

//Continua na próxima página...
```

Dica MVC versus Exception:

- É papel do controller tratar Exceptions .
- sistema.model.Icone foi definido para respeitar o tratamento de Exceção quando uma imagem a ser utilizada não está disponível.
- Evite tratar exceções em model e view.
- Comumente tratamento de exceções exibem *feedback* ao usuário.


```

//Tela.java
package sistema.view;

import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.Font;
import java.io.IOException;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

public class Tela extends JFrame {
    JLabel tituloLabel, nomeLabel, idadeLabel, infoLabel;

    JTextField nomeField, idadeField;
    JButton saberButton;

    public Tela(ImageIcon icone) throws IOException{
        super("Ano Nascimento");
        setSize(270, 150);
        setResizable(false);
        setLayout(new FlowLayout());
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setIconImage(icone.getImage());
        tituloLabel = new JLabel("Você sabe o ano de nascimento pela idade?", JLabel.TRAILING);
        tituloLabel.setFont(new Font("", Font.BOLD, 12));
        tituloLabel.setForeground(Color.DARK_GRAY);
        tituloLabel.setFocusable(true); //foco no título
        nomeLabel = new JLabel("Nome:");
        nomeField = new JTextField(16);
        idadeLabel = new JLabel("Idade:");
        idadeField = new JTextField("digite sua idade", 8);
        idadeField.setForeground(Color.LIGHT_GRAY);
        infoLabel = new JLabel("apenas números");
        infoLabel.setFont(new Font("", Font.ITALIC, 9));
        infoLabel.setForeground(Color.RED);
        saberButton = new JButton("Saber");
        saberButton.setEnabled(false); //botão inativo
        add(tituloLabel);
        add(nomeLabel);
        add(nomeField);
        add(idadeLabel);
        add(idadeField);
        add(infoLabel);
        add(saberButton);
        setVisible(true);
    }

    public JTextField getIdadeField() { return idadeField; }

    public JButton getSaberButton() { return saberButton; }
}

//Continua na próxima página...

```

```

//Controller.java
package sistema.controller;

import java.awt.Color;
import java.awt.HeadlessException;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.FocusEvent;
import java.awt.event.FocusListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.io.IOException;
import java.time.LocalDateTime;
import javax.swing.event.CaretEvent;
import javax.swing.event.CaretListener;
import sistema.model.Icone;
import sistema.view.Mensagem;
import sistema.view.Tela;

public class Controller{
    KeyHandler keyHandler;
    FieldHandler fieldHandler;
    ButtonHandler buttonHandler;
    Tela tela;
    Icone icone;
    public Controller(){
        keyHandler = new KeyHandler();
        fieldHandler = new FieldHandler();
        buttonHandler = new ButtonHandler();
        try {
            icone = new Icone(Icone.URL_ICONE);
            tela = new Tela(icone.getIcone());
        } catch (IOException e) {
            e.printStackTrace();
            Mensagem.exibirMensagemFalha(Mensagem.MENSAGEM_FALHA);
            System.exit(0);
        }
        control();
    }

    private void control(){
        tela.getIdadeField().addKeyListener(keyHandler);
        tela.getIdadeField().addCaretListener(fieldHandler);
        tela.getIdadeField().addFocusListener(fieldHandler);
        tela.getSaberButton().addActionListener(buttonHandler);
    }

    private class KeyHandler extends KeyAdapter{
        @Override
        public void keyTyped(KeyEvent event) {
            if (!Character.isDigit(event.getKeyChar()))
                event.consume();
        }
    }

    private class FieldHandler implements CaretListener, FocusListener{
        @Override
        public void caretUpdate(CaretEvent e) {
            if (tela.getIdadeField().getText().length()!=0 &&
                !tela.getIdadeField().getText().contains("digite sua idade"))
                tela.getSaberButton().setEnabled(true);
            else
                tela.getSaberButton().setEnabled(false);
        }
    }
}

```

Tratamento de Exception no controller:
possibilidade de inexistência de imagem

Em caretUpdate é verificado se o usuário digitou
uma idade válida. Se sim, libera o botão Saber, caso
contrário o botão permanece desabilitado.

Verifica se o campo de texto para idade
está sendo utilizado.

- Em focusGained remove o texto dica e configura para uma entrada padrão.
- Se outro campo estiver em uso e uma idade não tiver sido informada, então em focusLost é retomada a configuração inicial do campo de texto para idade.

//Continua na próxima página...

