



Exceções e tratamento

Prof. Renato Pimentel

2023/2



Sumário



1 Exceções e tratamento

No tratamento de problemas usando OO (e mesmo outros paradigmas de programação), precisamos obedecer certas **regras**, ou **restrições** que fazem parte do problema.

No caso de OO, podemos mencionar, por exemplo, *regras de negócio* a serem respeitadas na implementação dos métodos:

Exemplo: `ca1.defineDiaDoMes(35)`; – data inválida: método deve evitar atribuição do valor 35 como dia do mês do objeto `ca1`.

Como avisar quem chamou o método de que não foi possível realizar determinada operação?

Uma opção natural seria usar o **retorno** do método, porém há problemas...

Imagine a situação a seguir:

```
1 public Cliente procuraCliente(int id) {
2     if (id invalido) {
3         // avisa o método que chamou este que ocorreu
4         // um erro
5     } else {
6         Cliente cliente = new Cliente();
7         cliente.setId(id);
8         cliente.setNome("nome do cliente");
9         return cliente;
10    }
```

Não é possível descobrir se houve erro através do retorno, pois método retorna um objeto da classe Cliente.

Outro caso:

```
1 Conta minhaConta = new Conta();
2 minhaConta.deposita(100);
3 // ...
4 double valor = 5000; // valor acima do saldo (100) +
5 // limite
6 minhaConta.saca(valor); // sacar: booleano. Vai
7 // retornar false, mas verificação é ignorada!
8 caixaEletronico.emite(valor);
```



Nos casos anteriores, podemos ter uma situação onde o id do cliente é inválido, ou o valor para saque é muito alto, ou mesmo inválido (ex. negativo), o que configuram **exceções** a tais regras de negócio.

Exceção

Algo que normalmente não ocorre – não deveria ocorrer – indicando algo estranho ou inesperado no sistema.



Quando uma exceção é *lançada* (**throw**):

- JVM verifica se método em execução toma precaução para tentar contornar situação:
 - ▶ Se precaução não é tomada, a execução do método **para** e o teste é repetido no método anterior – aquele que chamou o método problemático.
 - ▶ Se o método que chamou não toma precaução alguma, tal processo é repetido até que se alcance o método `main()`;
 - ▶ Por fim, se `main()` não trata o erro, a JVM “morre”.



Tratando exceções: mecanismo *try/catch*.

- **Try**: *tentativa* de executar trecho onde pode ocorrer problema;
- **Catch**: quando exceção é lançada em tal trecho, a mesma é *capturada*. Seu tratamento é feito no bloco do **catch**.



Exemplo – **sem** *try/catch*

```
1 class TesteErro {
2     public static void main(String[] args) {
3         int[] array = new int[10];
4         for (int i = 0; i <= 15; i++) {
5             array[i] = i;
6             System.out.println(i);
7         }
8     }
9 }
```



Saída:

```
0
1
2
3
4
5
6
7
8
9
Exception in thread "main" java.lang.
    ArrayIndexOutOfBoundsException: 10
at excecoes.TesteErro.main(TesteErro.java:5)
```



Adicionando *try/catch* no código anterior:

```
1 class TesteErro {
2     public static void main(String[] args) {
3         int[] array = new int[10];
4         for (int i = 0; i <= 15; i++) {
5             try {
6                 array[i] = i;
7                 System.out.println(i);
8             } catch (ArrayIndexOutOfBoundsException e) {
9                 System.out.println("Erro: " + e);
10            }
11        }
12    }
13 }
```



