

Universidade Federal da Paraíba

Centro de Informática

---

Departamento de Informática

# Linguagem de Programação I

## Exceções

▶ Tiago Maritan

▶ [tiago@ci.ufpb.br](mailto:tiago@ci.ufpb.br)

---



# Motivação

---

- ▶ Independentemente de quão bons programadores somos, **não podemos controlar tudo!**
- ▶ **Problemas acontecem!!!**
  - ▶ O arquivo não está no local!!!
  - ▶ O servidor está travado!!!
- ▶ Essas **situações excepcionais** fogem do controle do programador... mas podem ser **contornadas** em Java!
- ▶ Aprenderemos como fazer isso na aula de hoje!

# Três tipos de erros de tempo de execução

---

## 1. Erros de lógica de programação

- ▶ Ex: limite do array ultrapassado, divisão por zero, etc.
- ▶ Devem ser corrigidos pelo programador

## 2. Erros graves onde não adianta tentar recuperação

- ▶ Ex: falta de memória, erro interno da JVM
- ▶ Fogem do controle do programador e não podem ser contornados

## 3. Erros devido a condições do ambiente de execução

- ▶ Ex: arquivo não encontrado, rede fora do ar, etc.
- ▶ Fogem do controle do programador **mas podem ser contornados em tempo de execução**

# Três tipos de erros de tempo de execução

---

## 1. Erros de lógica de programação

- ▶ Ex: limites do vetor ultrapassados, divisão por zero, etc.
- ▶ Devem ser corrigidos pelo programador

## 2. Erros graves onde não adianta tentar recuperação

- ▶ Ex: falta de memória, erro interno da JVM
- ▶ Fogem do controle do programador e não podem ser contornados

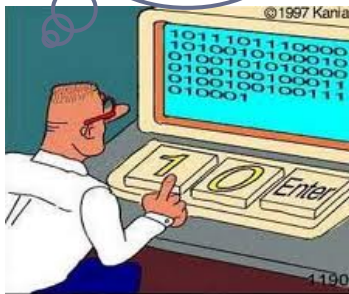
## 3. Erros devido a condições do ambiente de execução

- ▶ Ex: arquivo não encontrado, rede fora do ar, etc.
- ▶ Fogem do controle do programador **mas podem ser contornados em tempo de execução**

# Exceções

- ▶ Em Java, os **métodos podem indicar** que “situações excepcionais” (**Exceções**) podem ocorrer com ele;
- ▶ Se soubermos disso, podemos nos preparar para essas situações quando formos chamar esses métodos.

Preciso chamar o método  
mI() da classe CI;  
Será que ele pode travar?



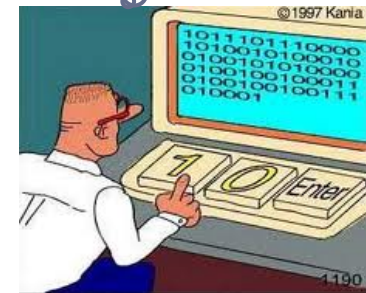
LOS VERDADEROS PROGRAMADORES  
PROGRAMAN EN BINARIO

Meu método mI()  
será interrompido se o  
servidor travar!



**Classe CI**

Ok!. Agora posso tomar  
minhas precauções!!!



LOS VERDADEROS PROGRAMADORES  
PROGRAMAN EN BINARIO

# Exceções

- ▶ Métodos que podem lançar uma exceção declaram isso em uma cláusula **throws**!



## Method Detail

### read

```
public int read()  
    throws IOException
```

Indica que o método `read()`  
pode lançar uma exceção `IOException`

Reads a byte of data from this input stream. This method blocks if no input is yet available.

#### Specified by:

read in class InputStream

#### Returns:

the next byte of data, or -1 if the end of the file is reached.

