

# Tutorial 1 – Parte A

## Exceções



Unidade Curricular de  
Laboratório de Programação

2020/2021

# Avaliação dos alunos

Como já viram no Moodle, nesta semana não terão que fazer um exercício adicional ao tutorial.

A vossa avaliação será através de *screenshots* que vocês irão fazendo ao longo do tutorial, que depois deverão entregar no Moodle, reunidos num zip chamado `ExcecoesScreenshots.zip`.

Deverão estar atentos, durante o tutorial, para a figura na margem da página.



Esta figura chama a atenção para ações que devem fazer que são importantes para a avaliação.

# Exceções

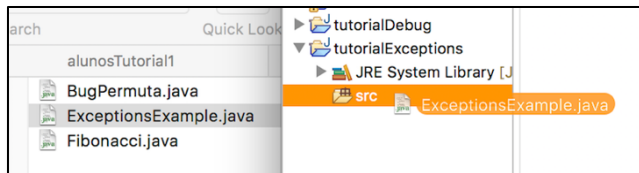
Se não o fez já, descarregue o ficheiro `alunosTutorial1.zip` acessível na página de LabP para o seu disco.

Descompacte o zip, obtendo a pasta `alunosTutorial1`.

No Eclipse, crie um projeto java (**File**→**New**→**Java Project**).

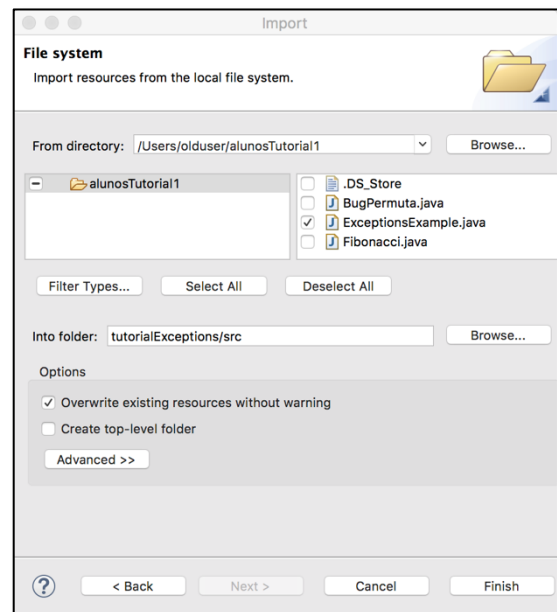
Agora vai adicionar-lhe a classe `ExceptionsExample` que está incluída naquela pasta. Pode fazer isto de várias formas:

- “arrastando” o ficheiro `ExceptionsExample.java`, no sistema de ficheiros, para cima da pasta `src` do novo projeto, no Eclipse;



**OU**

- Com o novo projeto selecionado, escolher **File**→**Import**
  - Na janela seguinte escolher **General/File System** e botão “Next”;
  - De seguida clicar em “Browse” no “From Directory” e seleccionar a pasta `alunosTutorial1` e botão “Open”;
  - Nas caixas, seleccionar `ExceptionsExample.java`
  - De seguida clicar em “Browse” no “Into folder” e seleccionar a pasta `src` do projeto `tutorialExceptions` e botão “Open”;
  - Clicar em “Finish”.



