



TRATAMENTO DE EXCEÇÕES

FAPESC – DESENVOLVEDORES PARA TECNOLOGIA DA INFORMAÇÃO

HERCULANO DE BIASI

herculano.debiasi@unoesc.edu.br



TÓPICOS

- *Multicatch*
- Lançando exceções
- Exceções verificadas (*checked*)
- Exceções não verificadas (*unchecked*)
- Propagando exceções
- Criando exceções personalizadas
- Boas práticas

MULTICATCH

- É possível realizar o mesmo tratamento para classes de exceções diferentes de forma mais concisa através do recurso de *multicatch* usando o operador *|* (pipe)

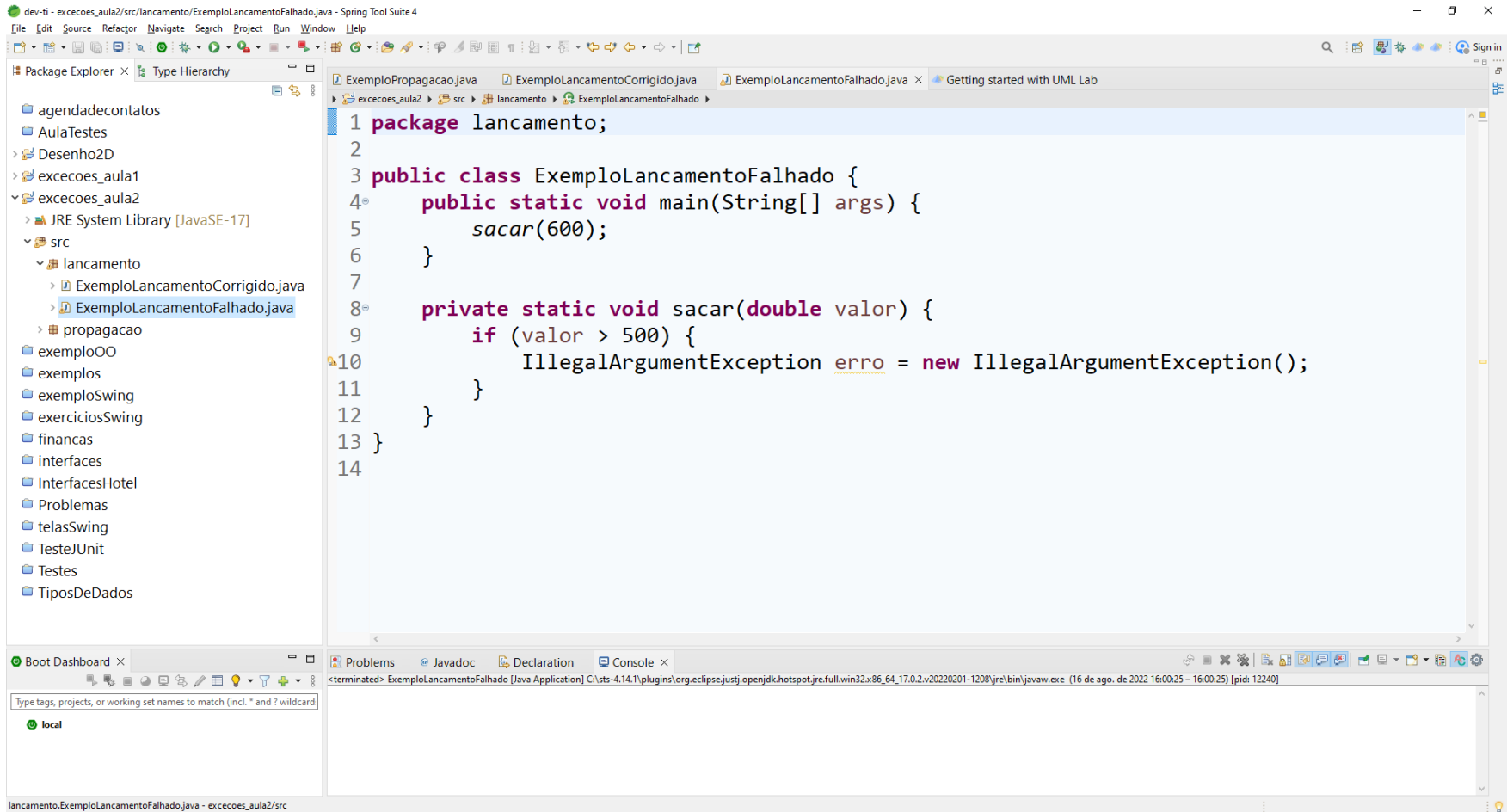
```
modulo2 - ExemploMulticatch.java

1 package problemas;
2
3 import java.util.InputMismatchException;
4 import java.util.Locale;
5 import java.util.Scanner;
6
7 public class ExemploMulticatch {
8     public static void main(String[] args) {
9         try {
10             Locale.setDefault(Locale.US);
11             Scanner ler = new Scanner(System.in);
12             System.out.print("Digite um número com parte decimal: ");
13             double numero = ler.nextDouble(); // Dependendo do Locale dará problema com , ou .
14                                             // gerando a exceção InputMismatchException
15             System.out.println(numero);
16
17             numero = Double.parseDouble("10,5"); // Levanta a exceção NumberFormatException
18         } catch (InputMismatchException | NumberFormatException e) {
19             System.out.println("Entrada/formato de número inválido");
20         }
21
22         System.out.println("Programa finalizado");
23     }
24 }
```