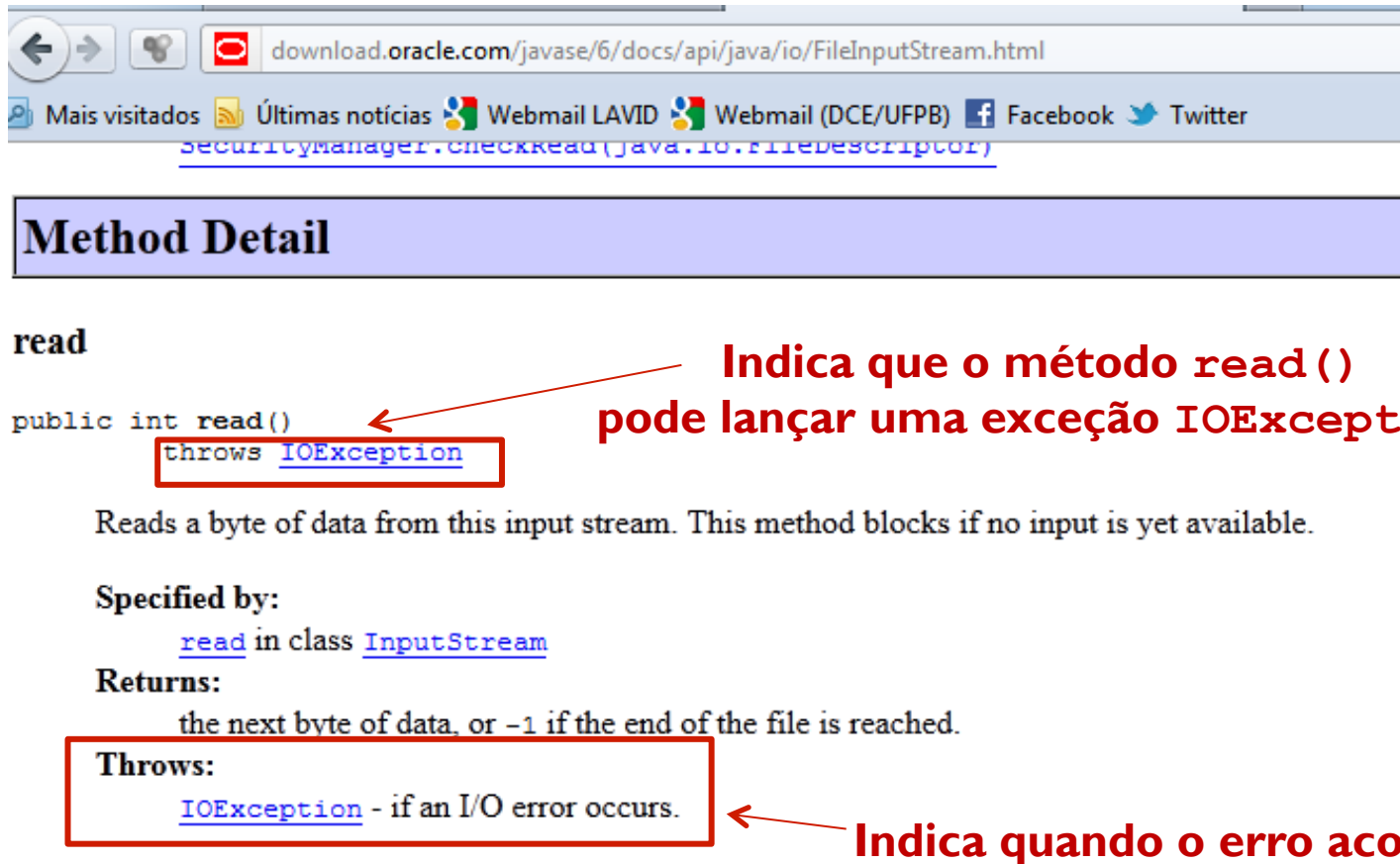
#### Throws:

IOException - if an I/O error occurs.

Indica quando o erro acontece.

# Exceções

- ▶ Métodos que podem lançar uma exceção declaram isso em uma cláusula **throws**!



The screenshot shows the Oracle Java API documentation for the `read()` method of the `FileInputStream` class. The browser address bar shows the URL `download.oracle.com/javase/6/docs/api/java/io/FileInputStream.html`. The page title is "Method Detail". The method signature is `public int read() throws IOException`. The `throws IOException` part is highlighted with a red box. A red arrow points from the text "Indica que o método read () pode lançar uma exceção IOException" to the `throws IOException` part. Below the signature, the description reads: "Reads a byte of data from this input stream. This method blocks if no input is yet available." The "Specified by:" section points to the `read` method in the `InputStream` class. The "Returns:" section states: "the next byte of data, or -1 if the end of the file is reached." The "Throws:" section is highlighted with a red box and contains the text: "`IOException` - if an I/O error occurs." A red arrow points from the text "Indica quando o erro acontece." to the "Throws:" section.

