

INE5404

Tratamento de Exceções

prof. Jônata Tyska
prof. Mateus Grellert



UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Tipos de Erros em Desenvolvimento de SW

- Erros de compilação
- Erros de implementação
- Erros de execução por falta de recursos
- Erros de execução por condições excepcionais

Tipos de Erros em Desenvolvimento de SW

- Erros de compilação
- Erros de **implementação**
- Erros de execução por **falta de recursos**
- Erros de execução por **condições excepcionais**

**Todos acontecem em
tempo de execução!**

Exceções

Exceções são condições anômalas ou excepcionais que ocorrem em tempo de execução e que exigem algum tipo de tratamento especial

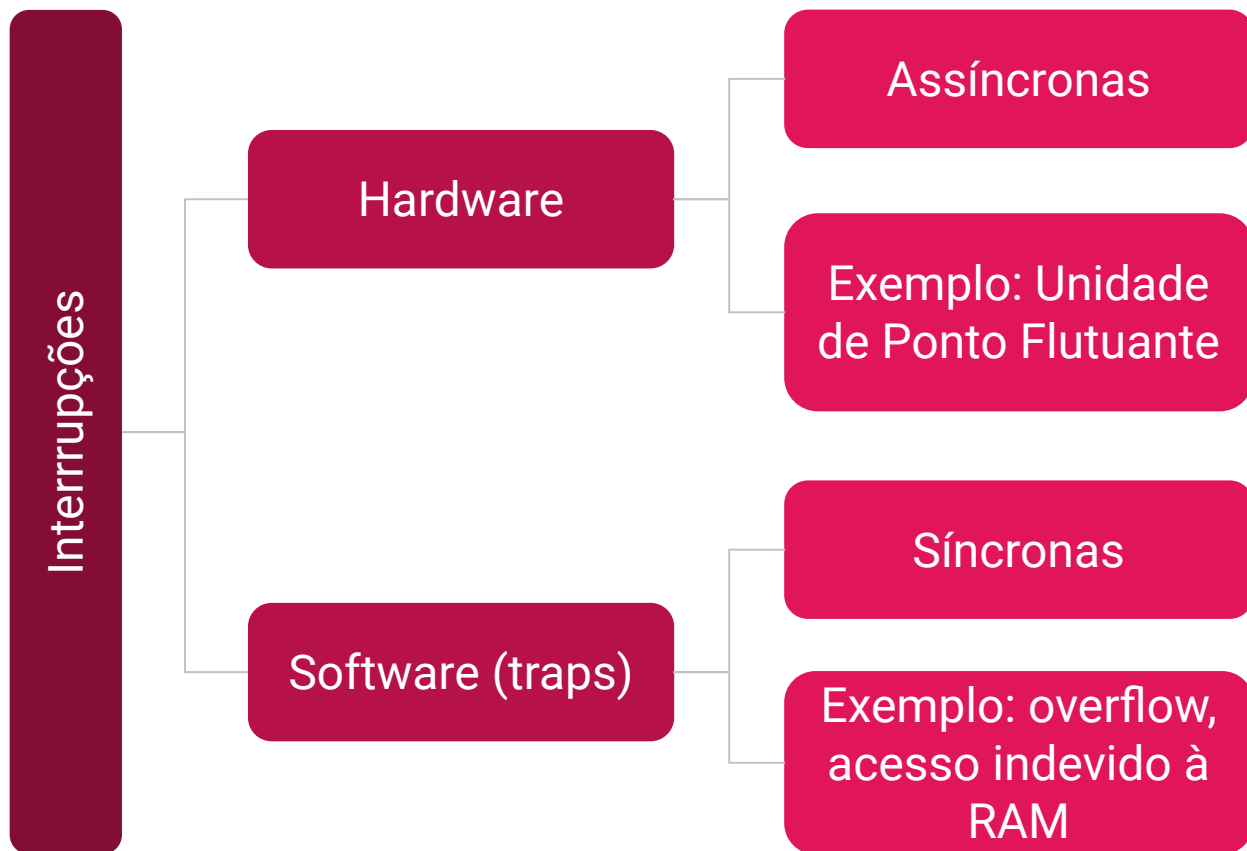
Exemplos:

- divisão por zero
- acesso a um índice não alocado de um array
- requisição HTTP sem resposta

Exceções x Erros

- Conceitos similares relacionados a um problema na execução de programas
- Erros normalmente denotam situações em que é **impossível recuperar** a execução (e.g.: OutOfMemory)*
- Exceções denotam **problemas recuperáveis/tratáveis** (e.g.: recurso não encontrado)

*Alguns autores usam o termo erro fatal para esse tipo



Exceções são
um tipo de
**interrupção
de software**

Situações de Exceção

```
public class ExceptionDemo {  
  
    public static void main (String[] args) {  
        System.out.println(divideArray(args));  
    }  
  
    private static int divideArray(String[] array) {  
        String s1 = array[0];  
        String s2 = array[1];  
        return divideStrings(s1, s2);  
    }  
  
    private static int divideStrings(String s1, String s2) {  
        int i1 = Integer.parseInt(s1);  
        int i2 = Integer.parseInt(s2);  
        return divideInts(i1, i2);  
    }  
  
    private static int divideInts(int i1, int i2) {  
        return i1 / i2;  
    }  
}
```

**Algun
Problema?**

**Compilou aqui,
professor! Deve
estar tudo certo**



Mas... o quê?

```
grellert@grellert-notebook:~/INE5404/app_exception$ python app.py 1 a
```

```
Traceback (most recent call last):
```

```
File "app.py", line 4, in <module>
```

```
    demo = ExceptionDemoSafe(argv[1:])
```

```
File "/Users/grellert/INE5404/app_exception/ExceptionDemoSafe.py", line 4, in __init__
```

```
    self.divideSafely(args)
```

```
File "/Users/grellert/INE5404/app_exception/ExceptionDemoSafe.py", line 8, in divideSafely
```

```
    print(self.divideArray(array));
```

```
File "/Users/grellert/INE5404/app_exception/ExceptionDemoSafe.py", line 19, in divideArray
```

```
    return self.divideStrings(s1, s2);
```

```
File "/Users/grellert/INE5404/app_exception/ExceptionDemoSafe.py", line 24, in divideStrings
```

```
    i2 = int(s2)
```

```
ValueError: invalid literal for int() with base 10: 'a'
```