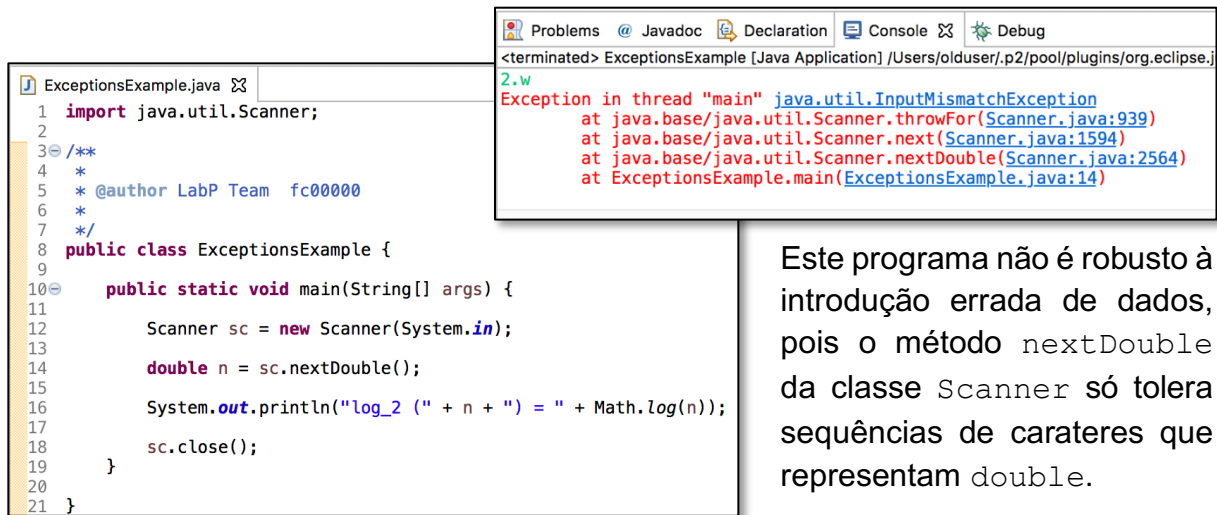
Verifique que já tem a classe `ExceptionsExample` na pasta `src` do seu novo projeto.

Clique duas vezes na classe para a abrir no editor do Eclipse.

Altere os nomes e número no `@author` para o seu próprio nome e número.



O main da classe `ExceptionsExample` lê um valor dado pelo utilizador e imprime no ecrã o seu logaritmo na base 2. Experimente executá-lo e introduza os caracteres `2.w` por exemplo.



Este programa não é robusto à introdução errada de dados, pois o método `nextDouble` da classe `Scanner` só tolera seqüências de caracteres que representam `double`.

Na API da classe `Scanner`, pode ver-se, em relação ao `nextDouble`:

#### `nextDouble`

```
public double nextDouble()
```

Scans the next token of the input as a double. This method will throw `InputMismatchException` if the next token cannot be translated into a valid double value. If the translation is successful, the scanner advances past the input that matched.

O mesmo se aplica aos métodos `nextInt`, `nextFloat`, etc.

Na disciplina de Introdução à Programação contornou-se este problema recorrendo a outros métodos da classe `Scanner` (por exemplo `hasNext`, `hasNextInt`, `hasNextDouble`) que permitem observar o canal de entrada antes de decidir ler o valor que lá está.

Outras exceções são de certeza já suas conhecidas:

`NumberFormatException`, `NullPointerException`,  
`ArrayIndexOutOfBoundsException`, `FileNotFoundException`.

Outra abordagem possível é permitir que o erro ocorra e reagir de forma adequada, sem deixar que o programa termine abruptamente. Isso é feito através do tratamento das exceções, como veremos neste tutorial.

“Exception” ou exceção, é uma abreviatura para “*Exceptional Event*”, que é um evento que ocorre durante a execução de um programa e que interrompe o seu fluxo normal.

Quando, durante a execução do programa, é detetado um erro:

- uma *exceção é lançada (throwing an exception)*, ou seja, é criado um objeto Java que contém as informações relevantes sobre o erro ocorrido:
  - o seu tipo e
  - o estado do programa quando o erro ocorreu.
- o sistema tenta lidar com o problema
  - recorre à lista ordenada de métodos que foram invocados para chegar ao método onde o erro ocorreu (*pilha de chamadas*) e
  - vê se algum desses métodos é capaz de *tratar a exceção (handle the exception)*.

Na figura da página anterior pode ver a pilha de chamadas que é apresentada na consola quando a exceção ocorre.

Existem várias alternativas para lidar com as exceções, como se explica neste guião.

## Lidar com uma exceção

Por norma, um método que pode lançar uma exceção tem de assumir o tratamento dessa mesma exceção. Existem duas formas de um método tratar uma exceção:

1. Apanhar a exceção, incluindo a instrução `try-catch` que permite tratar a exceção da forma que se ache adequada;
2. (Re)lançar a exceção para o método que o invocou, acrescentando a cláusula `throws` na assinatura do método.