**Method Detail**

**read**

```
public int read()
    throws IOException
```

Indica que o método `read ()` pode lançar uma exceção `IOException`

Reads a byte of data from this input stream. This method blocks if no input is yet available.

Specified by:  
read in class InputStream

Returns:  
the next byte of data, or -1 if the end of the file is reached.

**Throws:**  
IOException - if an I/O error occurs.

Indica quando o erro acontece.

# Exceções

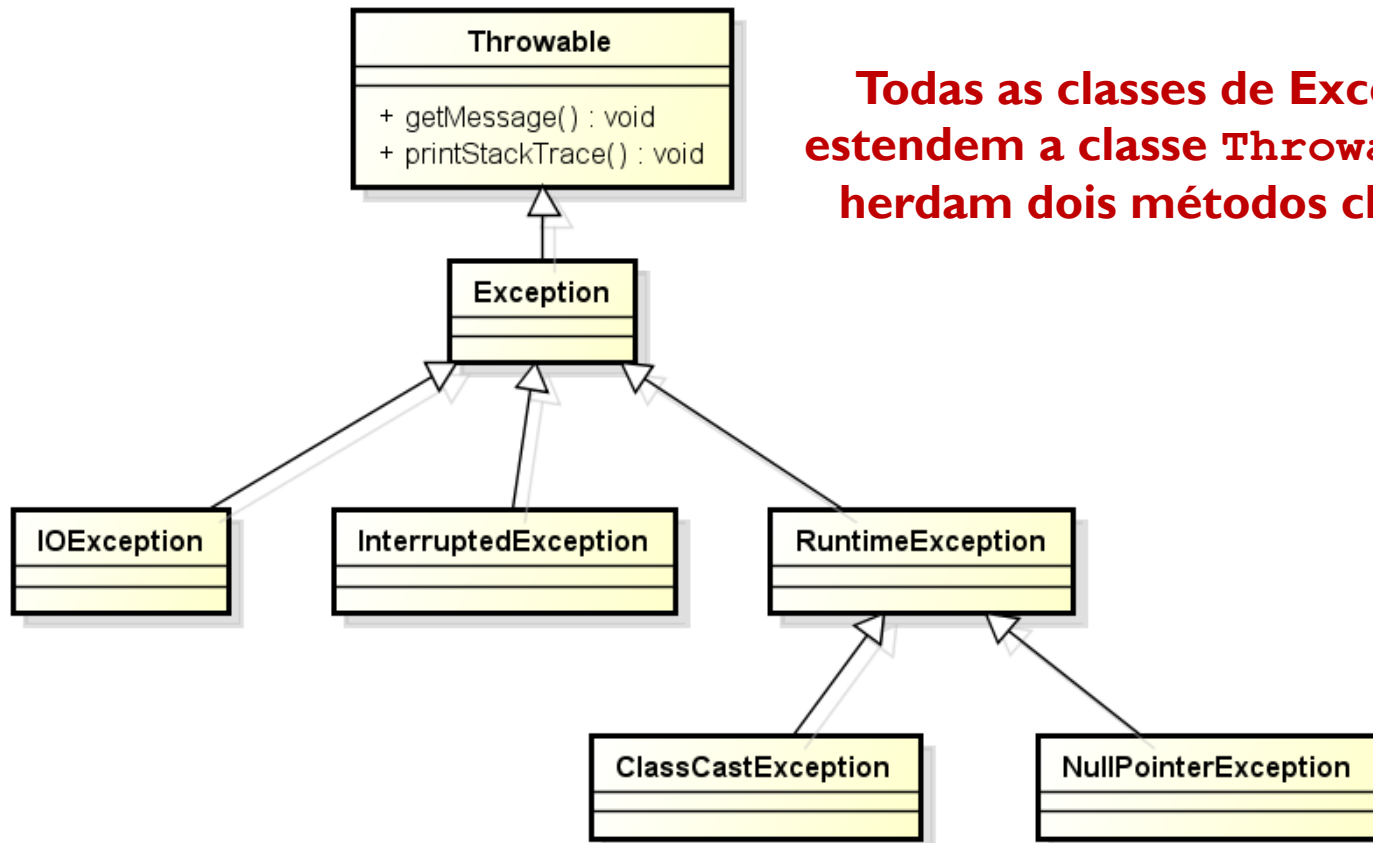
---

- ▶ Exceções são o mecanismo usado por Java para tratar erros
- ▶ O termo vem de “**Exceptional Events**”
- ▶ Quando um erro ocorre dentro de um método, o método **cria um objeto (exception)** e entrega para alguém tratar...
- ▶ **Exceção:** é um **objeto do tipo Exception**.
  - ▶ Todos os tipos de exceções são subclasses de Exception
  - ▶ Java já provê um grande conjunto de exceções.
  - ▶ **FileNotFoundException** - arquivo não encontrado;
  - ▶ **IOException** – algum erro de entrada e saída



# Exceções

## ► Parte da hierarquia de classes de **Exception**



**Todas as classes de Exceção estendem a classe Throwable e herdam dois métodos chaves**

# Criando suas classes de Exceção

---

- ▶ É só herdar de Exception:

```
public class MinhaException extends Exception{  
    // código da minha classe de Exceção  
}
```

# Lançando Exceções

---

- ▶ Objetos Exception precisam ser criados com **new** e depois lançados com a cláusula **throw**

```
MinhaException e = new  
    MinhaException("Erro!");  
  
throw e; // exceção foi lançada!
```

- ▶ A variável de referência é desnecessária
  - ▶ A sintaxe a baixo é mais usual:

```