Pensando bem,
nosso código
tem algumas
vulnerabilidades

Situações de Exceção

```
public class ExceptionDemo {  
  
    public static void main (String[] args) {  
        System.out.println(divideArray(args));  
    }  
  
    private static int divideArray(String[] array) {  
        String s1 = array[0];  
        String s2 = array[1];  
        return divideStrings(s1, s2);  
    }  
  
    private static int divideStrings(String s1, String s2) {  
        int i1 = Integer.parseInt(s1);  
        int i2 = Integer.parseInt(s2);  
        return divideInts(i1, i2);  
    }  
  
    private static int divideInts(int i1, int i2) {  
        return i1 / i2;  
    }  
}
```

Pensando bem,
nosso código
tem algumas
vulnerabilidades

Situações de Exceção

```
public class ExceptionDemo {  
  
    public static void main (String[] args) {  
        System.out.println(divideArray(args));  
    }  
  
    private static int divideArray(String[] array) {  
        { String s1 = array[0];  
          String s2 = array[1]; } ❶ Acesso indevido  
        return divideStrings(s1, s2);  
    }  
  
    private static int divideStrings(String s1, String s2) {  
        int i1 = Integer.parseInt(s1);  
        int i2 = Integer.parseInt(s2);  
        return divideInts(i1, i2);  
    }  
  
    private static int divideInts(int i1, int i2) {  
        return i1 / i2;  
    }  
}
```

Pensando bem,
nosso código
tem algumas
vulnerabilidades

Situações de Exceção

```
public class ExceptionDemo {
```

```
    public static void main (String[] args) {  
        System.out.println(divideArray(args));  
    }
```

```
    private static int divideArray(String[] array) {  
        { String s1 = array[0];  
          String s2 = array[1]; } ① Acesso indevido  
        return divideStrings(s1, s2);  
    }
```

```
    private static int divideStrings(String s1, String s2) {  
        { int i1 = Integer.parseInt(s1);  
          int i2 = Integer.parseInt(s2); } ② Typecast  
        return divideInts(i1, i2);  
    }
```

```
    private static int divideInts(int i1, int i2) {  
        return i1 / i2;  
    }
```

```
}
```

Pensando bem,
nosso código
tem algumas
vulnerabilidades

Situações de Exceção

```
public class ExceptionDemo {
```

```
    public static void main (String[] args) {  
        System.out.println(divideArray(args));  
    }
```

```
    private static int divideArray(String[] array) {  
        { String s1 = array[0];  
          String s2 = array[1]; } ① Acesso indevido  
        return divideStrings(s1, s2);  
    }
```

```
    private static int divideStrings(String s1, String s2) {  
        { int i1 = Integer.parseInt(s1);  
          int i2 = Integer.parseInt(s2); } ② Typecast  
        return divideInts(i1, i2);        inválido  
    }
```

```
    private static int divideInts(int i1, int i2) {  
        { return i1 / i2; } ③ Divisão por zero  
    }  
}
```

Exceções não tratadas (Uncaught Exceptions)

- Exceções não tratadas na implementação são normalmente tratadas por uma **rotina padrão** definida pela linguagem
- Normalmente, as rotinas padrão **terminam a execução do programa**

A Pilha de Chamadas (Call Stack)

Exceções normalmente geram um código de erro que imprime a **pilha de chamadas (ou call stack)**

Muitos se assustam com a call stack no início, mas na verdade ela é uma **forte aliada** na depuração de programas

stderr

```
File "app.py", line 4, in  
__init__  
    foo(args)  
File "app.py", line 8, in foo  
    bar(args)  
File "app.py", line 19, in bar  
    baz(s1,s2)  
File "app.py", line 24, in baz  
    i2 = int(s2)
```

A Pilha de Chamadas (Call Stack)

stderr

```
File "app.py", line 4, in __init__  
    foo(args)  
File "app.py", line 8, in foo  
    bar(args)  
File "app.py", line 19, in bar  
    baz(s1,s2)  
File "app.py", line 24, in baz  
    i2 = int(s2)
```



declaração que causou o erro

A Pilha de Chamadas (Call Stack)

stderr

```
File "app.py", line 4, in __init__  
    foo(args)  
File "app.py", line 8, in foo  
    bar(args)  
File "app.py", line 19, in bar  
    baz(s1,s2)  
File "app.py", line 24, in baz  
    i2 = int(s2)
```

Nome do fonte

Linha que causou o erro

A Pilha de Chamadas (Call Stack)

stderr

```
File "app.py", line 4, in __init__  
    foo(args)  
File "app.py", line 8, in foo  
    bar(args)  
File "app.py", line 19, in bar  
    baz(s1,s2)  
File "app.py", line 24, in baz  
    i2 = int(s2)
```

Encadeamento de chamadas (permite rastrear a execução desde o início)

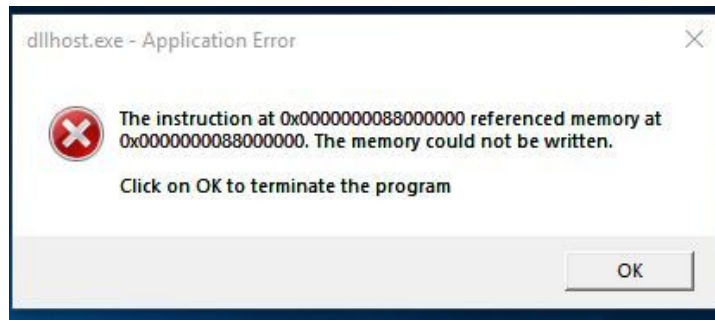
**Até a
próxima!**

Programação Defensiva



Programação Defensiva

- Exceções não tratadas geram **falhas de segurança**
- Também prejudicam significativamente a **experiência do usuário**
- Para resolver esses problemas, podemos trabalhar com o conceito de **programação defensiva**
 - ◆ Testes constantes
 - ◆ **Tratamento de exceções**



