

# Aula sobre Exceptions Java

---

## 1. Objetivo da aula (pra você se guiar)

---

Ao final da aula o aluno deve:

1. Saber **o que é uma Exception** e por que ela existe.

2. Entender a diferença entre:

- `Error`
- `Exception`
- `RuntimeException`
- Checked x Unchecked.

3. Saber usar:

- `try / catch`
- `finally`
- `throws` e `throw`

4. Entender o **stack trace** (mensagem de erro que aparece no console).

5. Criar uma **Exception personalizada** e usá-la em uma regra de negócio simples.

## 2. Parte teórica / conceitual

---

### 2.1. O que é uma Exception?

---

Exception é um “problema” que acontece em tempo de execução e interrompe o fluxo normal do programa.

Exemplos de situações que geram exception:

- Divisão por zero
- Acessar posição inválida de array
- Ler arquivo que não existe
- Conexão com banco caiu

- Conversão de texto para número inválida (ex: "abc" → `Integer.parseInt()`)

Na prática, quando isso acontece, o Java lança um objeto que representa esse erro (uma instância de `Throwable`).

## 2.2. Hierarquia básica

---

Mostra esse desenho na lousa ou slide:

```
java.lang.Object
  └─ java.lang.Throwable
    └─ java.lang.Error
    └─ java.lang.Exception
      └─ java.lang.RuntimeException
```

- **Throwable**

Tipo "raiz" de tudo que pode ser arremessado (`throw`).

- **Error**

- Problemas graves da JVM.
- Geralmente não tratamos: `OutOfMemoryError`, `StackOverflowError`, etc.
- São coisas que seu código normalmente **não resolve**.

- **Exception**

- Problemas que seu código **pode e deve tratar**.
- Divide em:
  - **Checked Exceptions** (verificadas)
  - **Unchecked Exceptions** (não verificadas)

## 2.3. Checked x Unchecked

---

### Checked Exceptions (verificadas)

- Herdam de `Exception` mas **não** de `RuntimeException`.
- O compilador **obriga** a tratar:
  - ou com `try/catch`

- ou declarando `throws` no método.
- Exemplo: `IOException`, `SQLException`.

## Unchecked Exceptions (Runtime)

---

- Herdam de `RuntimeException`.
- Não são obrigatorias de tratar.
- Normalmente representam **erros de programação**:
  - `NullPointerException`
  - `ArrayIndexOutOfBoundsException`
  - `NumberFormatException`
- **Você deve corrigir o código**, não só colocar um `try/catch` pra "mascarar".

## 2.4. Stack Trace

---

Mostra um stack trace real, por exemplo:

```
Exception in thread "main" java.lang.ArithmetricException: / by zero
at br.com.exemplo.App.main(App.java:10)
```

Explica:

- **Tipo da exception**: `java.lang.ArithmetricException`
- **Mensagem**: `/ by zero`
- **Caminho da origem**: `App.main(App.java:10)`
- Ele mostra a "pilha de chamadas" até chegar no ponto onde deu erro.

Sempre leia de cima para baixo. A linha mais importante é normalmente a primeira onde aparece **o seu código** (seu pacote/classe).

## 2.5. try / catch / finally

---

- `try`: bloco onde pode ocorrer a exception.
- `catch`: bloco que trata o erro.

- **finally**: bloco que sempre é executado (com ou sem erro), muito usado para:
  - fechar arquivos
  - fechar conexões
  - liberar recursos

## 2.6. throw x throws

---

- **throw** → usado **dentro** do método para **lançar** uma exception:

```
throw new IllegalArgumentException("Mensagem");
```

- **throws** → usado na **assinatura do método** para dizer que o método **pode lançar** uma exception:

```
public void lerArquivo() throws IOException {  
    // ...  
}
```

## 2.7. Vantagens e benefícios de usar Exception corretamente

---

- **Código mais robusto**: o sistema não “morre” com qualquer erro.
- **Mensagem clara para o usuário**: mensagem amigável ao invés de tela vermelha gigante.
- **Facilidade de debug**: stack trace bem tratado ajuda muito a encontrar o problema.
- **Separação de responsabilidades**:
  - lógica de negócio

- tratamento de erro
- **Melhor experiência para o usuário** → sistema estável, sem travar.