Se todos os métodos na pilha de chamadas também relançarem a exceção, incluindo o `main` onde tudo começou, o programa termina abruptamente indicando, como na figura anterior, qual a exceção ocorrida e a pilha de chamadas.

Vamos agora estudar estas duas formas de tratamento de exceções.

# 1. Apanhar uma exceção (try-catch)

O mecanismo base para tratar exceções em Java é o bloco de instruções `try-catch`, composto por:

- um bloco `try`:
  - neste bloco tenta-se executar um conjunto de instruções que, por poderem correr mal, estão dentro deste bloco;
- um ou mais blocos `catch` a serem executados quando alguma das instruções no bloco `try` provoca o lançamento de uma exceção.

Um bloco `catch` recebe a exceção como parâmetro.

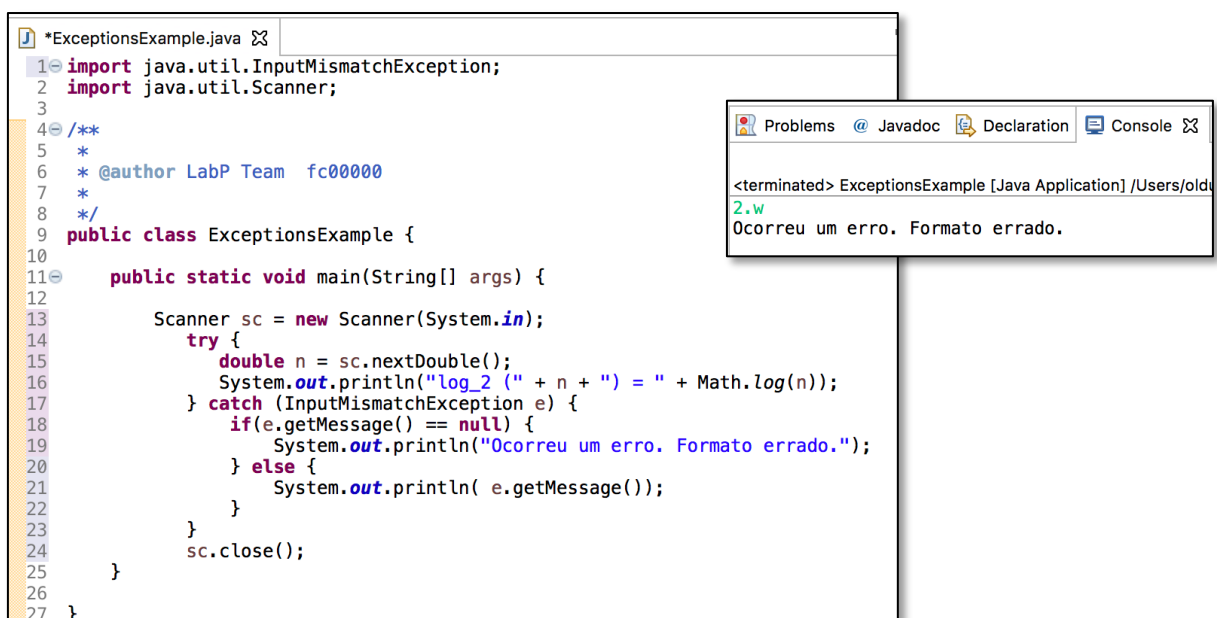
Em cada destes blocos podemos ter, em alternativa:

- a. código que vai ser executado para resolver o erro (por exemplo, informando o utilizador de que aconteceu algo errado durante a execução do programa);
- b. propagação da exceção, eventualmente de forma diferente (outro tipo de exceção), para o método que invocou o método corrente.

## 1.a. Resolver a exceção

Como exemplo da alternativa **a.**, quando é lançada a exceção `InputMismatchException` pelo método `nextDouble`, podemos “apanhar” essa exceção e informar o utilizador que o programa terminou sem o resultado esperado, porque um dos valores que o programa estava à espera de receber não é do tipo `double`.