Tratando Exceções em SW

- A maior parte das linguagens modernas de programação provêm mecanismos de tratamento de exceção
 - ◆ Estruturas de Dados (classes)
 - ◆ Comandos especiais (try...catch e throw)

Voltando ao nosso exemplo

```
class ExceptionDemo:
    def __init__(self, args):
        self.divide(args)

    def divide(self, array):
        print(self.divideArray(array))

    def divideArray(self, array):
        s1 = array[0]
        s2 = array[1]
        return self.divideStrings(s1, s2);

    def divideStrings(self, s1, s2):
        i1 = int(s1)
        i2 = int(s2)
        return self.divideInts(i1, i2);

    def divideInts(self, i1, i2):
        return i1 / i2
```

Python - Comandos Try... Except... Finally

A [spec de Python](#) define algumas construções para tratarmos exceções (muito similar em Java, C++, JS):

- [try](#): define trechos de código de podem causar situações de exceção
- [except](#): define o bloco onde onde um ou mais handlers (tratadores) serão implementados para recuperação da exceção

Try...except - Tentativa 1

```
def divideStrings(self, s1, s2):  
    try:  
        i1 = int(s1)  
        i2 = int(s2)  
    except:  
        print("Oops! Use valores inteiros", file=sys.stderr)  
        exit(1)  
  
    return self.divideInts(i1, i2);
```

Try...except - Tentativa 1

```
def divideStrings(self, s1, s2):  
    try:  
        i1 = int(s1)  
        i2 = int(s2)  
    except:  
        print("Oops! Use valores inteiros", file=sys.stderr)  
        exit(1)  
  
    return self.divideInts(i1, i2);
```

Como ter certeza
de que foi esse o
problema?

Python - Comandos Try... Except... Finally

- A boa prática requer que o erro seja **capturado**
- O comando **except** pode ser complementado para isso:

`except Exception as e:`



Tipo de exceção que queremos tratar neste bloco except. A **classe Exception** é a mais genérica e vai pegar todas.

Try... except - Tentativa 2

```
def divideStrings(self, s1, s2):  
    try:  
        i1 = int(s1)  
        i2 = int(s2)  
    except Exception as e:  
        print('Erro:', e)  
        print("Oops! Use valores inteiros", file=sys.stderr)  
        exit(1)  
  
    return self.divideInts(i1, i2);
```

Python - Comandos Try... Except... Finally

- Além dos comandos **try** e **except**, existe o comando **finally**.
- Ele define um bloco de código que é **sempre executado** (com ou sem exceção)
- Normalmente utilizado para código de limpeza (fechamento de arquivos, desalocar memória, etc)

Exemplo prático do comando finally

```
def processData(self, filePath):  
    fileIn = open(filePath)  
    parsed = None  
    try:  
        | parsed = self.parseData(fileIn)  
    except Exception as e:  
        | print("Erro ao tentar parsear dados")  
    finally:  
        | fileIn.close()  
  
    return parsed
```

Ativando uma exceção

- O próprio programador pode definir quando um código ativa uma exceção com o comando **raise**

```
try:  
    addr = fetchAddress(mem, data)  
    if addr == -1:  
        raise KeyError("Dado não encontrado")  
    ...
```

A Classe Exception

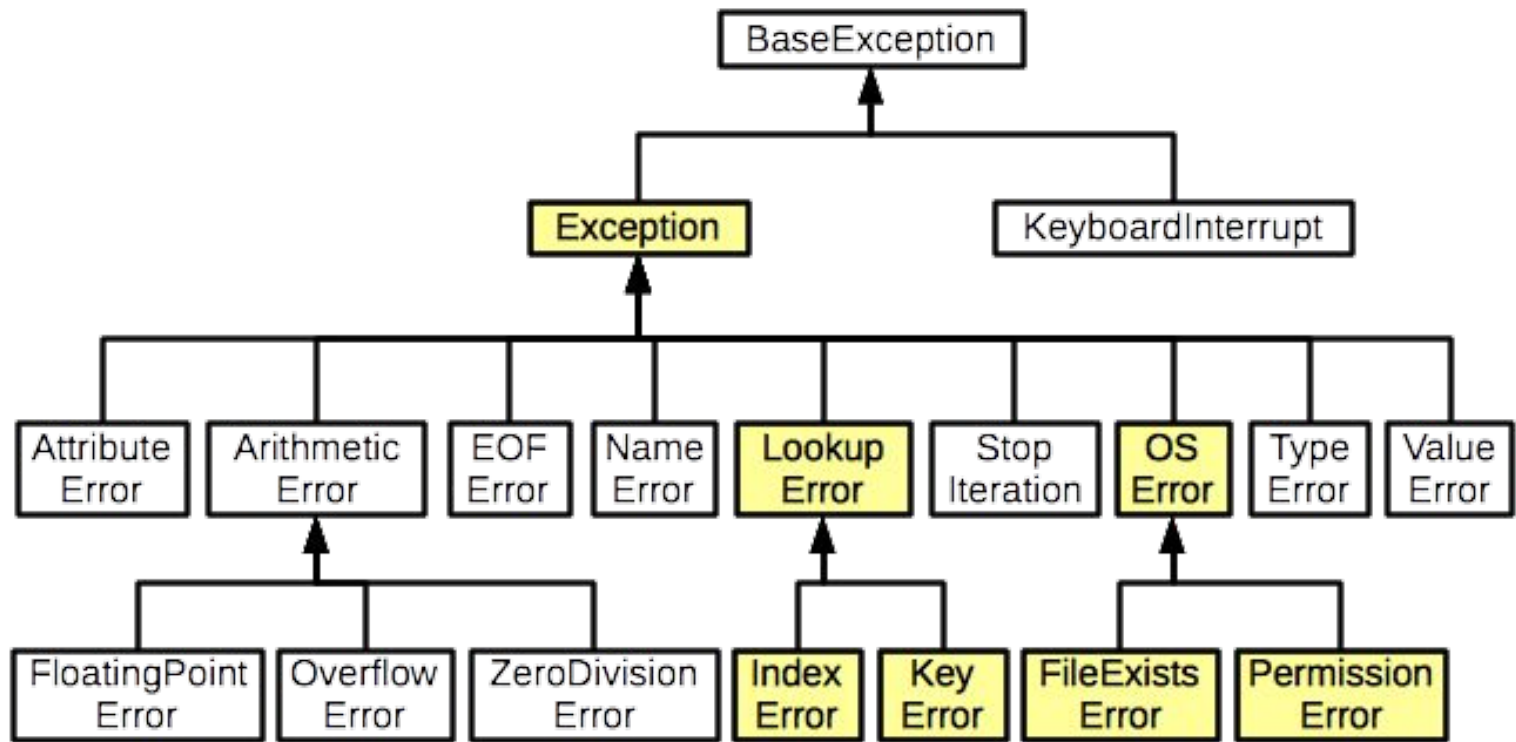
- Em linguagens com suporte a OO, exceções são normalmente modeladas como classes
- Em python, essa classe se chama Exception
- Diversas **subclasses herdam** essa base: referência completa aqui