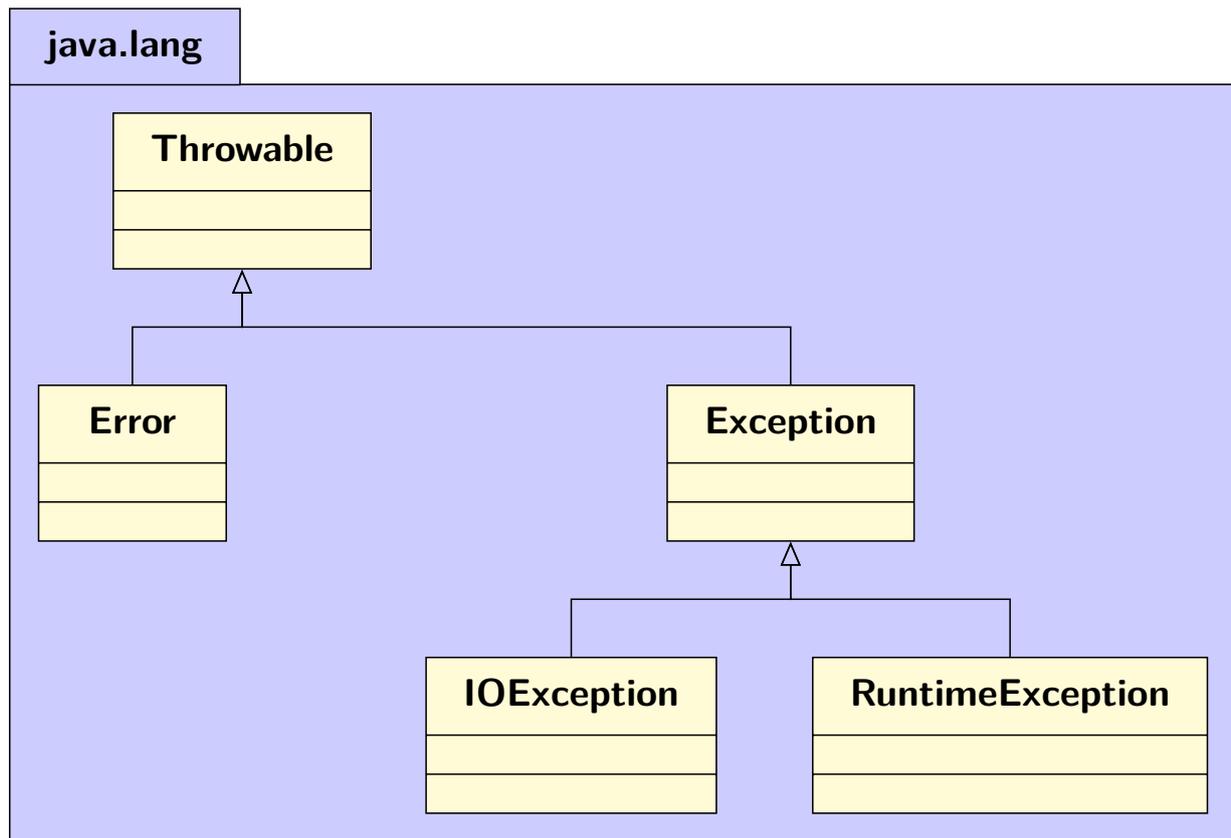
Saída:

```
...  
7  
8  
9  
Erro: java.lang.ArrayIndexOutOfBoundsException: 10  
Erro: java.lang.ArrayIndexOutOfBoundsException: 11  
Erro: java.lang.ArrayIndexOutOfBoundsException: 12  
Erro: java.lang.ArrayIndexOutOfBoundsException: 13  
Erro: java.lang.ArrayIndexOutOfBoundsException: 14  
Erro: java.lang.ArrayIndexOutOfBoundsException: 15
```



Note que quando a exceção é capturada, a execução do método `main()` procede normalmente, até o encerramento do programa.

A exceção `ArrayIndexOutOfBoundsException` é, na verdade, uma **classe** contida dentro do pacote `java.lang`.



A hierarquia segue além: `ArrayIndexOutOfBoundsException`, por exemplo, é subclasse de `RuntimeException`. Outros exemplos de subclasses incluem:

- `ArithmeticException`. Ocorre, por exemplo, quando se faz divisão por 0;
- `NullPointerException`. Referência nula.



Família Throwable III



A classe `Error` define um tipo de erro causado pelo sistema, ex.: `StackOverflowError`, `OutOfMemoryError`, que **não pode ser lançado** diretamente pelo programador;

Mais detalhes: <https://docs.oracle.com/javase/8/docs/api/java/lang/Error.html>

A hierarquia `Exception`, por sua vez, se divide em **vários ramos**.

`RuntimeException` – os erros em tempo de execução, de **captura não obrigatória** (*unchecked exception* ou **exceção não-verificada**); e

`IOException` – relacionado à entrada e saída de dados – são apenas dois exemplos.

Todas as exceções que não são `RuntimeException`, ou suas subclasses, devem ser capturadas e tratadas (*checked exceptions* ou **exceções verificadas**).

Mais detalhes: <https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html>



Família Throwable IV



Em caso de não tratamento das exceções *checked*, o compilador acusará erro, impedindo a geração do *bytecode*.