throw new MinhaException ("Erro!");
```

# Declarando Exceções

---

- ▶ Métodos (ou construtores) que **possam lançar exceção**, devem informar isso na sua declaração
- ▶ Usando a cláusula **throws**

```
public void metodoMau() throws MinhaException{  
  
    if (condicaoInesperada == true){  
        // lançando MinhaException  
        throw new MinhaException ();  
    }  
}
```



# O que fazer quando uma exceção é lançada?

---

- ▶ Quando chamamos um **método que pode lançar exceções...** precisamos estar preparados para lidar com essas exceções.
- ▶ Isto é, precisamos indicar ao compilador que faremos algo;
- ▶ Isso pode ser feito de duas formas
  1. **Manipulando (ou capturando) as Exceções**
  2. **Desviando (ou declarando) as Exceções**
- ▶ Caso contrário... **erro de compilação**;

# Manipular ou capturar uma Exceção

---

- ▶ Para manipular (capturar) exceções usamos um bloco `try/catch`
  - ▶ Indica que a exceção será manipulada, caso ela ocorra

- ▶ Sintaxe:

```
try{  
    // chamada para um metodo que pode  
    // lançar uma exceção  
}  
catch (<TipoException1> ex1) {  
    // tenta resolver o problema1  
}  
catch (<TipoException2> ex2) {  
    // tenta resolver o problema2  
}
```

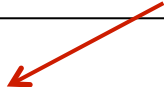
# Manipular ou capturar uma Exceção

- ▶ Para manipular exceções usamos um bloco `try/catch`
  - ▶ Indica que a exceção será manipulada, caso ela ocorra

- ▶ Sintaxe:

```
try{  
    // chamada para um método que pode  
    // lançar uma exceção  
}  
catch (<TipoException1> ex1) {  
    // tenta resolver o problema1  
}  
catch (<TipoException2> ex2) {  
    // tenta resolver o problema2  
}
```