A Classe Exception

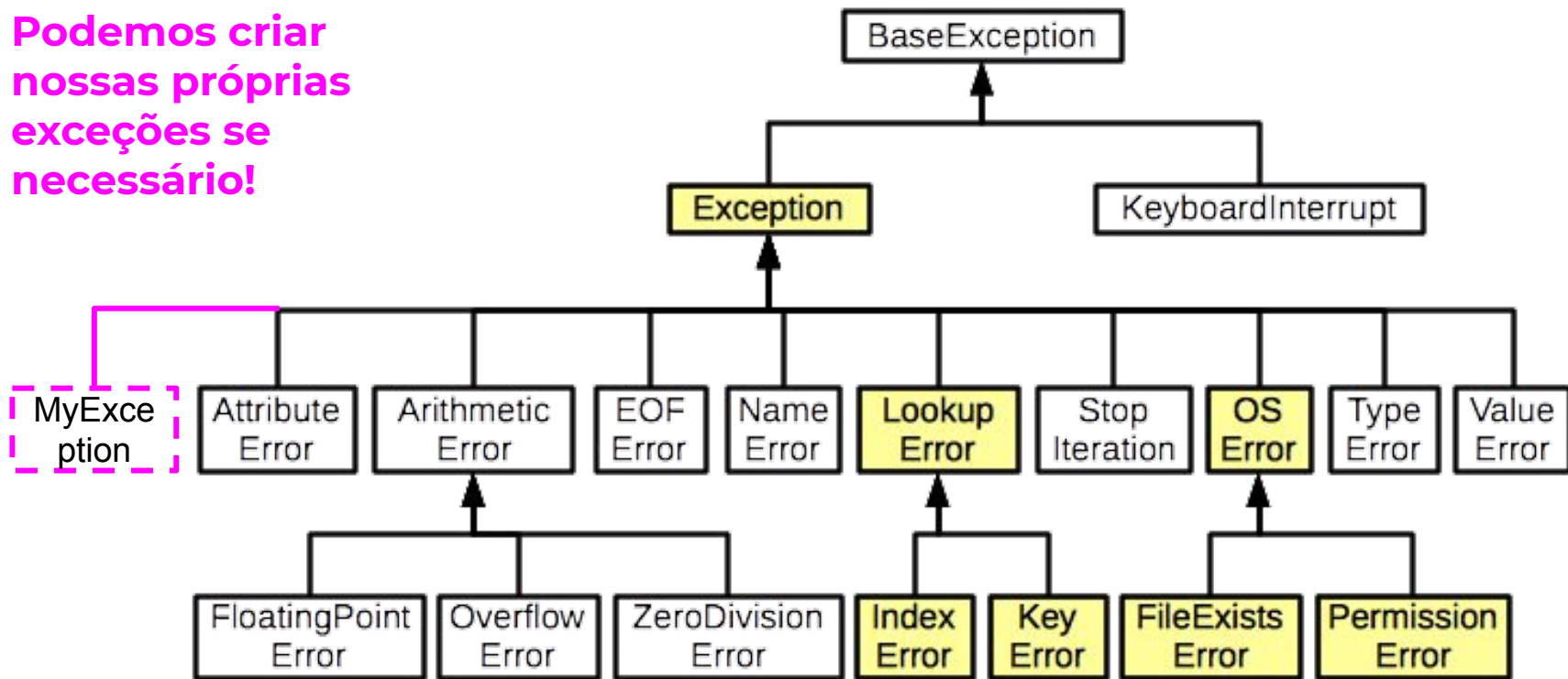
- Em linguagens com suporte a OO, exceções são normalmente modeladas como classes
- Em python, essa classe se chama Exception
- Diversas **subclasses herdam** essa base: referência completa aqui

Tem inclusive uma **exceção específica para Windows...**
por que será? 🤔

exception **WindowsError**
Only available on Windows.



Podemos criar
nossas próprias
exceções se
necessário!