Faça estas alterações à sua classe e execute de novo o programa, voltando a introduzir um valor que não é um número.



```
*ExceptionsExample.java
1 import java.util.InputMismatchException;
2 import java.util.Scanner;
3
4 /**
5  *
6  * @author LabP Team fc00000
7  *
8  */
9 public class ExceptionsExample {
10
11     public static void main(String[] args) {
12
13         Scanner sc = new Scanner(System.in);
14         try {
15             double n = sc.nextDouble();
16             System.out.println("log_2 (" + n + ") = " + Math.log(n));
17         } catch (InputMismatchException e) {
18             if(e.getMessage() == null) {
19                 System.out.println("Ocorreu um erro. Formato errado.");
20             } else {
21                 System.out.println( e.getMessage());
22             }
23         }
24         sc.close();
25     }
26
27 }
```

Problems Javadoc Declaration Console

<terminated> ExceptionsExample [Java Application] /Users/old  
2.w  
Ocorreu um erro. Formato errado.

Faça agora o seu primeiro *screenshot* do ambiente Eclipse, com atenção para apanhar a linha do `@author`, que já deverá conter o seu nome e número. Dê o nome `exceptionScreenshot1` ao seu ficheiro.



## 1.b. Lançar uma exceção (`throw`)

A solução apresentada acima resolve o problema quando algo corre mal com a aquisição do valor de `n`, contudo não resolve todos os problemas!

Por exemplo, o que acontece quando o número `n` é um número não positivo? Apesar de o programa não dar erro nem lançar uma exceção (devolve `NAN` ou `Infinity`), pode-se interpretar esta situação como um comportamento excecional e que, portanto, deve dar origem a uma exceção.

A criação de uma exceção é feita, como com qualquer outro objeto, com a instrução `new`. O construtor recebe uma *string* que pode depois ser acedida através da invocação do método `getMessage` da exceção.

O lançamento de uma exceção é feito usando a instrução `throw`.

No exemplo seguinte, o comportamento excecional é sinalizado lançando uma exceção do tipo `InputMismatchException`. Faça estas alterações à sua classe e execute de novo o programa, introduzindo o valor `-1`.

```
import java.util.InputMismatchException;
import java.util.Scanner;

/**
 * @author LabP Team fc00000
 */
public class ExceptionsExample {

    public static final double ERRO = 0.001;

    public static void main (String[] args){

        Scanner sc = new Scanner(System.in);

        try {
            double n = sc.nextDouble();

            if(n <= ERRO) {
                throw new InputMismatchException("Ocorreu um erro.\n" +
                    "O argumento nao eh positivo!");
            }

            System.out.println("log_2 (" + n + ") = " + Math.log(n));
        } catch (InputMismatchException e) {
            if(e.getMessage() == null) {
                System.out.println("Ocorreu um erro. Formato errado.");
            } else {
                System.out.println(e.getMessage());
            }
        }
        sc.close();
    }
}
```

```
<terminated> ExceptionsExample [Java Application] /Users/old
-1
Ocorreu um erro.
O argumento nao eh positivo!
```

Faça agora o seu segundo *screenshot* do ambiente Eclipse, com atenção para que a linha do `@author` também apareça. Dê o nome `exceptionScreenshot2` ao seu ficheiro.



## 2. (Re)lançar uma exceção para outro método (`throws`)

Quando não queremos apanhar uma exceção num método `m`, temos que “anunciar” que a execução de `m` pode levar ao seu lançamento.

Para isso temos que acrescentar à assinatura do método a palavra `throws` (notar o “s” no final da palavra) seguida do nome das exceções potencialmente geradas e não tratadas.