## 3. Montando o primeiro projeto (para iniciantes)

---

### 3.1. Estrutura do projeto

---

- Projeto simples: `exceptions-iniciante`
- Uma classe `App` com `public static void main(String[] args)`.
- Você pode ir adicionando exemplos dentro do `main` ou em métodos separados.

Exemplo de estrutura:

```
src
└ br.com.aula.exceptions
    └ App.java
```

### 3.2. Passo a passo (para explicar em aula)

---

1. Criar projeto Java (no Eclipse/IntelliJ ou até no VSCode).
2. Criar pacote: `br.com.aula.exceptions`.
3. Criar classe `App` com o método `main`.
4. Rodar um “Hello World” pra garantir que está tudo certo.
5. A partir daí, ir comentando/descomentando exemplos conforme explica.

## 4. Exemplos práticos (com explicação)

---

### Exemplo 1 – Exception sem tratamento (mostrar o problema)

---

**Código:**

```
package br.com.aula.exceptions;

public class App {

    public static void main(String[] args) {
        System.out.println("Início do programa");

        int a = 10;
        int b = 0;

        int resultado = a / b; // Aqui vai estourar a exception
        System.out.println("Resultado: " + resultado);

        System.out.println("Fim do programa");
    }
}
```

### O que acontece:

- Ao rodar, o programa lança **ArithmetricException: / by zero**.
- A linha `System.out.println("Fim do programa");` **não executa**.
- Mostra o stack trace no console.
- “Sem tratamento, o programa ‘morre’ na linha onde deu o erro.”
- “A exception interrompe o fluxo normal de execução.”

### Exemplo 2 – try / catch simples

---

#### Código:

```
package br.com.aula.exceptions;

public class App {

    public static void main(String[] args) {
        System.out.println("Início do programa");

        int a = 10;
        int b = 0;

        try {
            int resultado = a / b;
            System.out.println("Resultado: " + resultado);
        } catch (ArithmetricException e) {
            System.out.println("Erro ao dividir: " +
e.getMessage());
        }

        System.out.println("Fim do programa");
    }
}
```

### Explicação:

- O código dentro de `try` é “suspeito”.
- Se acontecer `ArithmetricException`, o fluxo pula para o `catch`.
- O programa não é encerrado bruscamente.
- `e.getMessage()` traz a mensagem da exception (`/ by zero`).

O objetivo não é esconder o erro, mas tratá-lo de forma controlada, continuar o programa ou pelo menos mostrar uma mensagem amigável.

### Exemplo 3 – Vários `catch` (tratando tipos diferentes)

---

Simula dois tipos de problema:

#### Código:

```

package br.com.aula.exceptions;

public class App {

    public static void main(String[] args) {
        String textoNumero = "abc"; // tenta trocar para "50" depois
        int divisor = 0;

        try {
            int numero = Integer.parseInt(textoNumero);
            int resultado = 100 / divisor;
            System.out.println("Resultado: " + resultado);
        } catch (NumberFormatException e) {
            System.out.println("Erro de conversão: valor não é um
número inteiro.");
        } catch (ArithmetricException e) {
            System.out.println("Erro de divisão: não é possível
dividir por zero.");
        } catch (Exception e) {
            System.out.println("Ocorreu um erro inesperado.");
        }

        System.out.println("Programa finalizado.");
    }
}

```

### Explicação:

- `Integer.parseInt("abc")` lança `NumberFormatException`.
  - Divisor `0` gera `ArithmetricException`.
  - A ordem dos `catch` importa:
    - Do mais específico para o mais genérico (`Exception` por último).
  - `catch (Exception e)` é um “guarda-chuva” geral.
- Sempre trate primeiro as exceções específicas que você sabe como lidar.  
Deixa o `Exception` genérico só como último recurso.

