Java: tratamento de exceções

DEVinHouse

Parcerias para desenvolver a sua carreira

SENAI

<LAB365>

AGENDA

- Terminologia
- Exceções checadas e não checadas
- Como tratar exceções
- Como especificar exceções
- Como saber se devemos especificar ou tratar com uma exceção

Terminologia

Call Stack: é a lista de métodos que foram chamados de maneira ordenada e que resultaram em erro.

Exception Class: identifica o tipo de erro ocorrido. Ex: NumberFormatException, ocorrido quando uma string está com formato errado para ser convertida em número.

Hierarchy: assim como toda classe, a classe de exceção é parte de uma hierarquia de herança. Originada em java.lang.Exception ou alguma de suas subclasses. Sua função é agrupar tipos similares de erros. Ex: illegalArgumentException - indica que o argumento de um método é invalido, superclasse do NumberFormatException.

Podemos estender as classes de exceção criando classes de exceção customizadas.

```
public class MyBusinessException extends Exception {

	private static final long serialVersionUID =
7718828512143293558L;

	public MyBusinessException() {
		super();
	}

	public MyBusinessException(String message,
Throwable cause, boolean enableSuppression, boolean
writableStackTrace) {
		super(message, cause, enableSuppression,
writableStackTrace);
	}

	public MyBusinessException(String message,
Throwable cause) {
		super(message, cause);
	}

	public MyBusinessException(String message) {
		super(message);
	}

	public MyBusinessException(Throwable cause) {
		super(cause);
	}
}
```

Terminologia

Exception Object: é uma instância de uma classe de exceção. Quando o fluxo do programa é interrompido por um evento durante o runtime, um objeto de exceção é criado. o que é chamado de “jogar” uma exeção. “to throw an exception”. Pois usamos a palavra-chave “throw” para apontar a exceção durante o runtime.

Exceções checadas e não checadas

Uma **exceção checada** estende a classe *Exception* e deve ser usada para todos os eventos excepcionais que pudermos antecipar. Métodos que jogam esse tipo de exceção precisam especificar ou tratar com a exceção.

Já uma **exceção não checada** estende a classe *RuntimeException*. Deve ser utilizada para erros não antecipáveis e que colocam a aplicação em dificuldade de recuperação. Neste caso, seus métodos não precisam antecipar nem tratar com as exceções. Ex: utilização imprópria de uma API que provoca um *IllegalArgumentException;*
ou ainda a inicialização de uma variável faltante que resultaria em uma *NullPointerException*

Como tratar Exceções

Duas abordagens: *Try-catch-finally* e *Try-with-resource.*

Try-catch-finally, abordagem mais clássica que pode ter 3 passos:

1. utilização de um bloco de *try* que circunscreve o bloco de código em que poderá haver exceção;
2. um ou mais “catch blocks” para tratar com a exceção;
3. um bloco “definitivo” a ser executado após o bloco de tentativa ser executado com sucesso ou de uma exceção ter sido tratada.

   De todo modo o “try block” é necessário, mas o “catch” e o “finally” (definitivo) são opcionais.

Como tratar Exceções

Try Block

Parte do código que circunscreve o código que pode dar erro.
Podemos usar um bloco para cada possível statement que pode dar erro ou então um único bloco para múltiplos statements que podem dar erro.
Exemplo:

```
public void performBusinessOperation() {
	try {
		doSomething("A message");
		doSomethingElse();
		doEvenMore();
	}
	// see following examples for catch and
finally blocks
}

public void doSomething(String input) throws
MyBusinessException {
	// do something useful ...
	throw new MyBusinessException("A message that
describes the error.");
}

public void doSomethingElse() {
	// do something else ...
}

public void doEvenMore() throws NumberFormatException{
	// do even more ...
 }
```

Como tratar Exceções

Catch Block

É possível implementar tratamentos para uma ou mais exceções dentro de um “catch block”. o Catch recebe a exceção como parâmetro. Exemplos:

```
public void performBusinessOperation() {
        try {
                doSomething("A message");
                doSomethingElse();
                doEvenMore();
        } catch (MyBusinessException e) {
                e.printStackTrace();
        } catch (NumberFormatException e) {
                e.printStackTrace();
        }
}
```

```
public void performBusinessOperation() {
        try {
                doSomething("A message");
                doSomethingElse();
                doEvenMore();
        } catch
(MyBusinessException|NumberFormatException
e) {
                e.printStackTrace();
        }
}
```

Como tratar Exceções

Catch Block

Os exemplos anteriores apenas mostram a classe a mensagem e a call stack da exceção a partir do método printStackTrace();

Em situações reais devemos buscar implementações mais robustas que apontem uma mensagem de erro ao usuário solicitando um input correto ou ainda registrar em um log a sua ocorrência.