Voltando ao nosso exemplo

```
class ExceptionDemo:
    def __init__(self, args):
        self.divide(args)

    def divide(self, array):
        print(self.divideArray(array))

    def divideArray(self, array):
        s1 = array[0]
        s2 = array[1]
        return self.divideStrings(s1, s2);

    def divideStrings(self, s1, s2):
        i1 = int(s1)
        i2 = int(s2)
        return self.divideInts(i1, i2);

    def divideInts(self, i1, i2):
        return i1 / i2
```

Criando um Código mais Defensivo

— — —

```
def divide(self, array):  
    print(self.divideArray(array))
```

```
def divideSafely(self, array):  
    try:  
        print(self.divideArray(array))  
    except IndexError as e:  
        print(e)  
        print("Uso: $python app.py <num1> <num2>")  
    except ValueError as e:  
        print(e)  
        print("Oops! Use valores inteiros.")  
    except ArithmeticError as e:  
        print(e)  
        print("Oops! Não podemos dividir por zero.")
```

Criando um Código mais Defensivo

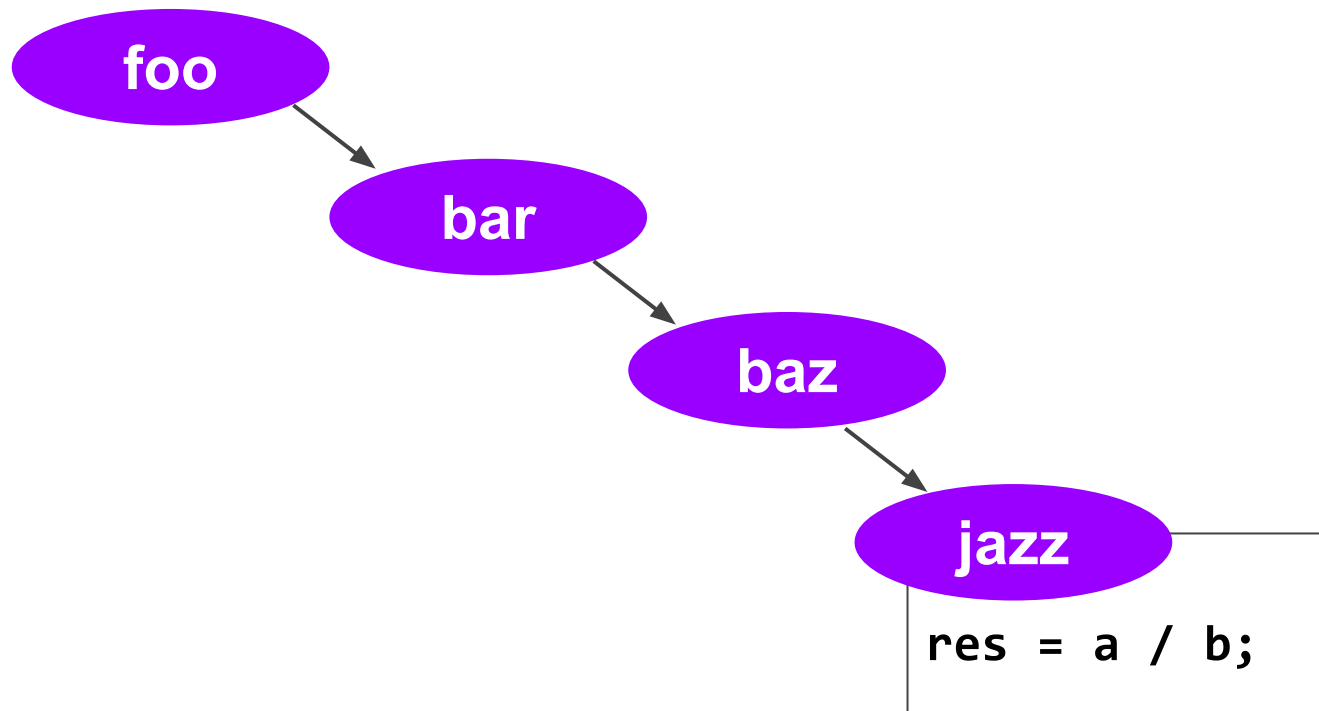
— — —

```
def divide(self, array):  
    print(self.divideArray(array))
```

Mas como vamos
tratar aqui se os
erros ocorrerem em
outros métodos?

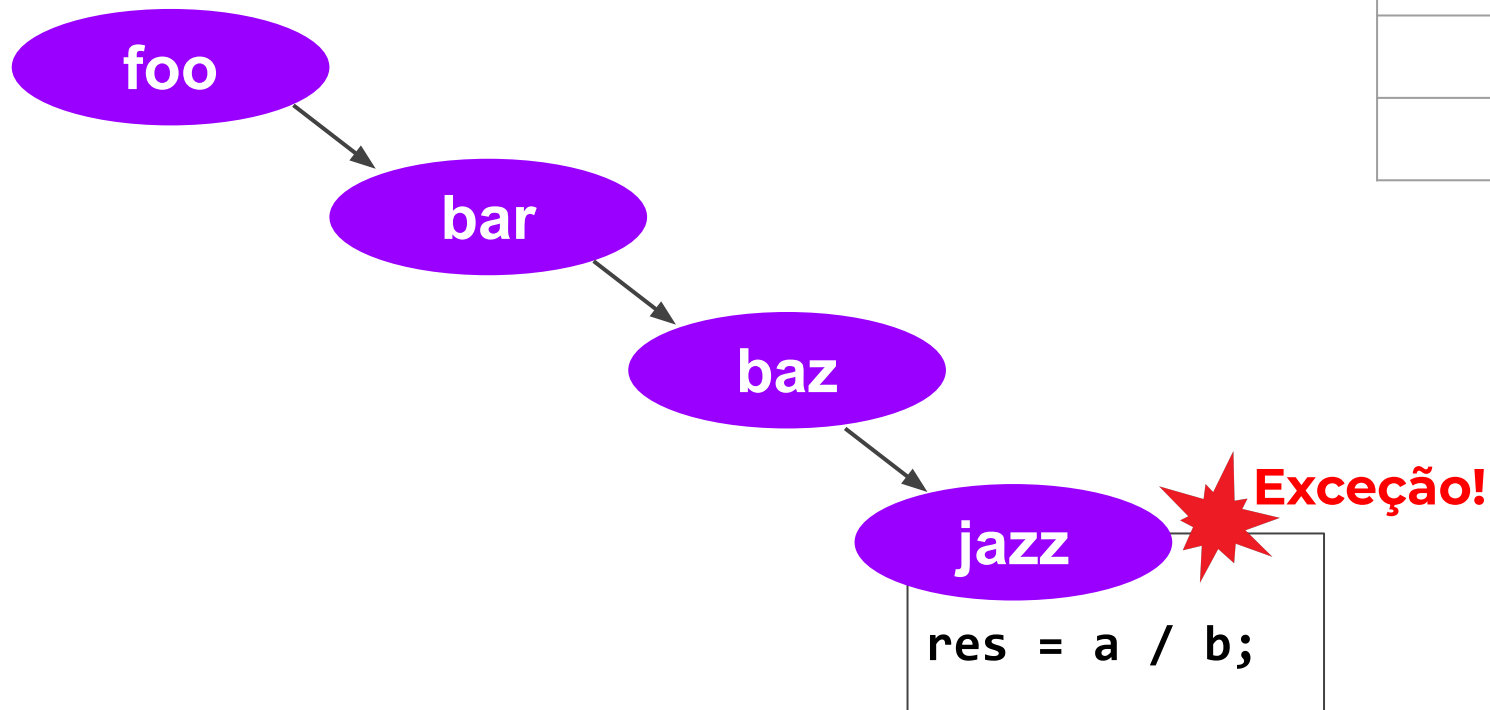
```
def divideSafely(self, array):  
    try:  
        print(self.divideArray(array))  
    except IndexError as e:  
        print(e)  
        print("Uso: $python app.py <num1> <num2>")  
    except ValueError as e:  
        print(e)  
        print("Oops! Use valores inteiros.")  
    except ArithmeticError as e:  
        print(e)  
        print("Oops! Não podemos dividir por zero.")
```

