

# Sérgio Taborda's Weblog

Alguns ensinam. Alguns fazem. O resto procura nos livros.

- [Início](#)
- [Blog](#)
- [Ciência](#)
- [Desenvolvimento](#)
- [Entretenimento](#)
- [Epistemologia](#)
- [Política](#)
- 



## Exceções: Classes Utilitárias

[Deixe o seu comentário](#) [Go to comments](#)

No fim do artigo anterior <sup>[2]</sup> ficou prometido um conjunto de padrões e classes que o auxiliariam a fazer o correto tratamento de todas as exceções do seu sistema.

Um dos principais problemas ao lidar com exceções é tratá-las correntemente. Se você trata a mesma exceção em muito lugares rapidamente isso se torna chato já que o código vai se repetindo. Esta a principal razão para o programador iniciante rapidamente mandar o tratamento de exceções às urtigas e começar a ignorá-lo ou a cometer uma das más práticas citadas antes.

Se o código de tratamento se repete nada melhor que fazer com ele o mesmo que fazemos com outros códigos que se repetem: colocá-los em um método. Melhor ainda, colocá-los em um método de uma classe.

## ExceptionHandler

A classe para tratar as exceções será na realidade uma interface chamada `ExceptionHandler` desta forma poderemos utilizá-la onde quisermos depois. Apenas necessitamos de um método que trate a exceção. Como vimos antes tratar uma exceção significa a maioria das vezes encapsulá-la em outra exceção, portanto, o tratamento da exceção do método da classe tem que permitir retornar uma nova exceção. O código ficaria mais ou menos assim:

```
1  
2 public interface ExceptionHandler<T extends Throwable, R extends Exception> {
```

```
3
4 public R handle ( T exception ) ;
5 }
```

---

#### Código 1:

Podemos tratar qualquer tipo de exceção, incluindo `Error` mas normalmente trataremos `Exception`. Contudo o retorno tem que ser uma instância de `Exception` ou `RuntimeException` conforme as regras de encapsulamento comentadas no artigo anterior. Não faz, em geral, sentido criarmos um `Error` partindo de outra `Exception`. Para os raros, casos em que fizer podemos escrever o tratamento à mão.

Um exemplo obvio do uso de `ExceptionHandler` é o tratamento de `IOException`. Podemos criar uma `IOExceptionHandler` que traduz as exceções para exceções com mais informação tal como ditam as boas práticas de forma centralizada. A utilização seria mais ou menos assim:

```
1
2 try {
3 // faz alguma coisa que lance IOException
4 } catch ( IOException e ){
5 throw new IOExceptionHandler () .handle ( e ) ;
6 }
```

---

#### Código 2:

O mesmo poderia ser feito para `SQLException`.

Dependendo da camada onde a exceção está acontecendo poderemos utilizar `ExceptionHandler`s diferentes e até nos utilizarmos de mecanismos de herança e sobre-escrita para alterarmos o comportamento do `ExceptionHandler`.

Utilizando `ExceptionHandler`s específicos podemos facilmente seguir a boa prática de “*Não capture o que você não pode segurar*”. Se não temos um `ExceptionHandler` para aquele tipo de exceção, então, muito possivelmente, não sabemos o que fazer com ela e portanto não a podemos tratar devidamente. Nestas circunstâncias basta-nos apenas encapsular a exceção em outra e relançar.

## ExceptionUtils

Uma classe com métodos utilitários à semelhança de `Math` ou `Collections` pode ser usada para as tarefas mais corriqueiras.

```
01
02 public final class ExceptionUtils {
03
04 private ExceptionUtils () {}
05
06 public static RuntimeException toRuntimeException ( Throwable t ) ;
07
08 public static RuntimeException toRuntimeException ( Class<? extends RuntimeException>
runtimeClass,Throwable t ) ;
09
```

```

10 public static Throwable fromRuntimeException ( RuntimeException r ) ;
11
12 }

```

Código 3:

O método `toRuntimeException` simplesmente encapsula uma qualquer exceção dentro de uma `RuntimeException`, com opção de dizer qual a filha específica de `RuntimeException` que deve ser usada. Métodos para fazer um inverso também pode ser adicionados (`fromRuntimeException`) permitindo um controle maior sobre os objetos de exceção.