LANÇANDO EXCEÇÕES

■ Exceções são classes convencionais Java, logo podem ser instanciadas

■ Mas isso não basta para lançar a exceção



The screenshot shows an IDE window with the following components:

- Package Explorer:** Displays a project structure with packages like `agendacontatos`, `AulaTestes`, `Desenho2D`, `excecoes_aula1`, `excecoes_aula2` (containing `JRE System Library [JavaSE-17]` and `src`), `propagacao`, `exemploOO`, `exemplos`, `exemploSwing`, `exerciciosSwing`, `financas`, `interfaces`, `InterfacesHotel`, `Problemas`, `telasSwing`, `TesteJUnit`, `Testes`, and `TiposDeDados`.
- Editor:** Shows the source code of `ExemploLancamentoFalhado.java` in the `lancamento` package. The code is as follows:

```
1 package lancamento;
2
3 public class ExemploLancamentoFalhado {
4     public static void main(String[] args) {
5         sacar(600);
6     }
7
8     private static void sacar(double valor) {
9         if (valor > 500) {
10             IllegalArgumentException erro = new IllegalArgumentException();
11         }
12     }
13 }
14
```
- Console:** Displays a terminated message: `<terminated> ExemploLancamentoFalhado [Java Application] C:\sts-4.14.1\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.2.v20220201-1208\jre\bin\javaw.exe (16 de ago. de 2022 16:00:25 - 16:00:25) [pid: 12240]`.
- Boot Dashboard:** Shows a search bar with the text "Type tags, projects, or working set names to match (incl. * and ? wildcard)" and a "local" button.

LANÇANDO EXCEÇÕES

- Para lançar um erro não basta simplesmente instanciar a classe, é necessário lançá-la utilizando o método `throw`



The screenshot shows an IDE window with the following components:

- Package Explorer:** Displays a project structure with packages like `agendacontatos`, `AulaTestes`, `Desenho2D`, `excecoes_aula1`, and `excecoes_aula2`. The `src` directory under `excecoes_aula2` contains a package `lancamento` with files `ExemploLancamentoCorrigido.java` and `ExemploLancamentoFalhado.java`.
- Editor:** Shows the source code of `ExemploLancamentoCorrigido.java`. The code is as follows:

```
1 package lancamento;
2
3 public class ExemploLancamentoCorrigido {
4     public static void main(String[] args) {
5         sacar(600);
6     }
7
8     private static void sacar(double valor) {
9         if (valor > 500) {
10             IllegalArgumentException erro = new IllegalArgumentException();
11             throw erro;
12         }
13     }
14 }
15
```
- Console:** Displays the output of the program, showing a runtime exception:

```
<terminated> ExemploLancamentoCorrigido [Java Application] C:\sts-4.14.1\plugins\org.eclipse.justj.openjdk.hotspot.jre.full\win32.x86_64_17.0.2.v20220201-1208\jre\bin\javaw.exe (16 de ago. de 2022 16:00:46 - 16:00:47) [pid: 34716]
Exception in thread "main" java.lang.IllegalArgumentException
    at lancamento.ExemploLancamentoCorrigido.sacar(ExemploLancamentoCorrigido.java:10)
    at lancamento.ExemploLancamentoCorrigido.main(ExemploLancamentoCorrigido.java:5)
```