**Método perigoso é  
invocado no bloco try**



**Blocos catch indicam o que será feito  
se cada situação excepcional ocorrer**



# Blocos try/catch

---

## ► Exemplo:

```
public class Teste{
    C1 obj1 = new C1();

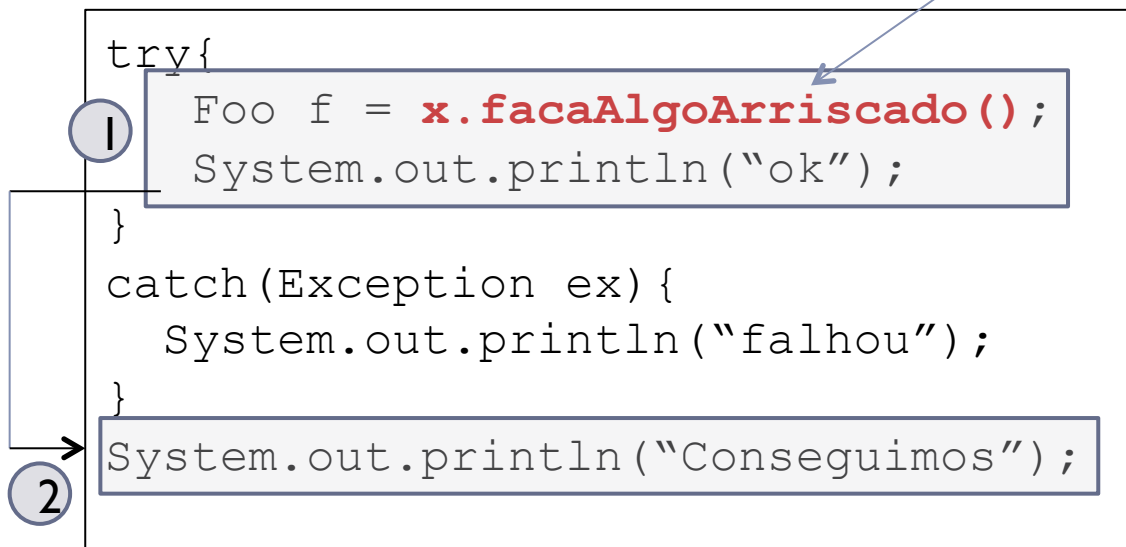
    public void iniciar(){
        try{
            obj1.m1(); // m1 é perigoso -lanca uma exceção
                      // quando o servidor trava
            System.out.println("Continua se não ocorreu");
        }
        catch(ServerNotActiveException ex){
            // Trata a exceção
            System.out.println("Servidor fora do ar");
        }
    }
}
```



# Controle de fluxo em bloco try/catch

- ▶ Quando um método perigoso é invocado, ele pode ser:
  - ▶ **Bem sucedido:** exceção não é lançada (i.e., não ocorre);
  - ▶ **Mal sucedido:** exceção ocorre e é lançada
- ▶ Se o método perigoso for bem sucedido