## Consistência

Os objetos têm um estado, e é da responsabilidade do objeto manter o estado consistente. Para mudar o seu estado o objeto aceita parâmetros do exterior nos métodos que alteram o estado. Nada impede que alguém utilize um valor proibido para o parâmetro corrompendo o estado do objeto. Para se proteger disto o objeto tem que verificar se os argumentos que recebe são corretos. O objeto tem que consistir os parâmetros.

A consistência do estado envolve normalmente o uso de estruturas de decisão e lançamento de exceções. Este princípio está de acordo com a boa prática “*Não deixe para os outros o que você pode lançar primeiro*” mas é normalmente chato de escrever além de aumentar, desnecessariamente, a complexidade e tamanho do método. Para ajudar nesta tarefa uma classe com métodos de consistência pode ser construída. Esta classe tem métodos cujo papel é semelhante aos métodos `assert` do JUnit (para quem não conhece, dê uma olhada neste *framework* de testes unitários) só que usados para consistir parâmetros. Fazendo uso do `static import` é ainda mais simples usar este tipo de métodos. Eis um exemplo de uma classe deste tipo:

```

01
02
03 public class Consistencies {
04
05 // lança IllegalArgumentException se obj for null
06 public static void consistNotNull ( Object obj ){ ... }
07
08 // lança IllegalArgumentException(mensagem) se obj for null
09 public static void consistNotNull ( Object obj, String message ){ ... }
10
11 // lança um objeto da classe de exceção passada como parametro
12 public static <E extends Exception > void consistNotNull ( Object obj, Class<E>
exceptionType ) throws E { ... }
13
14 // lança um objeto da classe de exceção passada como parametro com o texto passado message
15 public static <E extends Exception > void consistNotNull ( Object obj, String
message, Class<E> exceptionType ) throws E { ... }
16
17 // lança IllegalArgumentException se obj for vazio. Aplicável a CharSequence, Collection, Map
e arrays, por exemplo
18 public static void consistNotEmpty ( Object object ){ ... }
19
20 // lança IllegalArgumentException se value for falso
21 public static void consistTrue ( boolean value, String message ){ ... }

```

```

22
23 // lança IllegalArgumentException se value for verdadeiro
24 public static void consistFalse ( boolean value, String message ){ ... }
25
26 // lança IllegalArgumentException se o valor object não está no intervalo [min,max]
27 public static <T extends Comparable<T>> void consistInBetween ( T min, T max, T object );

```

---

#### Código 4:

Este é apenas um esboço dos métodos possíveis. Mais métodos e mais sobrecargas poderiam ser adicionadas. Com esta classe a consistência de um objeto passa de

```

01
02
03 public void doSomething ( String a, String b , int c ){
04 if ( a== null ){
05 throw new IllegalArgumentException ( "argument a is null" );
06 }
07 if ( b== null ){
08 throw new IllegalArgumentException ( "argument b is null" );
09 }
10 if ( b.isEmpty () ){
11 throw new IllegalArgumentException ( "argument b is empty" );
12 }
13 if ( ! ( c >= 0 && c <= 100 ) ){
14 throw new IllegalArgumentException ( "argument c is out of range" );
15 }
16
17 // real code
18 }

```

---

#### Código 5:

para algo mais simples como:

```

01
02
03 public void doSomething ( String a, String b , int c ){
04
05 consistNotNull ( a, "argument a is null" );
06 consistNotNull ( b, "argument b is null" );
07 consistNotEmpty ( b, "argument b is empty" );
08 consistInBetween ( 0 , 100 , c, "argument b is out of range" );
09
10 // real code
11 }

```

---

#### Código 6:

As formas sobrecarregadas permitem que especifique a exceção e/ou a mensagem a se usada o que segue a boa prática de “*Seja Específico*”.

## Resumo

Estas classes e padrões que facilitam o tratamento de exceções de forma orientada a objetos de forma a prover oportunidade de estender as funcionalidades básicas descritas aqui. Classes especiais como **DAO** ou **Repositorios** podem-se beneficiar do padrão `ExceptionHandler` para centralizar o tratamento de exceções entre métodos, tornando o código mais curto, simples e portanto legível.

Espero que a série de artigos sobre exceções lhe permitam entender melhor o conceito de Exceção e usufruir de todo o poder que o Java coloca na sua mão. Sem se perder, e sem más práticas.