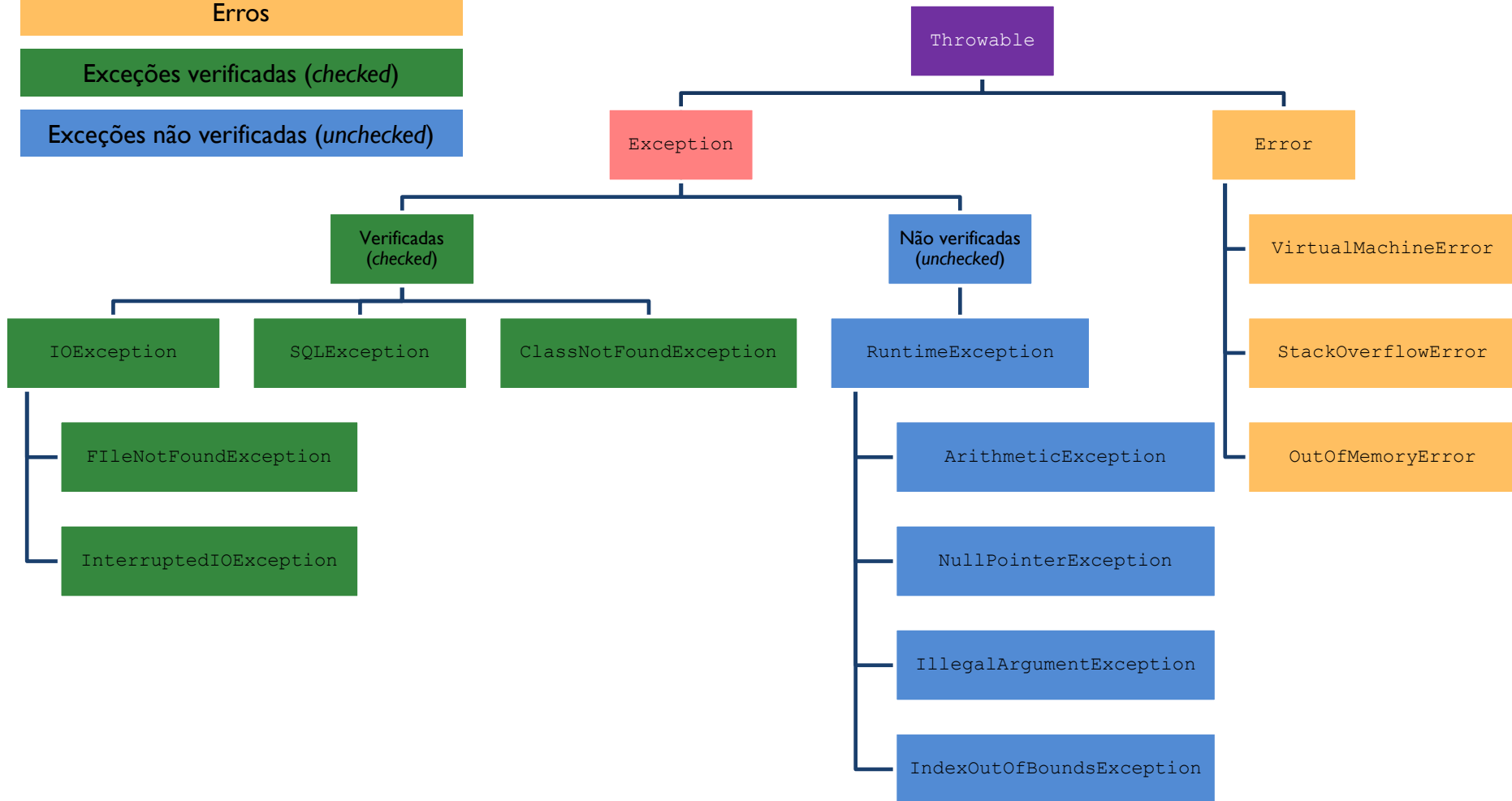
EXCEÇÕES VERIFICADAS VS. NÃO VERIFICADAS

Hierarquia de exceções em Java

Erros

Exceções verificadas (*checked*)

Exceções não verificadas (*unchecked*)



EXCEÇÕES VERIFICADAS (*CHECKED*)

- *Checked exceptions*: Exceções que acontecem fora do controle do programa, mas precisam, obrigatoriamente, ser tratadas dentro do programa
- Precisa obrigatoriamente ser capturada em algum local do código
- Se um método que lance uma exceção verificada for chamado, mas não capturar a exceção em algum local, o código **NÃO** será compilado
- Por isso são chamadas de verificadas – o compilador verifica para se certificar se elas foram declaradas ou tratadas
- Todas as exceções que **NÃO** são derivadas de `java.lang.RuntimeException` são verificadas (*checked*)

EXCEÇÕES NÃO VERIFICADAS (*UNCHECKED*)

- *Unchecked exception (runtime exceptions)*: Exceções que podem ser evitadas se forem tratadas e analisadas pelo desenvolvedor – o compilador não força o programador a capturá-las
- Caso não haja um tratamento para esse tipo de erro, o programa acaba parando em tempo de execução (*runtime*)
- Funcionalmente, exceções verificadas e não verificadas são equivalentes, tudo o que se pode fazer com uma também se pode fazer com a outra, a única diferença é que as do tipo verificadas precisam obrigatoriamente ser capturadas ou propagadas