Propagação de Exceções



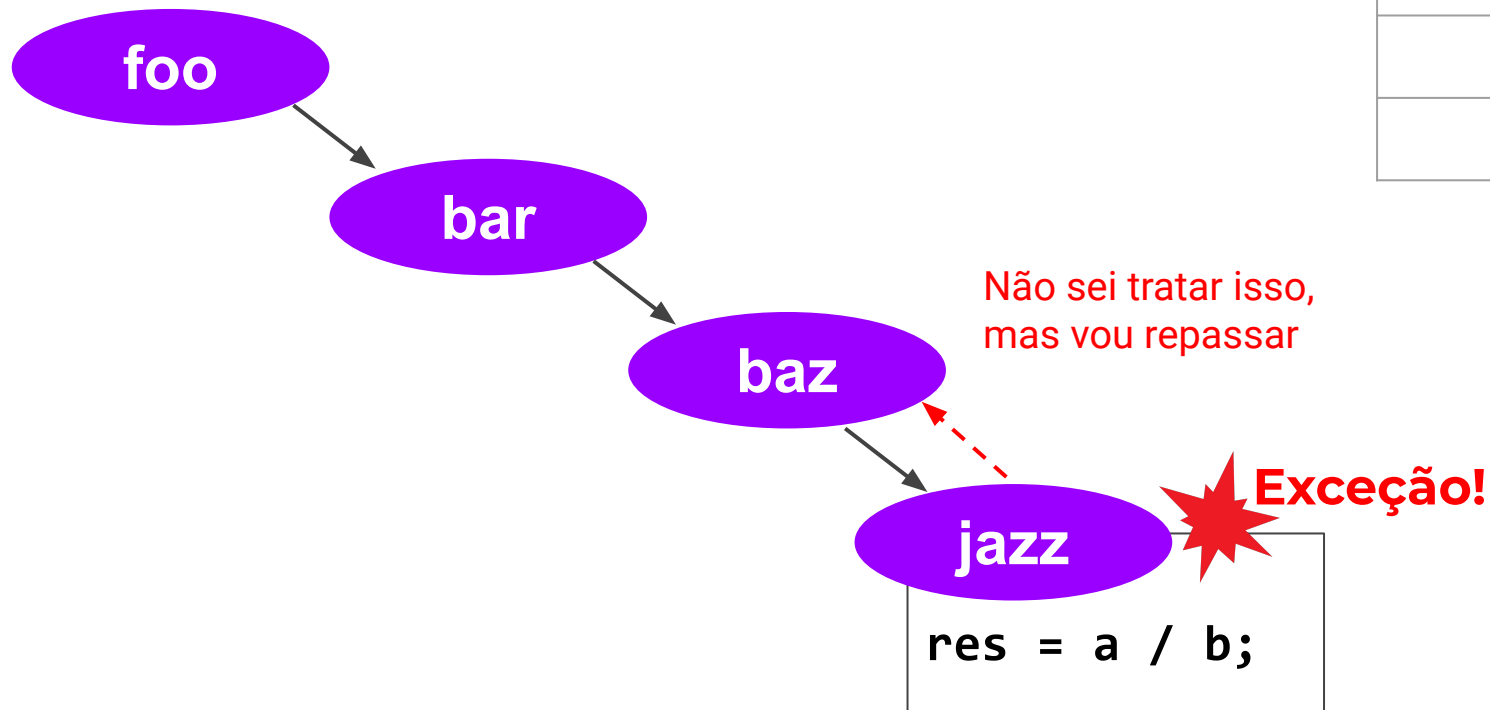
Call Stack
jaz()
baz()
bar()
foo()