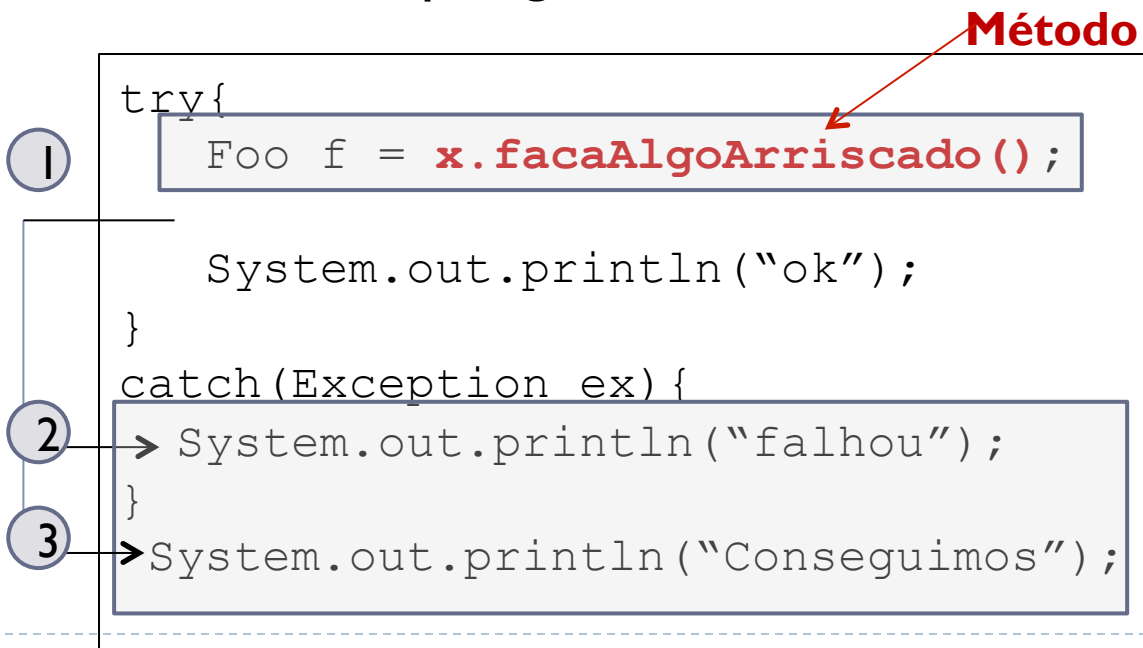
**Método perigoso**



**Bloco try é executado e, em seguida, o código abaixo do catch**

# Controle de fluxo em bloco try/catch

- ▶ Quando um método perigoso é invocado, ele pode ser:
  - ▶ **Bem sucedido:** exceção não é lançada (i.e., não ocorre);
  - ▶ **Mal sucedido:** exceção ocorre e é lançada
- ▶ Se o método perigoso for mal sucedido:



**Método perigoso**

**Método lança a exceção  
e o resto do bloco try  
não é executado**

**Bloco catch é executado  
e o método continua  
a execução daí em diante**

# Bloco finally

---

- ▶ Bloco finally é usado junto (com try/catch) para inserir código que deve ser executado independentemente de uma exceção
- ▶ Ex: Se você quer cozinhar algo, iniciará acendendo o forno.
  - ▶ Se a tentativa de cozinhar falhar, **teremos que desligar o forno;**
  - ▶ Se a tentativa de cozinhar der certo, **teremos que desligar o forno;**
  - ▶ ***Teremos que desligar o forno de qualquer maneira;***

```
try{
    ligarForno(); x.cozinhar();
}
catch(CozinhandoException ex){
    ex.printStackTrace();
}
finally{
    desligarForno();
}
```

# Bloco finally

---

- ▶ Sem o finally teríamos que inserir o código no try e no catch

## Com o finally

```
try{
    ligarForno(); x.cozinhar();
}
catch (CozinhandoException ex) {
    ex.printStackTrace();
}
finally{
    desligarForno();
}
```

## Sem o finally

```
try{
    ligarForno();
    x.cozinhar();
    desligarForno();
}
catch (CozinhandoException ex) {
    ex.printStackTrace();
    desligarForno();
}
```

# Bloco finally

---

- ▶ Se o bloco try falhar (ocorrer uma exceção):
  - ▶ Controle passará para o catch e depois para o finally;
- ▶ Se o bloco try for bem sucedido (sem ocorrer exceção):
  - ▶ Controle passará do bloco try direto para o bloco finally;
- ▶ Mesmo se o bloco try ou catch tiver uma instrução de retorno... **finally será executado antes de retornar.**

# Desviando (ou Declarando) Exceções

- ▶ Se não quisermos manipular (capturar) uma exceção usando um bloco try/catch... podemos **desviar** dela... **declarando-a**.
- ▶ **Desviar da exceção** significa deixar que o método que chamou você capture a exceção.
  - ▶ *“Repassa a exceção para outro método”*
- ▶ Isso é feito declarando a exceção com **throws**

```
public void foo() throws BadException{  
    Foo f = x.facaAlgoArriscado();  
    System.out.println("Conseguimos");  
}
```

foo() em vez de  
capturar (try/catch),  
exceção, desvia..

Pode lançar exceção

# Desviando (ou declarando) Exceções

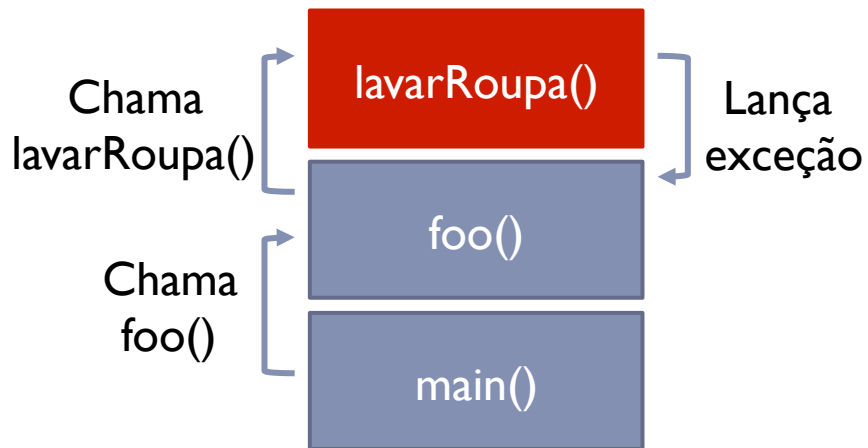
---

- ▶ Desviar só retarda o inevitável
  - ▶ Cedo ou tarde alguém terá que capturar a exceção... lidar com ela
- ▶ Mas se todos (até o main()) se desviarem dela?
  - ▶ Nesse caso, o programa será encerrado abruptamente