PROPAGANDO EXCEÇÕES

- Existem situações que o programador não quer tratar a exceção na própria classe ou método que ela aconteça, mas sim no método que o está chamando
- Nessas situações o método atual não tratará a exceção com um bloco `try...catch` e repassará a responsabilidade de tratar a exceção para quem a chamou
- Métodos podem lançar ou propagar exceções para indicar condições de erro – propagar uma exceção é melhor do que apenas retornar um erro porque a exceção contém informações detalhadas sobre o problema
- Quem invoca estes métodos deve capturar e tratar essas possíveis exceções
- Quando uma exceção é lançada, um objeto de um subtipo específico da classe `Exception` é instanciado e inserido no manipulador de exceções como um argumento para a cláusula `catch`
- Todo método deve tratar todas as exceções verificadas fornecendo uma cláusula `catch`, ou então listar cada exceção verificada que não tiver recebido tratamento como uma exceção lançada, regra conhecida como **tratar ou declarar**
- É possível capturar uma exceção verificada com `catch` e propagar outra que não seja verificada com `throw` ou capturar uma exceção qualquer e lançar outra em seu lugar, ou seja, converter uma exceção em outra

PROPAGANDO EXCEÇÕES

- `throw` e `throws`
 - `throw` lança uma exceção
 - `throws` declara uma exceção
 - Se um método não captura (`catch`) a exceção, deve ao menos declará-la, o que é feito através da palavra-chave `throws`
 - **Declarar** significa **repassar** esta `exception` para o método chamador
- Para identificar se um método em particular propaga uma ou mais exceções basta analisar sua assinatura

```
void metodo(int x) throws Excecao1, Excecao2;
```

PROPAGANDO EXCEÇÕES

- Exemplo: Como as exceções são não verificadas, **não é exigido** a declaração com throws, mas é considerada por muitos como uma boa prática

```
modulo2 - ExemploPropagacao.java

1 package propagacao;
2
3 import javax.swing.JOptionPane;
4
5 public class ExemploPropagacao {
6
7     public static void main(String[] args) {
8         String texto = JOptionPane.showInputDialog("Informe uma string");
9
10        try {
11            String reversa = inverter(texto);
12
13            System.out.println("String normal: " + texto);
14            System.out.println("String invertida: " + reversa);
15        } catch (IllegalArgumentException e) {
16            System.out.println(e.getMessage());
17        } catch (NullPointerException e) {
18            System.out.println(e.getMessage());
19        }
20    }
21
22    private static String inverter(String str) throws NullPointerException, IllegalArgumentException {
23        if (str == null) {
24            throw new NullPointerException("Exceção: Valor nulo!");
25        }
26
27        if (str.length() == 0) {
28            throw new IllegalArgumentException("Exceção: String vazia");
29        }
30
31        String inversa = "";
32        for (int i = str.length() - 1; i >= 0; i--) {
33            inversa += str.charAt(i);
34        }
35
36        return inversa;
37    }
38
39 }
```

PROPAGANDO EXCEÇÕES

■ Exemplo da calculadora com propagação de exceção

```
3 public class CalculadoraV2 {
4     public int somar(int num1, int num2) {
5         long resultado = (long) num1 + (long) num2;
6
7         if (resultado > Integer.MAX_VALUE) {
8             throw new IllegalArgumentException("\n*** PROBLEMA DETECTADO ***\n" +
9                                             "Resultado fora da faixa de números inteiros!\n");
10        }
11
12        return num1 + num2;
13    }
14 }
```

PROPAGANDO EXCEÇÕES

Exemplo da calculadora com propagação de exceção

The screenshot shows an IDE window with the following components:

- Package Explorer:** Displays the project structure. The 'src' folder contains 'testesmanuais', which includes 'Calculadora.java', 'CalculadoraTestesV1.java', 'CalculadoraTestesV2.java', 'CalculadoraTestesV3.java', 'CalculadoraTestesV4.java', 'CalculadoraTestesV5.java' (selected), and 'CalculadoraV2.java'.
- Editor:** Shows the code for 'CalculadoraTestesV5.java'. The code is as follows:

```
1 package testesmanuais;
2
3 public class CalculadoraTestesV5 {
4     public static void main(String[] args) {
5         CalculadoraV2 calc = new CalculadoraV2();
6         int soma = 0;
7
8         soma = calc.somar(41, 1);
9         if (soma == 42) {
10             System.out.println("Resultado " + soma + " está correto!");
11         } else {
12             System.out.println("Problema detectado!");
13         }
14
15         try {
16             soma = calc.somar(Integer.MAX_VALUE, 1);
17         } catch (Exception e) {
18             System.out.println(e.getMessage());
19         }
20     }
21 }
22
```
- Console:** Displays the output of the program execution:

```
<terminated> CalculadoraTestesV5 [Java Application] C:\sts-4.14.1\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.2.v20220201-1208\jre\bin\javaw.exe (16 de ago. de 2022 17:08:47 - 17:08:49) [pid: 2420]
Resultado 42 está correto!
*** PROBLEMA DETECTADO ***
Resultado fora da faixa de números inteiros!
```