```

@Override
public void focusGained(FocusEvent e) {
    if(e.getSource()==tela.getIdadeField()){
        if(tela.getIdadeField().getText().equals("digite sua idade")) {
            tela.getIdadeField().setText("");
            tela.getIdadeField().setForeground(Color.BLACK);
        }
    }
}

@Override
public void focusLost(FocusEvent e) {
    if(e.getSource()==tela.getIdadeField()){
        if(tela.getIdadeField().getText().equals("")) {
            tela.getIdadeField().setText("digite sua idade");
            tela.getIdadeField().setForeground(Color.LIGHT_GRAY);
        }
    }
}
}

private class ButtonHandler implements ActionListener{
    @Override
    public void actionPerformed(ActionEvent e) {
        int idade, anoAtual, anoNascimento;

        if (e.getSource()==tela.getSaberButton()){
            idade= Integer.parseInt(tela.getIdadeField().getText());
            anoAtual = LocalDateTime.now().getYear();
            anoNascimento = anoAtual-idade;
            try {
                Mensagem.exibirMensagemAniversario("Você nasceu em: " + anoNascimento,
Icone.URL_ICONE);
            } catch (HeadlessException | IOException e1) {
                e1.printStackTrace();
                Mensagem.exibirMensagemFalha(Mensagem.MENSAGEM_FALHA);
                System.exit(0);
            }
        }
    }
}

}

//App.java
package sistema.app;

import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import sistema.controller.Controller;

public class App {
    public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel("javax.swing.plaf.nimbus.NimbusLookAndFeel");
        } catch (ClassNotFoundException | InstantiationException
            | IllegalAccessException | UnsupportedLookAndFeelException e) {
            e.printStackTrace();
        }
        Controller controller = new Controller();
    }
}