Note que **quando o método gera uma exceção que não é tratada internamente a sua execução termina de imediato.**

No exemplo seguinte, o método `lerDeUmFicheiro` que lê um valor a partir de um ficheiro, não trata as exceções do tipo `FileNotFoundException` e, por isso, tem na sua assinatura a declaração `throws FileNotFoundException`.

```
public static double lerDeUmFicheiro(String nomeFicheiro)
    throws FileNotFoundException {
    double n = 0;

    Scanner sc = new Scanner(new File(nomeFicheiro));
    n = sc.nextDouble();
    if(n <= ERRO) {
        throw new InputMismatchException("O argumento eh negativo!");
    }
    sc.close();
    return n;
}
```

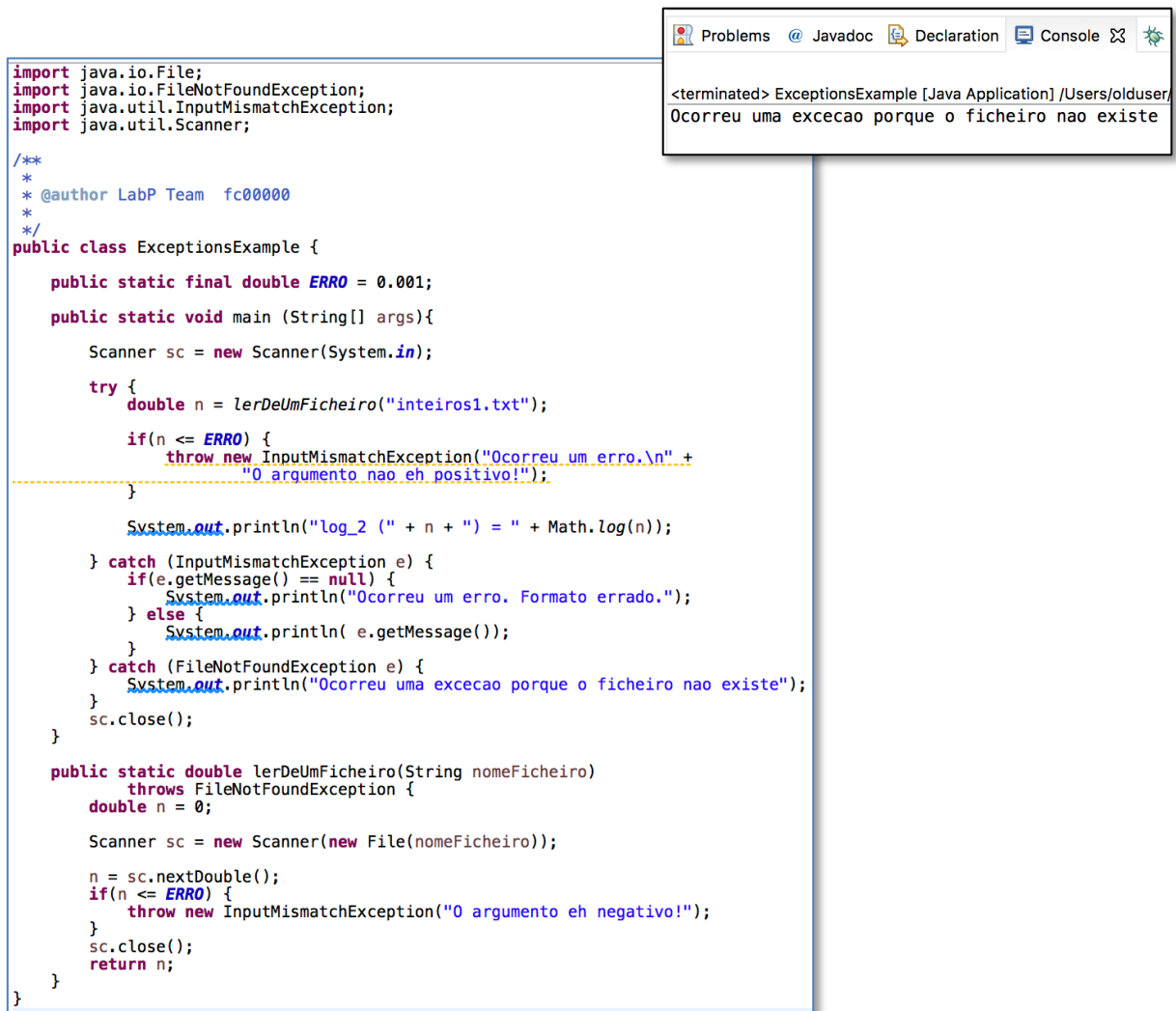
No caso de ocorrer uma exceção provocado por erros de acesso ao ficheiro, ela é passada para o método que invocou este método e assim sucessivamente, até encontrar um método que trate a exceção ou, em última instância, até chegar ao método `main`. Neste caso, se o `main` não trata as exceções (também tem `throws`), o programa termina a sua execução indicando uma mensagem de erro.