### Exemplo 4 – `finally` (recursos e limpeza)

Simular o uso de um recurso que precisa ser “limpo” no final:

### Código:

```

package br.com.aula.exceptions;

import java.util.Scanner;

public class App {

    public static void main(String[] args) {
        Scanner scanner = null;

        try {
            scanner = new Scanner(System.in);
            System.out.print("Digite um número inteiro: ");
            int numero = scanner.nextInt();
            System.out.println("Você digitou: " + numero);
        } catch (Exception e) {
            System.out.println("Erro ao ler o número: " +
e.getMessage());
        } finally {
            System.out.println("Fechando recursos...");
            if (scanner != null) {
                scanner.close();
            }
        }

        System.out.println("Programa encerrado.");
    }
}

```

### Explicação:

- **finally sempre** é executado:
  - com erro
  - sem erro
  - até se tiver **return** dentro do **try/catch**.
- Usado para **fechar recursos**:
  - arquivos
  - conexões

- scanners etc.

## Ponto para aula:

| Pense no `finally` como “independente do que acontecer, execute isso aqui antes de sair”.

## Exemplo 5 – `throws` e propagação de exception

### Código:

```
package br.com.aula.exceptions;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class App {

    public static void main(String[] args) {
        try {
            lerPrimeiraLinhaDoArquivo("arquivo_inexistente.txt");
        } catch (IOException e) {
            System.out.println("Erro ao ler arquivo: " +
e.getMessage());
        }

        System.out.println("Programa finalizado.");
    }

    public static void lerPrimeiraLinhaDoArquivo(String caminho)
throws IOException {
        BufferedReader br = new BufferedReader(new
FileReader(caminho));
        String linha = br.readLine();
        System.out.println("Primeira linha: " + linha);
        br.close();
    }
}
```

### Explicação:

- `FileReader` e `BufferedReader` podem lançar `IOException`.
- Em vez de tratar dentro de `lerPrimeiraLinhaDoArquivo`, declaramos:

```
java
```

```
public static void lerPrimeiraLinhaDoArquivo(...) throws  
IOException
```

- Quem chamar esse método é obrigado a tratar (ou também propagar).

**throws** não trata o erro, apenas avisa: “olha, eu posso lançar isso, trate você aí em cima”.

## **Exemplo 6 – Exception personalizada (regra de negócio)**

---

Cria uma regra simples de conta bancária:

### **6.1. Criar a exception customizada**

---

```
package br.com.aula.exceptions;  
  
public class SaldoInsuficienteException extends Exception {  
  
    public SaldoInsuficienteException(String mensagem) {  
        super(mensagem);  
    }  
}
```

## 6.2. Criar a classe Conta

---

```
package br.com.aula.exceptions;

public class Conta {

    private double saldo;

    public Conta(double saldoInicial) {
        this.saldo = saldoInicial;
    }

    public void sacar(double valor) throws
SaldoInsuficienteException {
        if (valor > saldo) {
            throw new SaldoInsuficienteException(
                "Saldo insuficiente. Saldo atual: " + saldo + ", "
valor solicitado: " + valor);
        }
        this.saldo -= valor;
    }

    public double getSaldo() {
        return saldo;
    }
}
```

### 6.3. Testar no `main`

```
package br.com.aula.exceptions;

public class App {

    public static void main(String[] args) {
        Conta conta = new Conta(100.0);

        try {
            System.out.println("Saldo inicial: " +
conta.getSaldo());
            conta.sacar(150.0);
            System.out.println("Saque realizado com sucesso.");
        } catch (SaldoInsuficienteException e) {
            System.out.println("Erro de negócio: " +
e.getMessage());
        }

        System.out.println("Saldo final: " + conta.getSaldo());
    }
}
```

#### Explicação:

- `SaldoInsuficienteException` representa um erro de **regra de negócio**, não um erro técnico.
- A conta só deixa sacar se tiver saldo suficiente.
- Se não tiver, lançamos a exception com uma mensagem clara.
- No `main`, tratamos de forma amigável.