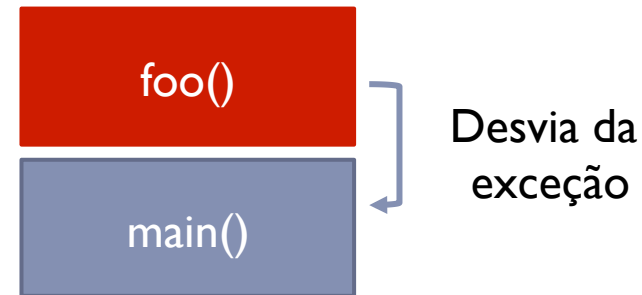
```
public class Lavadora{  
    //desvia-se da excecao  
    public void foo() throws RoupaException{  
        Lavanderia l = new Lavanderia();  
        l.lavarRoupa(); // lança a exceção RoupaException  
    }  
    //desvia-se da excecao novamente  
    public static void main(String args[]) throws RoupaException{  
        Lavadora a = new Lavadora();  
        a.foo();  
    }  
}
```

# Desviando (ou declarando) Exceções

## 1. lavarRoupa() lança uma RoupaException



## 2. foo() desvia da exceção



## 3. main () desvia da exceção



## 4. JVM é encerrada



# RuntimeExceptions não precisam ser capturadas ou desviadas?

---

- ▶ RuntimeExceptions não precisam ser capturadas ou desviadas
  - ▶ Podem ser lançadas a qualquer momento;
  - ▶ Em geral, estão associadas a **erros de lógica do programa**;
  - ▶ **Não associadas a falhas impossíveis de prever ou detectar**;
- ▶ Ex: **ArrayIndexOutOfBoundsException**: acesso a posição inválida de um array;

```
int a[] = new int[10];  
a[10] = 12; // acesso a posição inválida
```

- ▶ Ex: **DividedByZeroException**: dividindo um número por zero

```
int x = 0;  
int z = 10/x; // divisão por 0
```

# Exercício

---

- ▶ (a) Crie uma interface `IConta` com os métodos “`void sacar(double valor)`” e “`void depositar(double valor)`”.
- ▶ (b) Crie uma classe `Conta` que implementa `IConta` e que contenha os atributos `nomeCliente`, do tipo `String`, `salarioMensal`, `numeroConta`, `saldo` e `limite`, do tipo `double`, e os métodos para obter e alterar esses atributos (métodos `get` e `set`). Além disso, essa classe possui as seguintes características:
  - ▶ Os valores dos atributos `nomeCliente`, `salarioMensal`, `numeroConta` e `saldo` são configurados no construtor da classe.
  - ▶ O método `sacar` deve lançar uma exceção `SaldoNaoDisponivelException`, quando o valor a ser sacado é maior que o saldo disponível.
  - ▶ O método “`void definirLimite()`”, define o valor do atributo `limite` como 2 vezes o valor de `salarioMensal`.
- ▶ (c) Crie uma classe `ContaEspecial` que herda da classe `Conta` e sobrescreve o método `definirLimite()` como 3 vezes o valor de `salarioMensal`.

# Exercício

---

- ▶ (d) Escreva os códigos da classe de exceção SaldoNaoDisponivelException.
- ▶ (e) Crie um programa (classe) principal MinhaConta. O programa deve criar um objeto da classe Conta, chamar o método sacar (da classe Conta) e capturar a exceção SaldoNaoDisponivelException. Ao capturar a exceção, o programa deve imprimir uma mensagem informando o problema.

Universidade Federal da Paraíba

Centro de Informática

---

Departamento de Informática

# Linguagem de Programação I

## Exceções

▶ Tiago Maritan

▶ [tiago@ci.ufpb.br](mailto:tiago@ci.ufpb.br)