Acrescente este método à sua classe e altere a instrução `double n = sc.nextDouble();` do `main` por `double n = lerDeUmFicheiro("inteiros1.txt");`

Acrescente também um bloco `catch` no `main`, para tratar a exceção que a invocação a este novo método pode provocar.



Execute o programa completo.



The screenshot shows the Eclipse IDE with a Java file named `ExceptionsExample.java` open. The code is as follows:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.InputMismatchException;
import java.util.Scanner;

/**
 * @author LabP Team fc00000
 */
public class ExceptionsExample {
    public static final double ERRO = 0.001;
    public static void main (String[] args){
        Scanner sc = new Scanner(System.in);
        try {
            double n = lerDeUmFicheiro("inteiros1.txt");
            if(n <= ERRO) {
                throw new InputMismatchException("Ocorreu um erro.\n" +
                    "O argumento nao eh positivo!");
            }
            System.out.println("log_2 (" + n + ") = " + Math.log(n));
        } catch (InputMismatchException e) {
            if(e.getMessage() == null) {
                System.out.println("Ocorreu um erro. Formato errado.");
            } else {
                System.out.println(e.getMessage());
            }
        } catch (FileNotFoundException e) {
            System.out.println("Ocorreu uma excecao porque o ficheiro nao existe");
        }
        sc.close();
    }

    public static double lerDeUmFicheiro(String nomeFicheiro)
        throws FileNotFoundException {
        double n = 0;
        Scanner sc = new Scanner(new File(nomeFicheiro));
        n = sc.nextDouble();
        if(n <= ERRO) {
            throw new InputMismatchException("O argumento eh negativo!");
        }
        sc.close();
        return n;
    }
}
```

The console output shows the following message:

```
<terminated> ExceptionsExample [Java Application] /Users/olduser/
Ocorreu uma excecao porque o ficheiro nao existe
```

Faça agora o seu terceiro *screenshot* do ambiente Eclipse (sempre com atenção para que a linha do `@author` também apareça). Dê o nome `exceptionScreenshot3` ao seu ficheiro.



## Try-with-resources em Java

Ainda tomando o exemplo dado, pense no que acontece ao `Scanner` quando o método `lerDeUmFicheiro` termina lançando a exceção `InputMismatch` causada pelo facto do valor introduzido pelo utilizador ser negativo. Se assim for, as instruções `sc.close()` e `return n` não são executadas e portanto o `Scanner` fica “aberto”.

É necessário fechá-lo, isto é libertar este recurso para outras execuções!

De modo a simplificar a tarefa dos programadores, a instrução `try` pode ser reforçada com a indicação dos recursos que serão fechados de forma automática no final do bloco (mesmo que uma exceção seja lançada).

De notar que no bloco `try` pode haver vários recursos que necessitem de ser terminados. A indicação desses vários recursos é feita através de separação com `;` dentro do parêntesis que se segue à palavra reservada `try`. Assim uma solução mais adequada é:

```
public static double lerDeUmFicheiro(String nomeFicheiro)
    throws FileNotFoundException {
    double n = 0;

