Introdução a Java

//Exceções, classes utilitárias

IT BOARDING

BOOTCAMP

Índice

Java

IT BOARDING

BOOTCAMP

Exceções

IT BOARDING

BOOTCAMP

“Um programa pode falhar por vários motivos, alguns podem ser causados por erros no código, outros são completamente fora de nosso controle. As exceções são eventos que alteram o fluxo do programa”.

Tipos de exceções

// Exceções verificadas (Checked)

São aquelas que derivam da classe **Exception**. Devem ser tratadas ou declaradas, ou seja, **requerem o uso do bloco try / catch**. Entre as classes mais comuns, encontramos:

- **FileNotFoundException** → *lançada programaticamente quando o código tenta fazer referência a um arquivo que não existe*
- **IOException** → *lançada quando há um problema de leitura ou gravação de um arquivo*

// Exceções não verificadas (Unchecked)

Derivam da classe **RuntimeException**. Elas não devem ser manipulados ou declarados, ou seja, **não requerem** o uso obrigatório do bloco try / catch. Os mais comuns são:

- **ArrayIndexOutOfBoundsException** → *lançada pela JVM ao usar um índice ilegal ao acessar uma matriz*
- **IllegalArgumentException** → *lançada pelo programador para indicar que um argumento impróprio ou não permitido foi passado para um método*
- **NullPointerException** → *lançada pela JVM quando a referência a um objeto é nula no momento da solicitação do objeto*

Tipos de exceções

Exceções não verificadas e verificadas:

```
int valor = 0;

double resultado = 10 / valor;
```

```
try {
   FileInputStream fileInputStream = new
FileInputStream("prova.txt");
} catch (FileNotFoundException exception) {
   System.out.println("O arquivo indicado não existe");
}
```

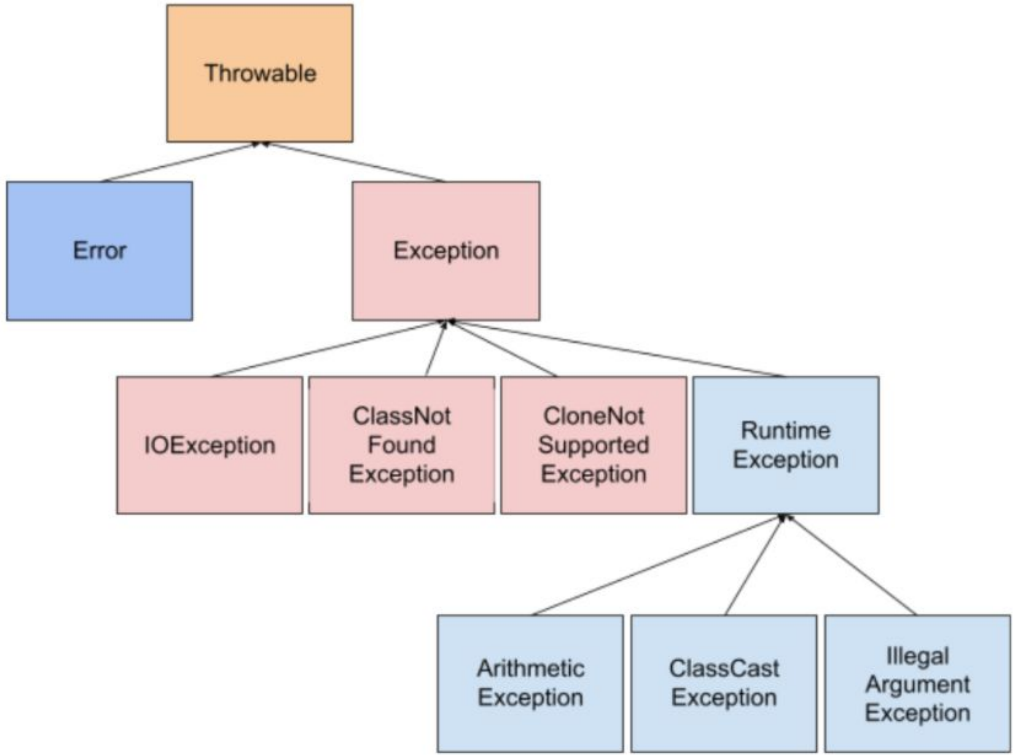
Erros

Os erros derivam da classe **Error**, são conhecidos por serem lançados pela JVM e não podem ser resolvidos ou corrigidos, portanto, o programa geralmente para. Eles indicam problemas sérios em nosso aplicativo, geralmente são raros, mas você pode ver alguns dos seguintes:

- **ExceptionInInitializerError** → *lançado pela JVM quando um inicializador estático lança uma exceção e não é tratado*
- **StackOverflowError** → *lançada pela JVM quando um método chama a si mesmo muitas vezes (isso é chamado de recursão infinita porque o método geralmente chama a si mesmo indefinidamente)*
- **NoClassDefFoundError** → *lançada pela JVM quando uma classe que o código usa está disponível em tempo de compilação, mas não em tempo de execução*

Hierarquia de exceções

Em Java, todas as exceções são representadas por **classes**. As classes de exceção derivam de uma classe chamada **Throwable**. Como podemos ver abaixo, existe uma hierarquia de exceções



Stack Trace

O rastreamento de pilha, ou **pilha de chamadas**, é uma lista de chamadas de métodos que foram feitas no aplicativo quando uma exceção foi lançada. Exibido a partir do método mais recente em que a exceção foi lançada. Esta ordem permite identificar mais facilmente a causa principal da falha, pois é mais provável que a encontremos nos métodos mais recentemente executados.

```
try {
    FileInputStream fileInputStream = new FileInputStream("prova.txt");
} catch (FileNotFoundException exception) {
    exception.printStackTrace();
}
```

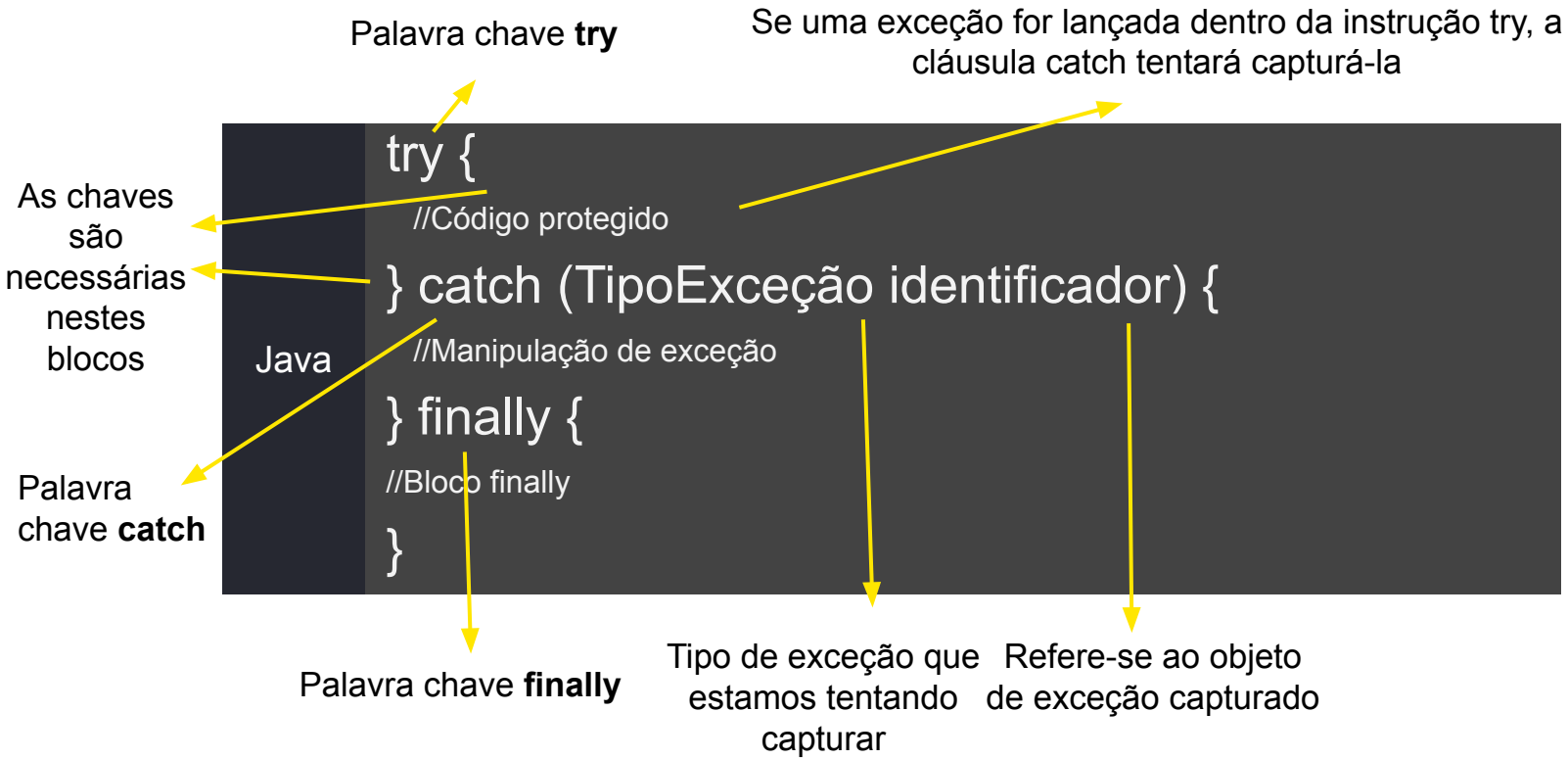
Experimente este trecho de código em seu IDE! Olha o que está impresso no console ...