PROPAGANDO EXCEÇÕES

- Código abaixo não compila pois `Scanner` declara uma exceção verificada (*checked*), `FileNotFoundException` – que não está sendo tratada nem propagada

```
public Scanner(File source) throws FileNotFoundException {
```

- A linha 13 gera também uma advertência já que o recurso `Scanner` não está sendo desalocado da memória

```
1 package exercicios;
2
3 import java.io.File;
4 import java.util.Scanner;
5
6 public class Ex2 {
7
8     public static void main(String[] args) {
9         // Cria um objeto representando o arquivo em disco
10        File arquivo = new File("c:\\temp\\arquivo.txt");
11
12        // Abre o arquivo usando o objeto arquivo
13        Scanner sc = new Scanner(arquivo);
14
15        // Enquanto houver linhas a serem lidas
16        while (sc.hasNextLine()) {
17            // Lê a próxima linha e a mostra na tela
18            System.out.println(sc.nextLine());
19        }
20    }
21
22 }
```

PROPAGANDO EXCEÇÕES

- Solução 1: Tratamento da exceção `FileNotFoundException` com o bloco `try...catch` e liberação do recurso com a utilização do bloco `finally`

```
1 package exercicios;
2
3 import java.io.File;
4 import java.io.FileNotFoundException;
5 import java.util.Scanner;
6
7 public class Ex2SolucaoV1 {
8
9     public static void main(String[] args) {
10         File arquivo = new File("c:\\temp\\arquivo.txt");
11         Scanner sc = null;
12
13         try {
14             sc = new Scanner(arquivo);
15             while (sc.hasNextLine()) {
16                 System.out.println(sc.nextLine());
17             }
18         } catch (FileNotFoundException e) {
19             System.out.println("Erro abrindo arquivo: " + e.getMessage());
20         } finally {
21             if (sc != null) {
22                 sc.close();
23             }
24         }
25     }
26
27 }
```

PROPAGANDO EXCEÇÕES

- Solução 2: Mesmo tratamento da exceção mas liberação do recurso com a nova funcionalidade `try-with-resources` do Java

```
1 package exercicios;
2
3 import java.io.File;
4 import java.io.FileNotFoundException;
5 import java.util.Scanner;
6
7 public class Ex2SolucaoV2 {
8
9     public static void mostraArquivo() {
10         File arquivo = new File("c:\\temp\\arquivo.txt");
11
12         try (Scanner sc = new Scanner(arquivo)) {
13             while (sc.hasNextLine()) {
14                 System.out.println(sc.nextLine());
15             }
16         } catch (FileNotFoundException e) {
17             System.out.println("Erro abrindo arquivo: " + e.getMessage());
18         }
19     }
20
21     public static void main(String[] args) {
22         mostraArquivo();
23     }
24
25 }
```


PROPAGANDO EXCEÇÕES

- Solução 3: Não tratamento da exceção no método `mostraArquivo()` mas com propagação usando `throws` e tratamento na função `main()`

```
1 package exercicios;
2
3 import java.io.File;
4 import java.io.FileNotFoundException;
5 import java.util.Scanner;
6
7 public class Ex2SolucaoV3 {
8
9     public static void mostraArquivo() throws FileNotFoundException {
10         File arquivo = new File("c:\\temp\\arquivo.txt");
11
12         try (Scanner sc = new Scanner(arquivo)) {
13             while (sc.hasNextLine()) {
14                 System.out.println(sc.nextLine());
15             }
16         }
17     }
18
19     public static void main(String[] args) {
20         try {
21             mostraArquivo();
22         } catch (FileNotFoundException e) {
23             System.out.println("Erro abrindo arquivo: " + e.getMessage());
24         }
25     }
26
27 }
```

CRIANDO EXCEÇÕES PERSONALIZADAS

- Um programador pode escrever suas próprias classes de exceção e instanciá-las da mesma maneira que cria e instancia outros objetos do Java
- Exceções personalizadas podem ser do tipo verificadas ou não verificadas, dependendo de qual classe se está herdando