    try(Scanner sc = new Scanner(new File(nomeFicheiro)))
    {
        n = sc.nextDouble();
        if(n <= 0) {
            throw new InputMismatchException("O argumento eh negativo!");
        }
        return n;
    }
}
```

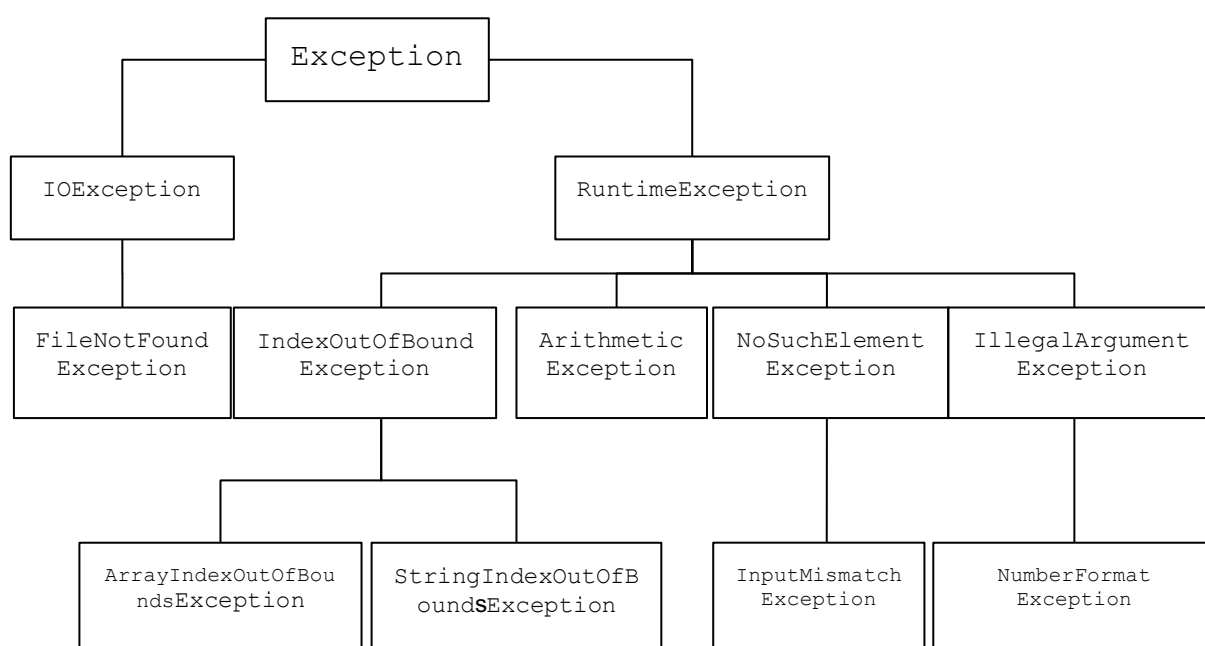
Note que assim não necessita de invocar o método `close()` do `Scanner`.

# Apanhar exceções genéricas vs Apanhar exceções específicas

Quando um programa lança e trata vários tipos de exceções, devemos organizar os `catch` dos diferentes tipos de possíveis exceções de acordo com o grau de especificidade de cada uma das exceções.

Assim, para que o programa dê a melhor informação possível ao utilizador sobre o erro que causou o fim do programa, os blocos `catch` devem ser ordenados do mais específico para o mais genérico, isto é, começando por tratar as exceções mais abaixo na figura que se segue e só depois as mais acima.

Em DCO estudará esta questão mais em pormenor.



Crie uma nova classe no seu projeto, com um método `main` contendo a seguinte instrução:

```
organizeExceptionsExample("myInput.txt", "myOutput.txt");
```

De seguida copie o método da página que se segue para essa nova classe. Já sabe que vai ter que corrigir os erros devidos à necessidade de declarações de `import`.

```

/**
 * Read integers from a texto file e write their logarithm, by reverse order,
 * into another file.
 * @param fileInName - The name of the file containing the integers
 * @param fileOutName - The name of the file where the logs are to be written.
 * @requires fileInName!= null && fileOutName!= null
 */
public static void organizeExceptionsExample(String fileInName,
                                             String fileOutName) {

    try (Scanner sc = new Scanner(new File(fileInName));
         PrintWriter out = new PrintWriter(fileOutName)) {

        Stack<Integer> myStack = new Stack<Integer>();

        while(sc.hasNextLine()) {
            String newLine = sc.nextLine();
            myStack.push(Integer.parseInt(newLine));
        }

        for (int number : myStack) {
            Double x = Math.log(number);
            out.write(x.toString());
        }