Exemplo:

```
1 class Excecoes {
2     public static void main(String[] args) {
3         new java.io.FileInputStream("arquivo.txt");
4     }
5 }
```

Pode ocorrer `FileNotFoundException` (subclasse de `IOException`). NetBeans sugere duas formas de tratamento, que veremos a seguir.



Quando uma exceção é lançada, há duas formas de tratamento, do ponto de vista de um **método**:

- Envolvê-la em um *try-catch* (*surround with try-catch*), como já foi visto.
- Delegar o tratamento da exceção para o método que chamou o atual (*add throws clause*).



Try-catch:

```
1 import java.io.FileNotFoundException;
2
3 class Excecoes {
4     public static void main(String[] args) {
5         try {
6             new java.io.FileInputStream("arquivo.txt");
7         } catch (FileNotFoundException ex) {
8             System.out.println("Não foi possível abrir o arquivo
9             para leitura.");
10        }
11    }
```



Passando o tratamento para o método que invocou o atual (cláusula `throws`):

```
1 import java.io.FileNotFoundException;
2 class Excecoes {
3     public static void main(String[] args) throws
4         FileNotFoundException {
5         new java.io.FileInputStream("arquivo.txt");
6     }
}
```

É desnecessário tratar no `throws` as *unchecked exceptions*, porém é permitido, e pode facilitar a leitura e documentação do código.



A propagação pode ser feita através de vários métodos:

```
1 public void teste1() throws FileNotFoundException {
2     FileReader stream = new FileReader("c:\\teste.txt");
3 }
4 public void teste2() throws FileNotFoundException {
5     teste1();
6 }
7 public void teste3() {
8     try {
9         teste2();
10    } catch(FileNotFoundException e) {
11        // ...
12    }
13 }
```



É possível tratar mais de uma exceção simultaneamente.

- Com o *try-catch*:

```
1 try {
2     objeto.metodoQuePodeLancarIOeSQLException();
3 } catch (IOException e) {
4     // ...
5 } catch (SQLException e) {
6     // ...
7 }
```



- Com a cláusula *throws*:

```
1 public void abre(String arquivo) throws IOException,
2     SQLException {
3     // ..
4 }
```



- Ou mesmo uma combinação de ambos: uma exceção tratada no próprio método (*try-catch*) em combinação com a cláusula **throws**:

```
1 public void abre(String arquivo) throws IOException {  
2     try {  
3         objeto.metodoQuePodeLancarIOeSQLException();  
4     } catch (SQLException e) {  
5         // ..  
6     }  
7 }
```



Lançando exceções I



Até o momento, abordou-se o tratamento de exceções que ocorrem em métodos ou mecanismos do Java, como acesso a elementos de *arrays* e métodos relacionados a arquivos (entrada/saída).

Como **lançar** (*to throw*) as exceções em Java?



Por exemplo:

```
1 boolean saca(double valor) {
2     if (this.saldo < valor) {
3         return false;
4     } else {
5         this.saldo -= valor;
6         return true;
7     }
8 }
```



Como mostrado previamente, método acima não foi tratado por quem o chamou:

```
1 Conta minhaConta = new Conta();
2 minhaConta.deposita(100);
3 // ...
4 double valor = 5000; // valor acima do saldo (100) +
5                       limite
6 minhaConta.saca(valor); // saca: booleano. Vai
7                           retornar false, mas verificação é ignorada!
8 caixaEletronico.emite(valor);
```



Podemos lançar uma exceção usando a palavra reservada `throw`. Veja como ficaria o método `saca()` visto anteriormente com o lançamento da exceção:

```
1 void saca(double valor) {
2     if (this.saldo < valor) {
3         throw new RuntimeException();
4     } else {
5         this.saldo -= valor;
6     }
7 }
```

Lançamos uma exceção `RuntimeException()`, que *pode* ser tratada por quem chamou o método `saca()` (*unchecked exception*).



Desvantagem:

`RuntimeException()` – muito genérica: como saber onde exatamente ocorreu problema?

Algo mais específico:

```
1 void saca(double valor) {
2     if (this.saldo < valor) {
3         throw new IllegalArgumentException();
4     } else {
5         this.saldo -= valor;
6     }
7 }
```

`IllegalArgumentException()` (subclasse de `RuntimeException()`) informa que o parâmetro passado ao método foi ruim (exemplo: valor negativo, valor acima do saldo, etc.)