Visto que em produção é importante o monitoramento da aplicação e de suas exceções e tratamentos.

Como tratar Exceções

Finally Block

Executado após a execução com sucesso de um try block ou ainda após um dos catch blocks tratarem uma exceção.

Local ideal para a implementação de lógicas de limpeza ou então fechar conexões, InputStream .. etc

No exemplo a seguir, o bloco finally é executado mesmo que a instância de *FileInputStream* produza um erro de *FileNotFoundException* ou ainda o processamento do arquivo produza qualquer outra exceção.

Os finally block são excelentes momentos para prevenir memory leaks, como vemos também no exemplo. Era a melhor prática antes do Java 7.

```
FileInputStream inputStream = null;
try {
        File file = new File("./tmp.txt");
        inputStream = new
FileInputStream(file);

        // use the inputStream to read a file

} catch (FileNotFoundException e) {
        e.printStackTrace();
} finally {
        if (inputStream != null) {
                try {
                        inputStream.close();
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

Como tratar Exceções

Finally Block

Executado após a execução com sucesso de um try block ou ainda após um dos catch blocks tratarem uma exceção.

Local ideal para a implementação de lógicas de limpeza ou então fechar conexões, InputStream .. etc

No exemplo a seguir, o bloco finally é executado mesmo que a instância de *FileInputStream* produza um erro de *FileNotFoundException* ou ainda o processamento do arquivo produza qualquer outra exceção.

Os finally block são excelentes momentos para prevenir memory leaks, como vemos também no exemplo. Era a melhor prática antes do Java 7.

```
FileInputStream inputStream = null;
try {
        File file = new File("./tmp.txt");
        inputStream = new
FileInputStream(file);

        // use the inputStream to read a file

} catch (FileNotFoundException e) {
        e.printStackTrace();
} finally {
        if (inputStream != null) {
                try {
                        inputStream.close();
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

Como tratar Exceções

Try-with-resource:

A partir do Java 7 é possível utilizar o statement *Try-with-resource* para garantir que os recursos sejam automaticamente fechados (precisam ter a interface AutoCloseable implementadas, mas é o caso da maioria dos objetos em Java)

Sendo apenas necessário instanciar o objeto dentro da cláusula try. Tratar e especificar exceções que podem surgir enquanto fecha-se o recurso.

```
File file = new File("./tmp.txt");
try (FileInputStream inputStream = new
FileInputStream(file);) {
        // use the inputStream to read a file
} catch (FileNotFoundException e) {
        e.printStackTrace();
} catch (IOException e) {
        e.printStackTrace();
}
```

como vemos no exemplo, o tratamento da IOException, que pode acontecer enquanto fecha-se o *FileInputStream* ocorre no catch block do try-with-resource e não precisa, então, de um try-catch statement aninhado. Como no caso do finally block do exemplo anterior.

Como especificar Exceções

Se não tratarmos uma exceção dentro de um método, isso irá afetar o restante da call stack. Se for uma exceção checada, precisamos especificar que o método precisa apontar a exceção.

Fazemos isso adicionando “throw clause” na declaração do método. Deste modo, todos os métodos chamados precisam tratar ou especificar exceções em si mesmos.

Exceções não checadas também precisam ser especificadas se quisermos que o método as sinalize.

```
public void doSomething(String input) throws
MyBusinessException {
        // do something useful ...
        // if it fails
        throw new MyBusinessException("A message that
describes the error.");
 }
```

Como especificar Exceções

Se devemos especificar ou tratar uma exceção depende do caso. E não há uma regra muito geral para isso.

Mas podemos nos basear em duas perguntas para nos ajudar:

1. É possível tratar a exceção dentro do método atual?
2. É possível prever as necessidades de todos os usuários da classe? Se sim, o tratamento da exceção vai atender todas essas necessidades?

Respondendo sim para todas as questões então devemos tratar a exceção dentro do método. De outro modo, é mais interessante especificar a exceção.

```
public void doSomething(String input) throws
MyBusinessException {
        // do something useful ...
        // if it fails
        throw new MyBusinessException("A message that
describes the error.");
 }
```

Material Complementar

- https://devopedia.org/java-modifiers

AVALIAÇÃO DOCENTE

O que você está achando das minhas aulas neste conteúdo?

Clique aqui ou escaneie o QRCode ao lado para avaliar minha aula.

Sinta-se à vontade para fornecer uma avaliação sempre que achar necessário.

DEVinHouse

Parcerias para desenvolver a sua carreira

OBRIGADO!

SENAI

<LAB365>