    } catch (NumberFormatException e) {
        System.out.println("Nao consegue converter num Integer");
    } catch (InputMismatchException e) {
        System.out.println("O valor nao tem o formato esperado");
    } catch (RuntimeException e) {
        System.out.println("Durante a execucao algo correu mal");
    } catch (FileNotFoundException e) {
        System.out.println("O ficheiro nao existe!");
    } catch (Exception e) {
        System.out.println("Algo correu mal!");
    }
}

```

Agora execute o programa. Como não existe nenhum ficheiro `myInput.txt`, na consola aparecerá a mensagem “O ficheiro não existe!”.

Faça agora o seu quarto *screenshot* do ambiente Eclipse (sempre com atenção para que a linha do `@author` também apareça). Dê o nome `exceptionScreenshot4` ao seu ficheiro.



Continuando...

Agora crie o ficheiro `myInput.txt` (menu **File** → **New** → **File** indicando o local onde deve ficar localizado e de seguida definir o seu conteúdo, usando a janela do editor do Eclipse).

Experimente alterar o conteúdo do ficheiro `myInput.txt` de modo a conseguir produzir os vários tipos de exceções tratadas nos vários blocos `catch`.

Existe ainda a possibilidade de, caso as exceções apanhadas serem de tipos não comparáveis e tratadas da mesma forma, isto é, com o mesmo código, simplificar o código juntando os dois tipos no mesmo bloco.

Suponhamos que, no exemplo anterior, se pretende que, para as exceções do tipo `NumberFormatException` e `InputMismatchException`, a mensagem enviada ao utilizador seja `"O input não tem o formato pretendido"`.

Podemos escrever:

```
catch (NumberFormatException e) {  
    System.out.println("O input não tem o formato pretendido");  
} catch (InputMismatchException e) {  
    System.out.println("O input não tem o formato pretendido");  
}
```

Mas, de forma mais compacta, poderíamos ter:

```
catch (NumberFormatException | InputMismatchException e) {  
    System.out.println("O input não tem o formato pretendido");  
}
```

## O bloco `finally`

O bloco `finally` é usado sempre que se pretende garantir que um dado bloco de instruções é executado mesmo que seja lançada uma exceção. Antes do Java 7, o papel de fecho dos recursos cabia ao programador e as instruções necessárias eram realizadas dentro de um bloco `finally`.

A partir de então, essa tarefa é automática com o `try-with-resources`, como já foi referido.

Contudo, o bloco `finally` não perdeu por completo a sua utilidade. Ainda é usado, por exemplo, para finalizar tarefas importantes de escrita (por exemplo em bases de dados), ou eliminação de ficheiros que foram criados como meios auxiliares ou colocar estruturas num estado específico.

```

/**
 * Using a texto file to fill a stack with integers
 * @param myStack The stack to fill with numbers
 * @param fileName The original file
 * @throws FileNotFoundException
 * @requires myStack != null && fileName!= null
 */
public void finallyExample(Stack<Integer> myStack, String fileName)
    throws FileNotFoundException{

    myStack = new Stack<Integer>();

    try (Scanner sc = new Scanner(new File(fileName))) {
        if(sc.hasNextLine()){
            myStack.push(sc.nextInt());
        }
    } catch (FileNotFoundException e){
        System.out.println("O ficheiro nao existe");
    } catch (InputMismatchException e) {
        System.out.println("O valor nao tem o formato esperado");
    } finally{
        if(myStack.isEmpty()){
            myStack = null;
        }
    }
}

```

Deve consultar a documentação da classe `Exception`, em particular dos métodos `getMessage` e `getStackTrace`.

Para complementar a informação deste guião pode consultar o capítulo 9 do livro *Java: Introduction to Problem Solving and Programming* da bibliografia da unidade curricular.

## Entrega para avaliação:

Os 4 ficheiros com os *screenshots* que fez durante este tutorial devem ser colocados numa pasta de nome `ExceptionScreenshots`.

Um *zip* desta pasta deve ser entregue no Moodle, juntamente com os *screenshots* do tutorial de Debug.

O material aqui descrito baseia-se num tutorial de anos anteriores e todos os créditos são devidos aos seus autores.