Exceptions personalizadas deixam o código mais expressivo: ao invés de uma `Exception` genérica, você sabe exatamente o que aconteceu (saldo insuficiente, idade inválida, CPF duplicado, etc.).

1. **Para aplicações Desktop (Java puro)** – usando um *Exception Handler global* com `Thread.setDefaultUncaughtExceptionHandler`.
2. **Para aplicações Web (Spring Boot)** – usando `@ControllerAdvice` + `@ExceptionHandler`.

Vou te entregar código, explicação e como isso é usado em produção.

## 1. Exception Handling Centralizado — DESKTOP (Java Puro)

---

### Objetivo

---

Ter **um único ponto** responsável por capturar *qualquer exception não tratada* da aplicação.

Assim:

- Você captura erros que "escapam" do código.
- Não deixa o programa explodir sem controle.
- Pode mostrar uma mensagem amigável ao usuário.

- Pode gravar logs em arquivo, mandar alertas etc.

## 1.1. Criar Handler Global

```
package br.com.aula.exceptions.handler;

public class GlobalExceptionHandler implements
Thread.UncaughtExceptionHandler {

    @Override
    public void uncaughtException(Thread t, Throwable e) {
        System.err.println("Erro não tratado na thread: " +
t.getName());
        System.err.println("Tipo: " + e.getClass().getName());
        System.err.println("Mensagem: " + e.getMessage());

        // Aqui você pode:
        // - gravar log em arquivo
        // - mandar email
        // - mostrar caixa de diálogo (Swing/JavaFX)
        // - reiniciar parte do sistema

        exibirMensagemAmigavel(e);
    }

    private void exibirMensagemAmigavel(Throwable e) {
        System.out.println("\nAlgo inesperado aconteceu. Tente
novamente.");
    }
}
```

## 1.2. Registrar Handler no início da aplicação

```
package br.com.aula.exceptions;

import br.com.aula.exceptions.handler.GlobalExceptionHandler;

public class App {

    public static void main(String[] args) {

        // 👉 registra o handler global
        Thread.setDefaultUncaughtExceptionHandler(new
GlobalExceptionHandler());

        System.out.println("Início da aplicação Desktop");

        // Forçar um erro qualquer
        String texto = null;
        System.out.println(texto.length()); // NullPointerException
    }
}
```

### Como funciona?

Tudo que *não for capturado por try/catch* vai parar no `uncaughtException`.

Isso te dá:

- **Um único ponto** para controlar falhas inesperadas.
- Menos try/catch espalhado no código.

- Segurança contra "crashes silenciosos".

## 📌 1.3. Se for Swing, mostrar dialog

```
import javax.swing.JOptionPane;

private void exibirMensagemAmigavel(Throwable e) {
    JOptionPane.showMessageDialog(
        null,
        "Ocorreu um erro inesperado:\n" + e.getMessage(),
        "Erro",
        JOptionPane.ERROR_MESSAGE
    );
}
```

Perfeito para aplicações desktop corporativas.

## ✓ Resumo do Desktop

- Configura um único handler para capturar *todos* os erros não tratados.
- Mantém o sistema funcionando mesmo em falhas inesperadas.
- Permite mostrar erros amigáveis e registrar logs.