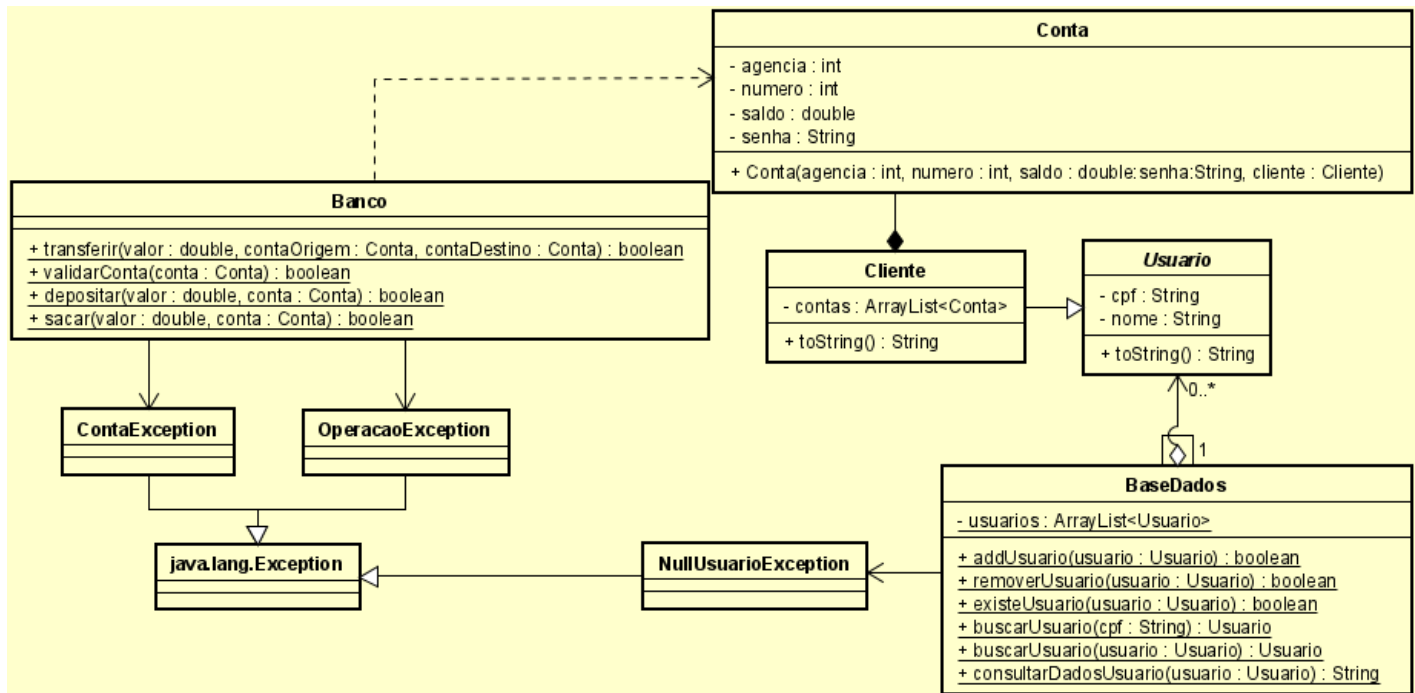
```

Tratamento de evento para o botão Saber: Exibe o ano de nascimento do usuário a partir da idade informada. É considerado o ano corrente do sistema.

Tratamento de Exception no controller: possibilidade de inexistência de imagem

4) Analise os problemas abaixo e codifique-os em Java:

Problema 1: O diagrama de classes abaixo apresenta as dependências entre classes e tratamento de exceção.



Algumas descrições do sistema:

- Cliente e Conta representam um caso de composição. (É permitida a substituição da composição por agregação!).
- As operações no saldo de uma conta são dadas pelos comportamentos de sacar e depositar. O saque só deve ser realizado se o valor do saque for igual ou superior ao valor do saldo.
- Falhas de depósito ou saque devem levantar a exceção OperacaoException, que tem como tratamento a exibição de um JOptionPane com a mensagem: "Erro ao efetuar o saque" ou "Erro ao efetuar o depósito".
- A transferência deverá utilizar as operações de saque e depósito.
- A consulta dos dados de um usuário deve utilizar o método toString().
- Na classe BaseDados, os usuários deverão ser manipulados pelos métodos definidos.
 - Um usuário só deve ser adicionado se não existir na base. Use como critério de existência o cpf do usuário.
 - A verificação da existência de um usuário é dada por `boolean existeUsuario(Usuario usuario) {}`. Utilize o cpf como critério de existência.
 - Observe as diversas maneiras para buscar um usuário: ou pelo cpf do usuário ou por um usuário. Um usuário é identificado pelo seu cpf.

(continua na próxima página)

- O método da classe Banco `public static boolean validarConta(Conta conta) throws ContaException {}` deve levantar a exceção ContaException caso a conta informada não seja uma conta válida. Esse método deverá ser utilizado ao se tentar realizar uma transferência. Faça o devido uso do bloco `try - catch` na chamada do método e `throw` e `throws` para o tratamento da exceção. No tratamento da validação de uma Conta, caso uma conta não seja válida, exiba em JOptionPane a mensagem: "Conta Origem Inválida" ou "Conta Destino Inválida".
- O método existeUsuario de BaseDados deve levantar a exceção NullUsuarioException caso um usuário não pertença a base. Como tratamento deve-se exibir um JOptionPane com a mensagem: "Cliente não existe" se o usuário for do tipo Cliente.
- As contas do banco devem ser auto-increment.

Problema 2:

Crie uma App, na qual:

- Possui os clientes:
 - José, cpf 111.111.111-11.
 - Conta agencia: 1, numero 1, saldo R\$ 0.0, senha: 123.
 - Conta agencia: 1, numero 2, saldo R\$ 100.0, senha: jose123.
 - Maria, cpf 222.222.222-22, conta agencia: 1, numero 3, saldo R\$ 1000.0, senha: Maria222.
- Possui uma transferência de R\$100 de Maria para José.
- Exibe em console os valores antes e depois da transferência.
- Ilustra todas as exceções. Antes de cada comando, adicione um `//comentário` explicando que Exceção está sendo realizada.

Você Sabia?

Você já ouviu falar em **Relançamento de exceção**? Esta é um artifício quando um bloco **catch**, ao receber uma exceção, decide que não pode processar essa exceção ou que só pode processá-la parcialmente. Relançar uma exceção adia o tratamento de exceções (ou parte dele) para outro bloco **catch** associado com uma instrução **try** externa. Uma exceção é relançada utilizando-se a palavra-chave **throw**, seguida por uma referência ao objeto de exceção que acabou de ser capturado. Observe que as exceções não podem ser relançadas a partir de um bloco **finally**, uma vez que o parâmetro de exceção (uma variável local) do bloco **catch** não mais existe.

Quando ocorre um relançamento, o próximo bloco **try** circundante detecta a exceção relançada e os blocos **catch** desse bloco **try** tentam tratá-la.

É erro comum de programação versus prevenção de erro:



- **Erro:** Se uma exceção não tiver sido capturada quando o controle entrar em um bloco **finally** e esse bloco lançar uma exceção que não será capturada por ele, a primeira exceção será perdida e a exceção do bloco será retornada ao método chamador.
- **Prevenção:** Evite colocar código que possa lançar (**throw**) uma exceção em um bloco **finally**. Se esse código for necessário, inclua o código em um bloco **try-catch** dentro do bloco **finally**.

Desafio!

5) **(Desafio):** Analise a codificação:

```
1. public class UsandoExcecao{
2.     public static void main( String[] args ){
3.         try{
4.             LevantarExcecao();
5.         }
6.         catch (Exception exception){
7.             System.err.println("Exceção capturada e tratada no main");
8.         }
9.         naoLevantarExcecao();
10.    }
11.
12.    public static void levantarExcecao() throws Exception{
13.        try{
14.            System.out.println("Método levantarExcecao:");
15.            throw new Exception();
16.        }
17.        catch (Exception exception){
18.            System.err.println("Exceção capturada no próprio método levantarExcecao");
```

```

19.         throw exception;
20.     }
21. }
22.
23. public static void naoLevantarexecao(){
24.     try {
25.         System.out.println("Método que não levanta exceção");
26.     }
27.     catch (Exception exception){
28.         System.err.println(exception);
29.     }
30.     System.out.println("Fim do método nãoLevantarexecao");
31. }
32. }

```

Responda:

5.1) Aponte onde ocorreu o relançamento de exceção. Explique-o

5.2) O que aconteceria se o método levantarExcecao() estivesse assim definido:

```

public static void levantarExcecao() throws Exception{
    try{
        System.out.println("Método levantarExcecao:");
        throw new Exception();
    }
    catch (Exception exception){
        System.err.println("Exceção capturada no próprio método levantarExcecao");
        throw exception;
    }
    finally{
        System.out.println("Sempre executado");
        throw new Exception();
    }
}

```

5.3) Qual diferença do método levantarExcecao() abaixo com o definido na questão 3.2)

```

public static void levantarExcecao() throws Exception{
    try{
        System.out.println("Método levantarExcecao:");
        throw new Exception();
    }
    catch (Exception exception){
        System.err.println("Exceção capturada no próprio método levantarExcecao");
        throw exception;
    }
    finally{
        System.out.println("Sempre executado");
        try{
            throw new Exception();
        }
        catch (Exception exception){
            System.err.println("Exceção capturada");
        }
    }
}

```

5.4) Pesquise e explique quando devemos utilizar os métodos System.out.println e System.err.println().