## Referências

### [1] Exceções : Conceitos

Sérgio Taborda

Editor:

URL: <https://sergiotaborda.wordpress.com/2009/04/06/excecoes-conceitos/>

### [2] Exceções : Boas práticas, más práticas

Sérgio Taborda

Editor:

URL: <https://sergiotaborda.wordpress.com/2009/04/06/excecoes-boas-praticas-mas-praticas/>

+ Seguir

Follow “Sérgio Taborda's Weblog”

Get every new post delivered to your Inbox.

Introduza o seu endereço de e-mail

Sign me up

## Licença



Sérgio Taborda

Build a website with WordPress.com

Este trabalho é licenciado sob a

[Licença Creative Commons Atribuição-Uso Não-Comercial-Não a obras derivadas 3.0 Genérica](https://creativecommons.org/licenses/by-nc-nd/3.0/).

★ Gosto

Be the first to like this.

[Comentários \(5\)](#) [Trackbacks \(0\)](#) [Deixe o seu comentário](#) [Trackback](#)

1.



Emerson

2009/04/06 às 17:57

[Responder](#)

Olá, como vai?

Primeiramente, muito bom seus artigos sobre exceções. Está ajudando muito.

Fiquei com uma dúvida. Como eu poderia implementar o seguinte método que você citou?

```
public static void consistNotNull ( Object obj, String message,Class exceptionType ) throws E {
// How to do }
```

Desde já agradeço.

2.



sergiotaborda

2009/04/07 às 13:07

[Responder](#)

Seria mais ou menos assim:

```
public static void consistNotNull(Object obj, String message, Class exceptionType)
throws E{
if (obj == null){
E exception = ReflectionUtils.newInstance(exceptionType, message); // cria exceção com
um parametro de mensagem
throw exception;
}
}
```

O segredo está dentro do newInstance que usa reflection para obter o construtor com um parametro de string da exceção E e depois é só lançar essa exceção.



3.

Tomaz.Lavieri

2009/05/18 às 14:13

[Responder](#)

Sergio, se possivel teria como vc exemplificar uma implementacao real do padrao handle??

muitas tenho visto algo parecido mas sempre fico com duvidas quanto a ele...

e tenho duvidas tb de como separa as excecoes da camada de persistencia, principalmente no entorno do repositório

Estou escrevendo do meu celular e por isso o texto pode sair meio tronxo e sem acento...



o

sergiotaborda

2009/05/19 às 8:17

[Responder](#)

O padrão exception handler é apenas um método que lida com exceções. Pode-se colocar em uma classe se as exceções são muito usadas como IOException ou pode ser apenas um método dentro da sua classe. Uma implementação seria por exemplo:

```
public RuntimeException handle(IOException e){
try{
throw e;
}catch (FileNotFoundException e){
String fileName = ... // parse da mensagem de e
throw new MyFileNotFoundException(fileName);
} catch (ConnectionTimeoutException e){
throw new MyConnectionTimeoutException(e);
} catch (IOException e){
throw new MyIOException(e);
}
}
```

Aqui usei o try catch como se fosse um switch mas pode usar if e instanceof. A ideia é analisar a exceção que aconteceu, retirar dela o máximo de informações e criar a sua própria exceção com essas informações de forma que poder obtê-las depois. Por exemplo, MyFileNotFoundException teria um método getFile com o nome do arquivo que não foi encontrado.

Para isolar a sua camada em todos os métodos públicos da interface pública (ou seja, os métodos chamados pela camada acima) vc deve colocar um try-catch e capturar todas as exceções. Encapsulá-las em uma exceção da camada e pronto. Para repositório vc pode criar um RepositoryException. Como tem que fazer isso em todos métodos o tratamento da exceção pode ficar num método privado à parte (o handler) que vc chama em todos os métodos. Lembre-se que RepositoryException deve ter vários filhos que especifiquem da melhor forma possível o que aconteceu. Nisso o handler tb pode dar uma grande ajuda. Este handlers privados são mais específicos e portanto podem fazer tratamentos melhores.

A única exceção a esta regra é na última camada do seu sistema, ou seja, quando vc perde o controle sobre o fluxo de dados. Basicamente quando chega na camada que interage com o usuário ou com um sistema externo. Aqui vc deve fazer o handler logar a exceção e traduzi-la para uma exceção da api superior. Por exemplo, dentro de um servlet vc traduz para ServletException ou IOException. Em swing vc deve mostrar um messagebox avisando o usuário, mas apenas se a exceção foi inesperada.