No método que chamou `saca()`, pode-se, por exemplo, usar *try-catch* para tentar “laçar” a exceção lançada pelo mesmo:

```
1 Conta minhaConta = new Conta();
2 minhaConta.deposita(100);
3 try {
4     minhaConta.saca(100);
5 } catch (IllegalArgumentException e) {
6     System.out.println("Saldo insuficiente ou valor
7     inválido.");
8 }
```



Outra forma: passar no *construtor* da exceção qual o problema – usando `String`:

```
1 void saca(double valor) {
2     if (this.saldo < valor) {
3         throw new IllegalArgumentException("Saldo insuficiente ou
4         valor inválido");
5     } else {
6         this.saldo -= valor;
7     }
8 }
```



Neste caso, o que ocorreu pode ser recuperado com o método `getMessage()` da classe `Throwable`.

```
1 try {
2     cc.saca(100);
3 } catch (IllegalArgumentException e) {
4     System.out.println(e.getMessage());
5 }
```



Observação

Todo método que lança uma exceção **verificada** deve conter a cláusula **throws** em sua assinatura.

```
1 public void setData(int dia, int mes, int ano) throws
   IOException {
2     if (dia<1 || dia>31) throw new IOException("Dia inválido");
3     if (mes<1 || mes>12) throw new IOException("Mês inválido");
4     this.dia = dia;
5     this.mes = mes;
6     this.ano = ano;
7 }
```



É possível também criar uma **nova classe** de exceção.
Para isso, basta “estender” alguma subclasse de Throwable.



Voltando ao exemplo anterior:

```
1 public class SaldoInsuficienteException extends
   RuntimeException {
2     SaldoInsuficienteException(String message) {
3         super(message);
4     }
5 }
```



Ao invés de se usar `IllegalArgumentException`, pode-se lançar a exceção criada, contendo uma mensagem que dirá Saldo insuficiente, por exemplo:

```
1 void saca(double valor) {
2     if (this.saldo < valor) {
3         throw new SaldoInsuficienteException("Saldo Insuficiente,
4         " + " tente um valor menor");
5     } else {
6         this.saldo -= valor;
7     }
}
```



Testando:

```
1 public static void main(String[] args) {
2     Conta cc = new ContaCorrente();
3     cc.deposita(10);
4
5     try {
6         cc.saca(100);
7     } catch (SaldoInsuficienteException e) {
8         System.out.println(e.getMessage());
9     }
10 }
```



Para transformar a exceção em *checked*, **forçando** o método que chamou `saca()` a tratar a exceção, basta criar a exceção como subclasse de `Exception`, ao invés de `RuntimeException`:

```
1 public class SaldoInsuficienteException extends
   Exception {
2     SaldoInsuficienteException(String message) {
3         super(message);
4     }
5 }
```



finally I



Um bloco *try-catch* pode apresentar uma **terceira cláusula**, indicando o que deve ser feito após um `try` ou `catch` qualquer.

A ideia é, por exemplo, liberar um recurso no `finally`, como fechar um arquivo ou encerrar uma conexão com um banco de dados, independente de algo ter falhado no código: bloco **sempre** executado, independente de exceção ter ocorrido.



Exemplo:

```
1 try {  
2     // bloco try  
3 } catch (IOException ex) {  
4     // bloco catch 1  
5 } catch (SQLException sqlEx) {  
6     // bloco catch 2  
7 } finally {  
8     // bloco que será sempre executado, independente  
9     // se houve ou não exception  
10 }
```



Exercícios I



- 1 Construa um programa que leia 2 valores para que se possa fazer a divisão. No caso, crie uma exceção para tratar o problema de divisão por zero.
- 2 Construa um programa que crie uma classe para tratar a exceção relacionada a um caractere minúsculo em uma String. Faça um programa que em que dada uma String, possa avaliar se há um caractere minúsculo.



- ① Apostila de Java e POO Caelum: disponível em <https://www.caelum.com.br/download/caelum-java-objetos-fj11.pdf> – acesso em: MAI/2017.
- ② Documentação Java Oracle: <https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html>

Os slides de parte desta seção foram cedidos por Marcelo Z. do Nascimento, FACOM/UFU

LaTeXagem e adaptações: Renato Pimentel, FACOM/UFU