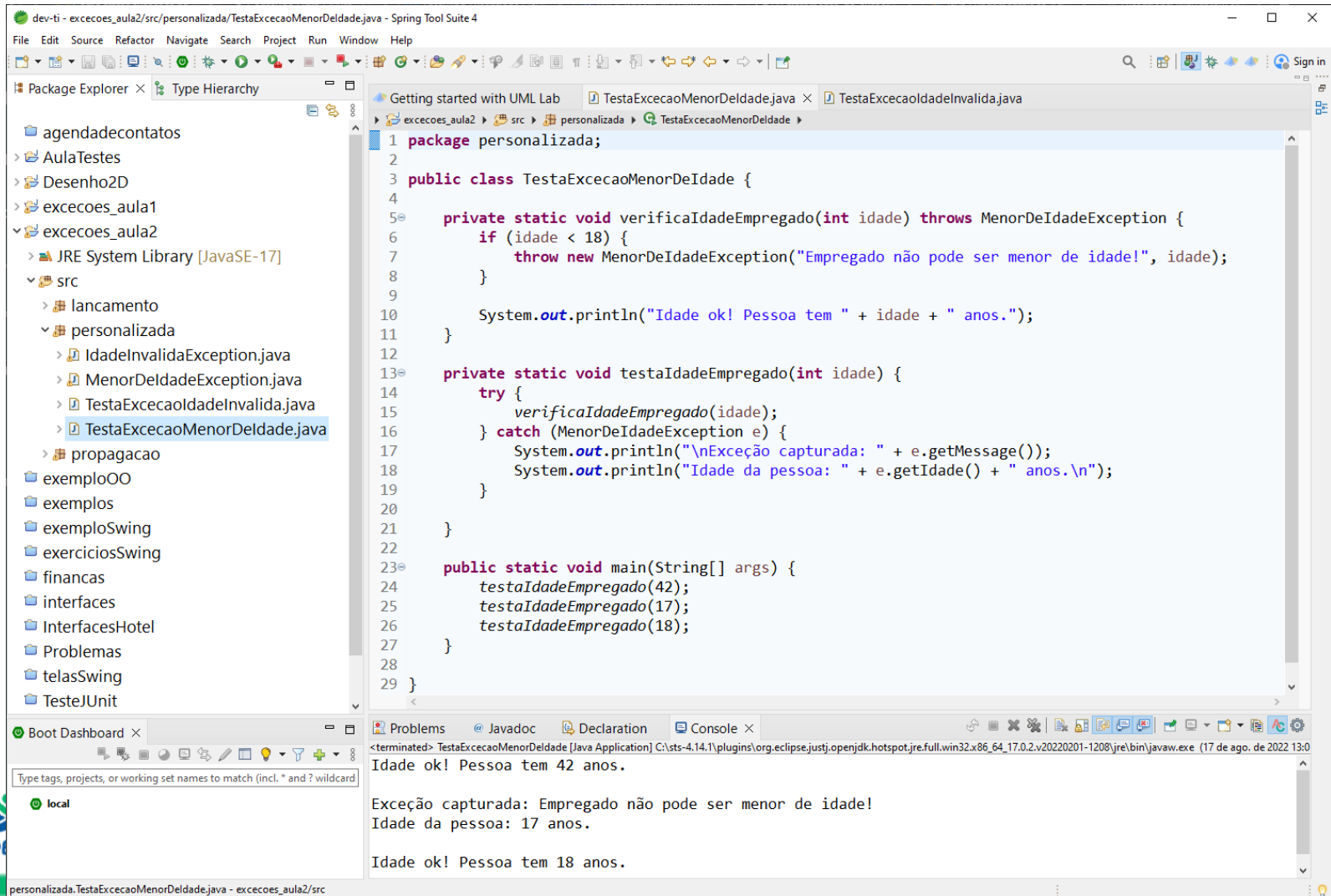
CRIANDO EXCEÇÕES PERSONALIZADAS

- Para demonstrar uma aplicação desse tipo de recurso, suponha que você está desenvolvendo um sistema para uma empresa onde é preciso validar a idade dos seus empregados para realizar a contratação e para isso foram passadas as seguintes regras
- A idade do empregado será válida se ele já tiver 18 anos no dia da contratação
- Empregados com menos de 18 anos devem ser considerados inválidos, pois a empresa não permite contratar trabalhadores menores de idade

```
1 package personalizada;
2
3 public class MenorDeIdadeException extends Exception {
4     private int idade;
5
6     public MenorDeIdadeException(String mensagem, int idade) {
7         super(mensagem);
8         this.idade = idade;
9     }
10
11     public int getIdade() {
12         return idade;
13     }
14 }
```