Tratamento de exceções

O Java nos permite controlar exceções para que nosso programa **continue sua execução** mesmo se ocorrer uma exceção. Para isso, temos a estrutura try-catch-finally.

- Bloco **try** → *Significa "tentar" em inglês. Todo código que vai dentro deste bloco tentará executar uma operação sujeita à ocorrência de uma exceção.*
- Bloco **catch** → Significa "pegar" em inglês. *Aqui nós definimos o conjunto de instruções necessárias para o tratamento da exceção.*
- Bloco **finally** → Significa "finalmente" em inglês. *Neste bloco podemos definir um conjunto de instruções necessárias que serão executadas quer ocorra a exceção ou não, pois é **SEMPRE** executado.*

Sintaxe de estrutura try - catch - finally



Exceções

O código no bloco try é executado normalmente; se alguma de suas linhas lançar uma exceção, o bloco try para e a execução das instruções no bloco catch começa.

Se nenhuma das instruções no bloco try lançar uma exceção que possa ser detectada, a cláusula catch não será executada.

Existem dois caminhos possíveis quando há um bloco catch e um finally. Se uma exceção for lançada, o bloco finally será executado após o bloco catch. Se nenhuma exceção for lançada, o bloco finally será executado após o bloco try.

De acordo com o que foi revisto até agora...

Dado o seguinte bloco de código ... Compila corretamente? Por quê?

```
{}
try
        inserirNaBaseDados();
catch (Exception e)
        System.out.println(“Erro ao inserir os dados”);
```

A resposta é **NÃO**... o problema é que as chaves são necessárias em ambos os blocos.

```
try {
    insertarEnBaseDeDatos();
} catch (Exception e) {
    System.out.println("Error al insertar los datos");
}
```

O que você acha do código a seguir?

Compila corretamente? Por quê?

```
try {
        inserirNaBaseDeDados();
}
```

Neste caso ele **NÃO** compila, porque o bloco try não é seguido por mais nada, lembre-se que o objetivo do bloco try é realizar alguma ação no caso de uma exceção ser lançada. Sem outra cláusula de acompanhamento, a instrução try não tem sentido.

```
try {
    insertarEnBaseDeDatos();
} catch (Exception e) {
    System.out.println("Error al insertar los datos");
}
```

Java

Vamos revisar esses exemplos

Qual deles está correto?

```
try {
        registrarUsuario();
} finally {
        encerrarConexao();
} catch (Exception e) {
System.out. println("Erro ao registrar usuario");
}
```

INCORRECTO

```
try {
        registrarUsuario();
} catch (Exception e) {
System.out. println("Erro ao registrar usuario");
} finally {
        encerrarConexion();
}
```