Propagação de Exceções



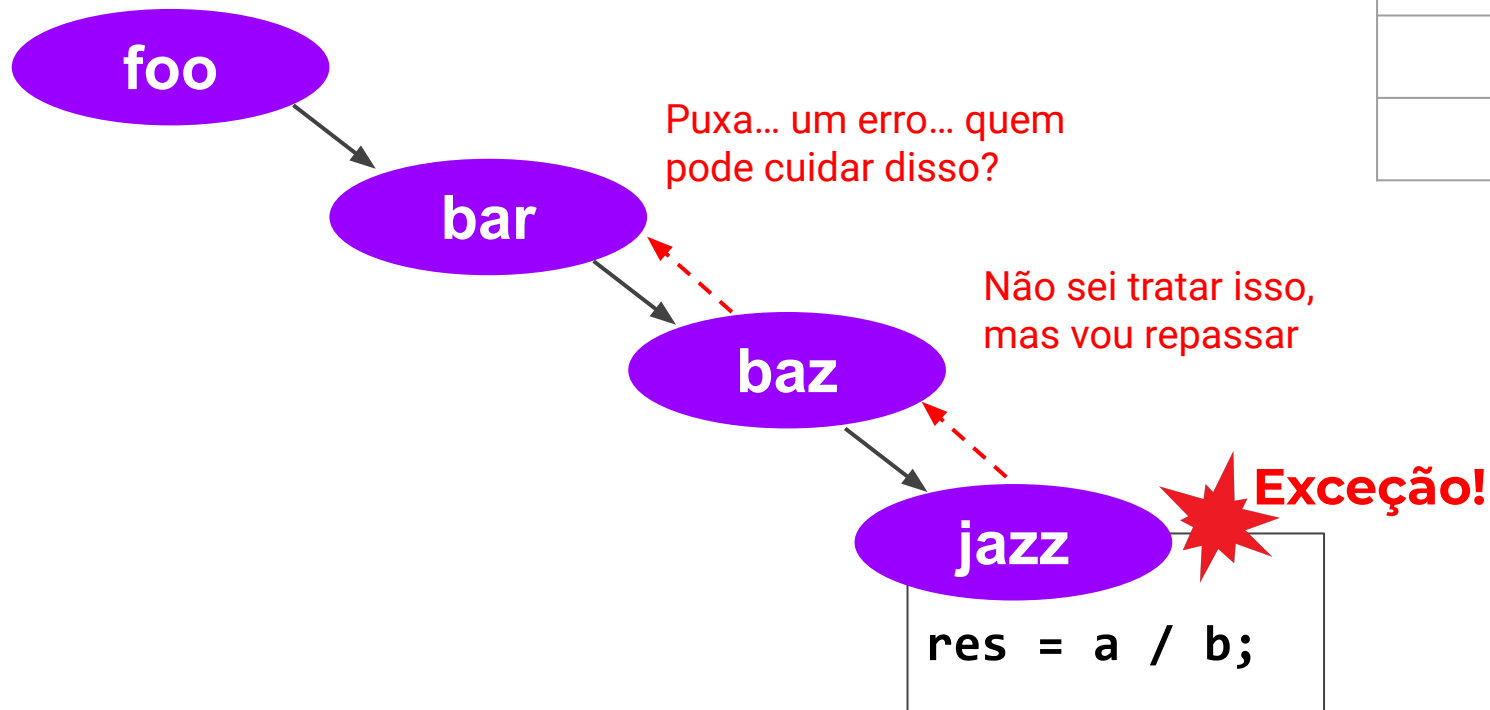
Call Stack
jaz()
baz()
bar()
foo()

Propagação de Exceções



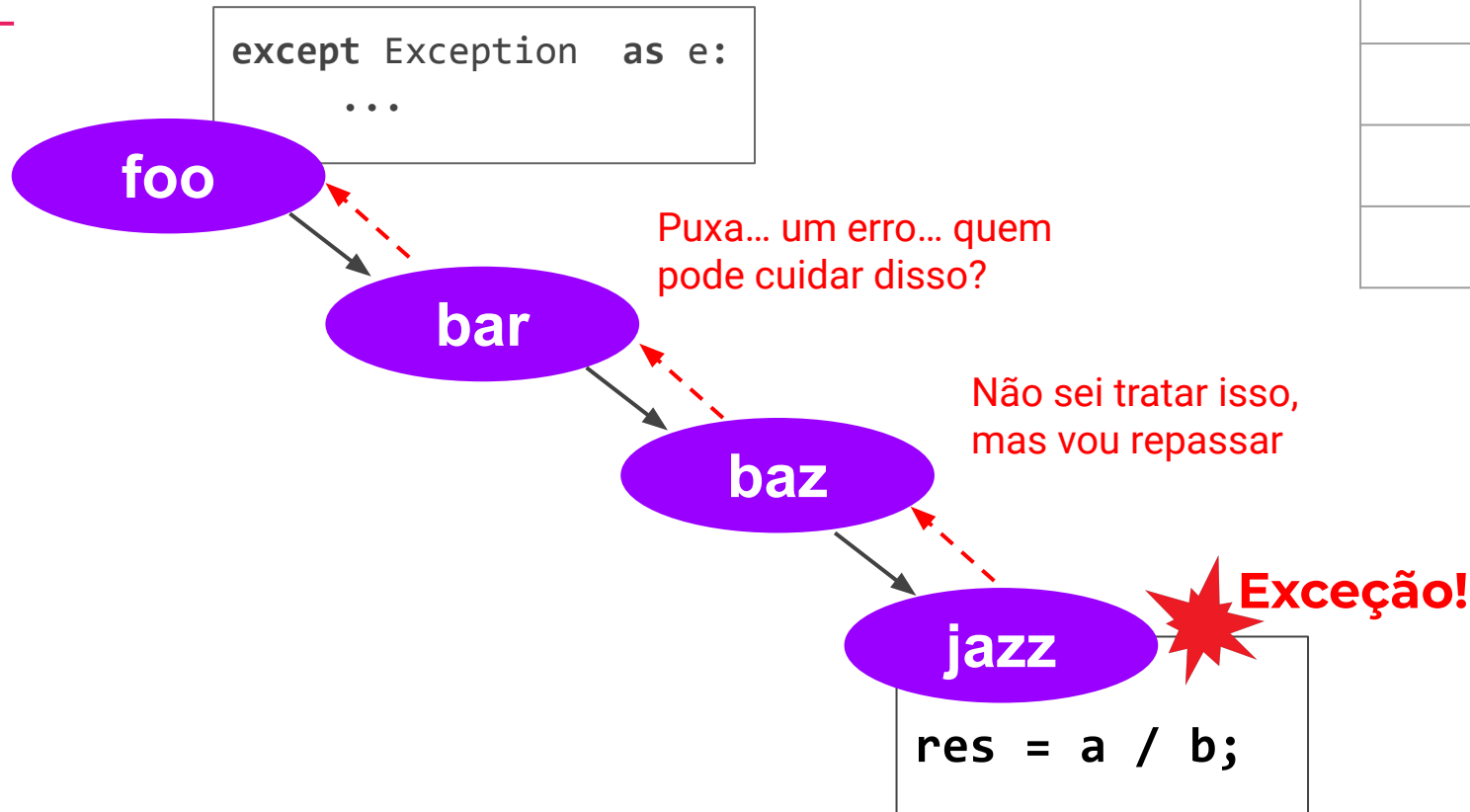
Call Stack
jazz()
baz()
bar()
foo()