CRIANDO EXCEÇÕES PERSONALIZADAS

Aplicando



The screenshot displays the Spring Tool Suite 4 IDE. On the left, the Package Explorer shows the project structure, with the file `TestaExcecaoMenorDeldade.java` selected under the `personalizada` package. The main editor shows the code for this class, which defines a custom exception `MenorDeIdadeException` and a `main` method to test it.

```
1 package personalizada;
2
3 public class TestaExcecaoMenorDeIdade {
4
5     private static void verificaIdadeEmpregado(int idade) throws MenorDeIdadeException {
6         if (idade < 18) {
7             throw new MenorDeIdadeException("Empregado não pode ser menor de idade!", idade);
8         }
9
10        System.out.println("Idade ok! Pessoa tem " + idade + " anos.");
11    }
12
13    private static void testaIdadeEmpregado(int idade) {
14        try {
15            verificaIdadeEmpregado(idade);
16        } catch (MenorDeIdadeException e) {
17            System.out.println("\nExceção capturada: " + e.getMessage());
18            System.out.println("Idade da pessoa: " + e.getIdade() + " anos.\n");
19        }
20    }
21
22
23    public static void main(String[] args) {
24        testaIdadeEmpregado(42);
25        testaIdadeEmpregado(17);
26        testaIdadeEmpregado(18);
27    }
28
29 }
```

At the bottom, the Console window shows the output of the program:

```
<terminated> TestaExcecaoMenorDeldade [Java Application] C:\sts-4.14.1\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.2.v20220201-1208\jre\bin\javaw.exe (17 de ago. de 2022 13:0)
Idade ok! Pessoa tem 42 anos.

Exceção capturada: Empregado não pode ser menor de idade!
Idade da pessoa: 17 anos.

Idade ok! Pessoa tem 18 anos.
```

CRIANDO EXCEÇÕES PERSONALIZADAS

- Alguns autores defendem que exceções devem ser criadas apenas em situações excepcionais, para o caso de idades negativas por exemplo, e retornando verdadeiro ou falso no caso de ser maior ou menor de idade, respectivamente
- Renomeando a classe para `IdadeInvalidaException` deixando-a mais genérica

CRIANDO EXCEÇÕES PERSONALIZADAS

■ Melhorando a aplicação da exceção personalizada

```
1 package personalizada;
2
3 public class TestaExcecaoIdadeInvalida {
4
5     private static boolean verificaIdadeEmpregado(int idade) throws IdadeInvalidaException {
6         if (idade < 0) {
7             throw new IdadeInvalidaException("Erro de entrada! Idade não pode ser menor do que zero!", idade);
8         }
9
10        if (idade < 18) {
11            return false;
12        } else {
13            return true;
14        }
15    }
16
17    private static void testaIdadeEmpregado(int idade) {
18        try {
19            if (verificaIdadeEmpregado(idade)) {
20                System.out.println("Idade Ok! Pessoa tem " + idade + " anos.");
21            } else {
22                System.out.println("Idade inválida! Pessoa tem " + idade + " anos.");
23            }
24        } catch (IdadeInvalidaException e) {
25            System.out.println("\nExceção capturada: " + e.getMessage());
26            System.out.println("Valor de idade: " + e.getIdade() + "\n");
27        }
28    }
29
30    public static void main(String[] args) {
31        testaIdadeEmpregado(42);
32        testaIdadeEmpregado(17);
33        testaIdadeEmpregado(18);
34        testaIdadeEmpregado(-1);
35    }
36
37
38 }
```

```
<terminated> TestaExcecaoIdadeInvalida [Java Application] C:\sts-4.14.1\plugins\org.eclipse.justj.o
Idade Ok! Pessoa tem 42 anos.
Idade inválida! Pessoa tem 17 anos.
Idade Ok! Pessoa tem 18 anos.

Exceção capturada: Erro de entrada! Idade não pode ser menor do que zero!
Valor de idade: -1
```

BOAS PRÁTICAS

- Capture a exceção da forma mais específica/exata possível
 - Não capture a superclasse da exceção quando existir uma subclasse mais específica
 - Use mensagens claras nas próprias exceções
- Exceções não devem ser ‘silenciadas’ (blocos `catch` vazios), devem ser tratadas, propagadas ou então logadas
- Nunca lance nenhuma exceção com `throw` de dentro de um bloco `finally` pois dessa forma se perderá a exceção original
- Ao agrupar uma exceção em outra, não basta lançar a outra exceção, mantenha o *stacktrace*