## 🔥 2. Exception Handling Centralizado — WEB (Spring Boot)

### 🎯 Objetivo

Ter *uma classe única* que intercepta **todas as exceptions** das controllers:

- Sem try/catch em todos os controllers
- Respostas JSON padronizadas
- Log centralizado
- Mensagens limpas e estruturadas

O padrão para isso é:

```
@ControllerAdvice + @ExceptionHandler
```

## 📌 2.1. Criar DTO de resposta de erro

```
package br.com.aula.exceptions.dto;

import java.time.LocalDateTime;

public class ErroDTO {

    private String mensagem;
    private String erro;
    private int status;
    private LocalDateTime timestamp;

    public ErroDTO(String mensagem, String erro, int status) {
        this.mensagem = mensagem;
        this.erro = erro;
        this.status = status;
        this.timestamp = LocalDateTime.now();
    }

    // getters
}
```

## 📌 2.2. Criar Exception Personalizada (Regra de Negócio)

```
package br.com.aula.exceptions;

public class SaldoInsuficienteException extends RuntimeException {
    public SaldoInsuficienteException(String mensagem) {
        super(mensagem);
    }
}
```

Note: agora ela é Runtime → mais prático em REST.

## 2.3. Criar Handler Global Web

```
java

package br.com.aula.exceptions.handler;

import br.com.aula.exceptions.SaldoInsuficienteException;
import br.com.aula.exceptions.dto.ErroDTO;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(SaldoInsuficienteException.class)
    public ResponseEntity<ErroDTO>
handleSaldo(SaldoInsuficienteException e) {

        ErroDTO erro = new ErroDTO(
            e.getMessage(),
            "Saldo insuficiente",
            HttpStatus.BAD_REQUEST.value()
        );

        return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body(erro);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErroDTO> handleGenerico(Exception e) {

        ErroDTO erro = new ErroDTO(
            "Erro interno no servidor.",
            e.getClass().getSimpleName(),
            HttpStatus.INTERNAL_SERVER_ERROR.value()
        );

        e.printStackTrace(); // log

        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(erro);
    }
}
```

## ✓ O que esse Handler faz?

- Trata **SaldoInsuficienteException** com status 400.
- Trata **qualquer outra exception** com 500.
- Retorna JSON padronizado:

```
{  
    "mensagem": "Saldo insuficiente",  
    "erro": "SaldoInsuficienteException",  
    "status": 400,  
    "timestamp": "2025-12-10T15:30:00"  
}
```

## 📌 2.4. Controller de exemplo

```
package br.com.aula.controller;  
  
import br.com.aula.exceptions.SaldoInsuficienteException;  
import org.springframework.web.bind.annotation.*;  
  
@RestController  
@RequestMapping("/conta")  
public class ContaController {  
  
    @GetMapping("/sacar/{valor}")  
    public String sacar(@PathVariable double valor) {  
  
        double saldo = 100;  
  
        if (valor > saldo) {  
            throw new SaldoInsuficienteException("Tentativa de saque acima do saldo.");  
        }  
  
        return "Saque realizado com sucesso!";  
    }  
}
```

## ✓ Resultado ao acessar /conta/sacar/200

```
{  
    "mensagem": "Tentativa de saque acima do saldo.",  
    "erro": "Saldo insuficiente",  
    "status": 400,  
    "timestamp": "2025-12-10T15:31:22"  
}
```

## 🚀 BENEFÍCIOS DO HANDLER CENTRALIZADO (WEB)

- Código mais limpo (sem try/catch espalhado).
- Mensagens padronizadas.
- Logging centralizado.
- Respostas REST amigáveis.
- Segurança (você controla o que é exposto ao cliente).

## 🌐 COMPARAÇÃO ENTRE DESKTOP E WEB

Item	Desktop (Java Puro)	Web (Spring Boot)
Ponto central	<code>Thread.UncaughtExceptionHandler@ControllerAdvice</code>	
Tratamento	Mensagens, logs locais, diálogos	JSON padronizado, HTTP Status
Abordagem	Para erros não tratados	Para erros que chegam ao controller
Benefício	Evita crash silencioso	APIs mais profissionais e seguras