CORRECTO

Esta é a opção correta, lembre-se que os blocos catch e finally devem estar na **ordem correta**

Java

Vamos analisar um caso específico

A seguir, vamos tentar fazer uma divisão por zero e ver como uma exceção é lançada:

```java
public class Divisor {

    public static void main(String[] args) {
        System.out.println("Antes de hacer la división");

        double division = 5 / 0;

        System.out.println("Después de la división");
    }
}
```

```
Divisor
"C:\Program Files\Java\jdk-11.0.11\bin\java.exe" "-javaagent:C:\Program Files\JetBr
Antes de hacer la división
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at Divisor.main(Divisor.java:7)

Process finished with exit code 1
```

Java

Como podemos ver, nosso programa executa a primeira instrução corretamente indicando que estamos no momento antes de fazer a divisão, mas ao atingir a segunda instrução é lançada uma **ArithmeticException** e a execução do programa para sem chegar à última linha.

Vamos ver o que acontece quando tratamos essa exceção para que a execução de nosso programa continue, apesar deste inconveniente.

Para saber que tipo de exceção capturar, devemos saber a classe da exceção que pode ser lançada, neste caso será do tipo **ArithmeticException**, que devemos capturar no bloco catch. Vamos ver como fica nosso programa.

```java
public class Divisor {

    public static void main(String[] args) {
        System.out.println("Antes de hacer la división");

        try {
            double division = 5 / 0;
        } catch (ArithmeticException exception) {
            System.out.println("Error en la división: " + exception.getMessage());
        } finally {
            System.out.println("Después de la división");
        }
    }
}
```

```
Divisor
"C:\Program Files\Java\jdk-11.0.11\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2021.1.1\lib\id
Antes de hacer la división
Error en la división: / by zero
Después de la división
```

Throw

A palavra reservada **throw** nos permite lançar uma exceção, ela deve ser seguida pelo operador **new** e o **tipo de exceção** que queremos lançar. A execução para imediatamente após a instrução throw, portanto, nenhuma das instruções seguintes serão executadas:

```java
public class Excepciones {
    private static int dividendo = 5;
    private static int divisor = 0;

    public static void main(String[] args) {
        dividir();
    }

    public static void dividir() {
        try {
            if (divisor == 0)
                throw new IllegalArgumentException("No se puede dividir por cero");
        } catch (IllegalArgumentException exception) {
            exception.printStackTrace();
        }
    }
}
```

```
"C:\Program Files\Java\jdk-11.0.11\bin\java.exe" "-javaagent:C:\Program Files\J
java.lang.IllegalArgumentException Create breakpoint : No se puede dividir por cero
    at Excepciones.dividir(Excepciones.java:13)
    at Excepciones.main(Excepciones.java:7)

Process finished with exit code 0
```

Clases Utilitárias

IT BOARDING

BOOTCAMP

As Classes Utilitárias definem um conjunto de métodos que realizam funções que têm a ver entre si, e são, geralmente muito utilizadas. A maioria dessas classes define métodos estáticos. Por exemplo, a classe **java.lang.Math**

```
public class Math {

public static double cos(double a) {...}
public static double tan(double a) {...}
public static double pow(double a, double b) {...}
public static long round(double a) {...}

}
```

```
//Fazendo uso dos métodos

double cosseno = Math.cos(30);
double tangente = Math.tan(15);
double potencia = Math.pow(2, 5);
long arredondamento = Math.round(20.5);
```

Outro exemplo de classes utilitárias pode ser a classe LocalDateTime, que serve para lidar com datas e horas, e que foi introduzida a partir do Java 8 para solucionar problemas das classes Date e Time usadas anteriormente.

```
package java.time;

public final class LocalDateTime ... {
public static LocalDateTime now() {...}
public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond) {...}
public static LocalDateTime parse(CharSequence text) {...}
```

```
//Fazendo uso dos métodos

LocalDateTime ldt1 = LocalDateTime.now();

LocalDateTime ldt2 = LocalDateTime.of(2021, Month.MAY, 20, 8, 20, 15, 11);

LocalDateTime ldt3 = LocalDateTime.parse("2021-08-03T10:15:30");
```

Obrigado

IT BOARDING

BOOTCAMP