BOAS PRÁTICAS

- Alguns defendem que em determinadas situações, como por exemplo no caso da divisão por zero, em vez de ser aplicado o tratamento de exceções, deve-se simplesmente fazer uma verificação antecipada do valor do denominador
- Alguns defendem que exceções devem ser utilizadas somente para cenários excepcionais, outros defendem que elas podem ser aplicadas mesmo em situações que não envolvem erros, sendo usadas para alterar o fluxo de execução do código
- A decisão de onde a exceção deve ser tratada deve ser sempre a que faça com que o código seja o mais reutilizável possível

BOAS PRÁTICAS

- Princípio “*throw early, catch later*”
 - Lance a exceção (usando `throw`) o quanto antes, no momento que o erro ocorrer, pois é quando se tem mais informações sobre o que a disparou e fica mais fácil descobrir qual foi a causa de origem
 - Capture a exceção (usando `catch`) o mais tarde possível, assim o programa pode ter toda a informação que precisa para tratá-la adequadamente
 - Em geral, exceções deveriam ser capturadas somente pelo código que está na melhor situação para fazer algo de útil com ela

BOAS PRÁTICAS

- Princípio “Tell, don't ask” (“afirme/mande, não pergunte”), artigo de [Martin Fowler](#)
 - Se precisar executar alguma ação, mandamos o objeto fazer; não perguntamos ao objeto se podemos fazer isso e então dependendo da resposta procedemos com a ação ou não
 - Exemplo de utilização de [enumerações](#)
 - O código ao lado está pedindo informações do objeto, perguntando sobre seu *status* e após isso toma uma ação caso contrário lança uma exceção

```
1 package telldontask_v1;
2
3 class PostJaPublicadoException extends RuntimeException {
4     public PostJaPublicadoException(String mensagem) {
5         super(mensagem);
6     }
7 }
8
9 enum PostStatus {
10     RASCUNHO, PUBLICADO
11 }
12
13 class Post {
14     public PostStatus status = PostStatus.RASCUNHO;
15 }
16
17 public class ExemploTellDontAskV1 {
18
19     public static void main(String[] args) {
20         Post postagem = new Post();
21
22         if (postagem.status == PostStatus.RASCUNHO) {
23             postagem.status = PostStatus.PUBLICADO;
24         } else {
25             throw new PostJaPublicadoException("Post já publicado!");
26         }
27     }
28 }
29 }
```

BOAS PRÁTICAS

■ Princípio “Tell, don't ask” (“afirme/mande, não pergunte”)

- Exemplo aplicando o padrão
- Objeto agora encapsula seu comportamento, e consegue decidir quando publicar ou não, lançando uma exceção caso necessário
- Código principal agora só executa a ação

```
1 package telldontask_v2;
2
3 class PostJaPublicadoException extends RuntimeException {
4     public PostJaPublicadoException(String mensagem) {
5         super(mensagem);
6     }
7 }
8
9 enum PostStatus {
10     RASCUNHO, PUBLICADO
11 }
12
13 class Post {
14     private PostStatus status = PostStatus.RASCUNHO;
15
16     public void publicar() {
17         if (this.status == PostStatus.RASCUNHO) {
18             this.status = PostStatus.PUBLICADO;
19         } else {
20             throw new PostJaPublicadoException("Post já publicado!");
21         }
22     }
23 }
24
25 public class ExemploTellDontAskV2 {
26
27     public static void main(String[] args) {
28         Post postagem = new Post();
29
30         // Aplicando o princípio "tell, don't ask"
31         postagem.publicar();
32     }
33
34 }
```

BOAS PRÁTICAS

■ Princípio “Fail fast” (“falhar rápido”), artigo de [Martin Fowler](#)

- Esse princípio procura analisar e tratar a condição de erro primeiro, com isso eliminando a necessidade de usar o `else`

```
1 package telldontask_v3;
2
3 class PostJaPublicadoException extends RuntimeException {
4     public PostJaPublicadoException(String mensagem) {
5         super(mensagem);
6     }
7 }
8
9 enum PostStatus {
10     RASCUNHO, PUBLICADO
11 }
12
13 class Post {
14     private PostStatus status = PostStatus.RASCUNHO;
15
16     public void publicar() {
17         // Aplicando o princípio "fail fast"
18         if (this.status != PostStatus.RASCUNHO) {
19             throw new PostJaPublicadoException("Post já publicado!");
20         }
21
22         this.status = PostStatus.PUBLICADO;
23     }
24 }
25
26 public class ExemploTellDontAskV3 {
27
28     public static void main(String[] args) {
29         Post postagem = new Post();
30
31         // Aplicando o princípio "tell, don't ask"
32         postagem.publicar();
33     }
34 }
35 }
```