Não há muito mais a dizer do que isso...

4.

[Ricardo](#)

2011/08/26 às 9:04

[Responder](#)

Bom Tutorial =D.

1. No trackbacks yet.

## Deixar uma resposta

Escreva o seu comentário aqui...

[RSS feed](#)

## Artigos recentes

- [O movimento perpétuo e a energia eterna](#)
- [O fuso e a roca](#)
- [Voto Consciente](#)

## Blog no Java Buinding

Com a inauguração do [JavaBuilding](#) as minhas obverações sobre desenvolvimento de software em geral e sobre Java e Scrum em particular podem agora ser seguidas no meu novo blog [Caderno Sérgio Taborda no JavaBuinding](#). Este blog permance apenas para assuntos não relacionados a desenvolvimento de software.

### [Caderno no Javabuilding](#)

- [O Paradoxo do Inventor](#)
- [Coleções turbinadas](#)
- [Streams no Java 8 e em outras Linguagens](#)
- [Variância](#)
- [Java 8 – Prólogo](#)
- [Monads em Java](#)
- [Scala: O vencedor da batalha Java vs .Net](#)

### [MiddleHeaven](#)

- [O caso de Enumerable infinito](#)
- [MiddleHeaven e Java 8](#)
- [Javadoc Disponível](#)
- [Lista de Discussão](#)
- [Seis anos e muito para fazer](#)
- [Seguindo em frente](#)
- [No céu do meio](#)
- [Novo Conteudo](#)
- [Utilitários: Coleções aumentadas](#)
- [Nosso novo blog](#)

## [Twitter](#)

- O Paradoxo do Inventor - Como pensar grande dá mais resultado [ow.ly/NiDAH 1 month ago](#)
- Entenda mais sobre como a nova API de Stream vai mudar sua forma de programar e como ela afetou o design do java 8 [ow.ly/LGYPG 2 months ago](#)
- A variancia em java e outras linguagens [ow.ly/Li320 2 months ago](#)
- Monads em Java [ow.ly/qs5Qo 1 year ago](#)
- O vencedor da batalha Java vs .Net [ow.ly/qcSCr 1 year ago](#)

## Meta

- [Registar](#)
- [Iniciar sessão](#)
- [RSS dos artigos](#)
- [Feed RSS dos comentários.](#)
- [Blog em WordPress.com.](#)

## Páginas



- [Desenvolvimento de Software](#)
  - [A Arte de Fabricar Software](#)
  - [Arquitetura](#)
    - [Arquitetura Orientada ao Domínio](#)
    - [Arquitetura Web](#)
  - [Java](#)
    - [Coleções: Como não usar Arrays](#)
    - [Do DAO ao Domain Store](#)
    - [Exceções: Boas Práticas, Más Práticas](#)
    - [Exceções: Classes Utilitárias](#)
    - [Exceções: Conceitos](#)
    - [FAQ](#)
      - [Primeiro Programa](#)
      - [Sorteio aleatório sem Repetição](#)
      - [Trabalhando com Números](#)
    - [Igualdade em Java](#)
    - [Introspeção](#)
    - [java.lang.Object](#)
    - [OO](#)
      - [Herança](#)
      - [Polimorfismo](#)
      - [Separação de Responsabilidades e Encapsulamento](#)
    - [Os 10 mandamentos do bom programador Java](#)
    - [Palavras Reservadas](#)
    - [Patterns](#)
      - [Adapter](#)
      - [Bean](#)
      - [Builder](#)
      - [Composite](#)
      - [DAO](#)
      - [Factory](#)
      - [Factory Method](#)
      - [Fastlane](#)
      - [Iterator](#)
      - [Money](#)
      - [MoneyBag](#)
      - [MVC](#)
      - [Query Object](#)
      - [Repository](#)
      - [Singleton](#)
      - [Transfer Object](#)
      - [Value Object](#)
  - [Scrum](#)
    - [Equipe](#)
    - [Planejamento](#)
    - [Produto e Projeto](#)
    - [Projeções](#)
    - [Sprint](#)
    - [Valores](#)
- [Física](#)
  - [Mecânica Quântica](#)
- [Livros](#)
- [Magic: The Gathering](#)
  - [O Segredo do Magic](#)

- [Sobre mim](#)

[Topo](#)

[Blog em WordPress.com.](#) [O tema INove.](#)