Propagação de Exceções



Call Stack
jazz()
baz()
bar()
foo()

Propagação de Exceções



Call Stack
jaz()
baz()
bar()
foo()

**Exceções vão se
propagar na call
stack até serem
capturadas**

**Até a
próxima!**

Exceções: Por que e quando?



Java - Checked Exceptions

- Java criou um mecanismo chamado **checked exceptions** que obriga programadores a tratarem (ou repassarem) as exceções
- Implementações que não fazem isso geram erro de compilação

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("C:\\test\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\\test\\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```



```
import java.io.*;

class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("C:\\test\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

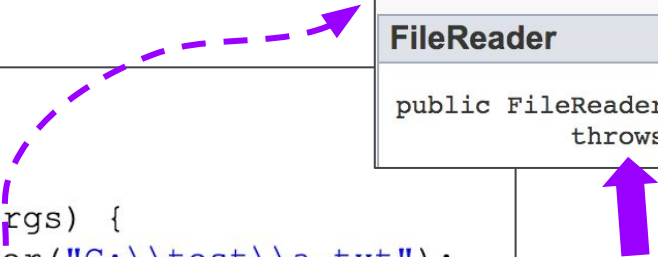
        // Print first 3 lines of file "C:\\test\\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

Constructor Detail

FileReader

```
public FileReader(String fileName)
    throws FileNotFoundException
```



```
import java.io.*;

class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("C:\\test\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\\test\\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

Constructor Detail

FileReader

```
public FileReader(String fileName)
    throws FileNotFoundException
```

```
Exception in thread "main" java.lang.RuntimeException: Uncompilable source
unreported exception java.io.FileNotFoundException; must be caught or decl
thrown
    at Main.main(Main.java:5)
```

[Fonte](#)

Exceções - Overheads

Tratar exceções custa caro em **tempo** e em legibilidade

Benchmark	Time	CPU	Iterations
BM_exitWithBasicException	1407 ns	1407 ns	491232
BM_exitWithMessageException	1605 ns	1605 ns	431393
BM_exitWithReturn	142 ns	142 ns	5172121
BM_exitWithErrorCode	144 ns	143 ns	5069378

[Fonte](#)

A documentação da linguagem diz que blocos try **são eficientes**. O problema é o bloco de exceção.

How fast are exceptions?

A try/except block is extremely efficient if no exceptions are raised. Actually catching an exception is expensive. In versions of Python prior to 2.0 it was common to use this idiom:

```
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

This only made sense when you expected the dict to have the key almost all the time. If that wasn't the case, you coded it like this:

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

*Veja as discussões [aqui](#) e [aqui](#).

Exceções - Por que e quando usar?

— — —

Por quê?

- exceções impedem que os erros sejam ignorados ou esquecidos
- o mecanismo de propagação permite que os erros possam ser tratados em outras etapas do fluxo de execução

Ler: [relatório técnico de desempenho do C++](#)
(seção 5.4)

Exceções - Por que e quando usar?

Por quê?

- exceções impedem que os erros sejam ignorados ou esquecidos
- o mecanismo de propagação permite que os erros possam ser tratados em outras etapas do fluxo de execução

Ler: [relatório técnico de desempenho do C++](#)
(seção 5.4)

Quando?

- Situações que são de fato excepcionais
- Aplicações que serão reutilizadas por outros clientes (API)
- Aplicações em que resiliência a erros é fundamental e mais importante do que desempenho

Antipadrões de Tratamento de Exceções

- 1) **Exceções silenciosas:** ignoram as exceções

```
try:  
    data = unsafeFetch()  
    ...  
except:  
    pass
```

```
from collections import namedtuple

Bread = namedtuple('Bread', 'color')

class ToastException(Exception):
    pass

def toast(bread):
    try:
        put_in_toaster(bread)
    except:
        raise ToastException('Could not toast bread')

def put_in_toaster(bread):
    bread.color = 'light_brown' # Note the typo

toast(Bread('yellow'))
```



```
from collections import namedtuple
```

```
Bread = namedtuple('Bread', ['color'])
```

```
class ToastException(Exception):  
    pass
```

```
def toast(bread):
```

```
    try:
```

```
        put_in_toaster(bread)
```

```
    except:
```

```
        raise ToastException('Could not toast bread')
```

```
def put_in_toaster(bread):
```

```
    bread.color = 'light_brown' # Note the typo
```

```
toast(Bread('yellow'))
```

Traceback (most recent call last):

File "python-examples/reraise_exceptions.py", line 19, in <module>
 toast(Bread('yellow'))

File "python-examples/reraise_exceptions.py", line 12, in toast
 raise ToastException('Could not toast bread')

__main__.ToastException: Could not toast bread -- ???

Antipadrões de Tratamento de Exceções

2) Exceções pouco específicas/mensagens misteriosas:
impedem detecção e resolução o problema

```
try:  
    data = unsafeFetch()  
    ...  
except Exception as e:  
    print("Deu erro")
```

Leiam também

[The most Diabolical Python Anti-Pattern](#)

[Performance Overhead of Exceptions in C++](#)

[Exceptional Logging of Exceptions in Python](#)