Sérgio Taborda's Weblog

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

Type text to search here...

Exceções: Conceitos

Deixe o seu comentário Go to comments

Conceito Inovador

Durante a codificação o programador se depara muitas vezes com a necessidade de fazer verificações antes de proceder ao real propósito do código. Por exemplo, verificar que o arquivo que quer ler, de fato existe ou que a conexão à internet realmente está aberta. Quando se verifica que a condição é falsa, então o programa não tem como continuar pois as condições essenciais ao seu funcionamento não estão satisfeitas. O conceito de Exceção foi introduzida pela linguagem C++ para tentar libertar o programador de continuamente ter que resolver o que fazer quando uma condição essencial não se verifica. O programador pode decidir o que fazer mais tarde no código. Este foi realmente um mecanismo inovador, que praticamente todas as linguagens adotaram desde então.

A linguagem Java introduziu pela primeira vez o conceito de exceção verificada (`Checked Exception`). A base para isto é que certas condições são tão importantes que o programador não deve se escusar de tratar o problema imediatamente. Normalmente este tipo de situação existe quando o programa tem que interagir com o ambiente em que executa, por exemplo com o sistema de arquivos ou a rede.

Exceções em Java

Exceção é um evento que acontece durante a execução de um prmpendo o curso normal do seu fluxo lógico. Em Java exceções são representadas por uma classe particular de objetos. A classe que representa uma exceção é `Throwable`

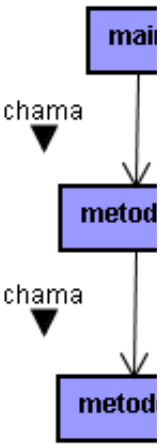

Ilustração 1: Invocação de método vs lançamento de excções

Todo o mecanismo da linguagem relativo a exceções é baseado no conceito de que exceções são lançadas e capturadas. Quando uma exceção é criada ela é lançada de dentro do método onde aconteceu. Se o método não capturar essa exceção ela será passada ao método que chamou o método onde ela foi lançada, Isso acontece assim, sucessivamente, até que a exceção seja apanhada ou ela chegue na JVM, caso em que será capturada automaticamente. Trabalhar com exceções é decidir onde capturar quais exceções e o que fazer uma vez que elas são apanhadas.

Tipos de Exceção

Existem três categorias de exceções: Erro, Falha e Exceção de Contingência representadas respectivamente pelas classes: `Error`, `RuntimeException` e`Exception.` Todas estas classes são filhas de `Trowable`.

A hierarquia de exceções em Java não tem como objetivo criar implementações ligeiramente diferentes da mesma coisa. Cada tipo de exceção tem um interpretação especial que se reflete na forma como o programador tem que lidar com elas.

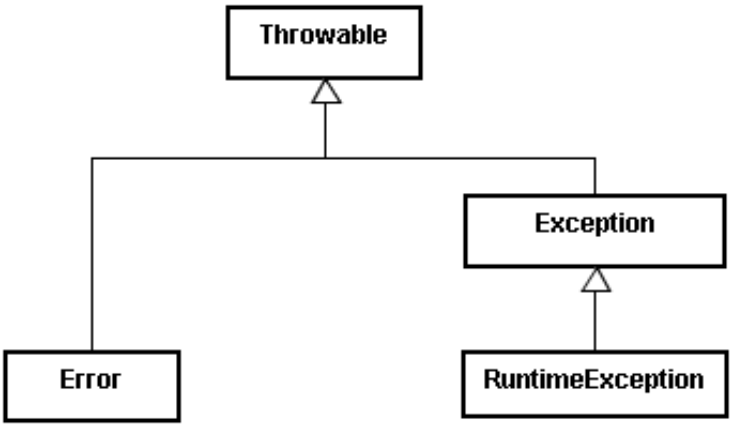

Ilustração 2: Hierarquia de tipos exceções em Java

Erros

Erros são exceções tão graves que a aplicação não tem o que fazer. São erros todas as classes que descendem diretamente de `Error`.

É importante que os erros sejam reportados e que se saiba que aconteceram, mas o programa não tem o que fazer para resolver o problema que eles apontam. Erros indicam que alguma coisa está realmente muito errada na construção do código ou no ambiente de execução. Exemplos de erros são

`OutOfMemoryError` que é lançada quando o programa precisa de mais memória mas ela não está disponível, e StackOverflowError que acontece quando a pilha estoura, por exemplo, quando um método se chama a si mesmo sem nunca retornar.

```
public int stackOverFlow(int a){ return
stackOverFlow(a);}
```

Código 1: Exemplo de um método que causa `StackOverflowError`.

Falhas

Falhas são exceções que a aplicação causa e pode tratar. Digo pode tratar porque não é obrigada a fazê-lo. São falhas todas as classes que descendem diretamente de `RuntimeException`.

Se a aplicação nunca apanhar este tipo de exceção, tudo bem, a JVM irá capturá-la. Mas provavelmente a sua aplicação não mais funcionará corretamente. Exemplos de falhas são `IllegalArgumentException` e `NullPointerException`. A primeira acontece quando se passa um parâmetro para um método e o método não o pode usar. A segunda acontece sempre que tentar invocar um método em uma variável de objeto não inicializada. Isso é bastante comum e por isso ela é, provavelmente, a exceção mais reportada de todas. Exceções deste tipo existem em outras linguagem que têm o conceito de exceção e são as que nos devem preocupar enquanto programamos porque traduzem situações que desafiam a lógica do programa.

Exceções de Contingência

Exceções de Contingência são aquelas que a aplicação pode causar ou não, mas que tem que tratar explicitamente. Exceções de Contingência são todas aquelas que descendem diretamente de `Exception` excepto as que descendem de `RuntimeException.`

Devido ao nome sugestivo é comum confundir o conceito de exceção com a própria classe `Exception`. Tenha sempre em mente que ao falarmos de exceções e tratamentos de exceções nos estamos referindo a qualquer um dos conceitos acima, ao mecanismo em si e não apenas às classes nem a uma classes em particular.

As exceções de contingência se chamam assim porque freqüentemente representam exceções para as quais o programa deve ter um plano de contingência. O exemplo clássico é a exceção `FileNotFoundException` que significa que o arquivo que estamos tentando ler, não existe. Isto é uma exceção no sentido que o programa espera que o arquivo exista, contudo, se ele não existir o programa deve ter um plano B, que pode ir de simplesmente não fazer nada até apresentar uma mensagem ao usuário final ou invocar outro método que irá buscar o arquivo em outro lugar; na rede ou via FTP, por exemplo.

Ambientes que não se controlam não são confiáveis. Não se pode assumir que esses ambientes estão funcionado corretamente, ou sequer que estão funcionando. Como acessar a outros ambientes é o que uma aplicação normalmente faz é necessário que as exceções que daí decorrem sejam exceções de contingência.

Exceções verificadas e não-verificadas

Java foi a primeira linguagem a introduzir o conceito de exceção verificada. Por padrão as exceções em Java são verificadas.

A aplicação é obrigada a tratar estas exceções explicitamente na medida que o método que recebe este tipo de exceção é forçado a verificar se pode resolver o problema. Se o método sabe que não poderá resolver a exceção, ele deve declarar isso explicitamente.

Note que este mecanismo de verificação é muito útil quando temos que antever planos de contingência para a ocorrência da exceção. O uso da palavra contingência não é coincidência. Java entende que para todas as exceções devem existir planos de contingência, alternativas que permitam que o programa continue funcionando normalmente.Assim todas as classes que descendem diretamente de `Throwable` ou `Exception` são exceções verificadas.

Mas o mundo não é perfeito e Java entende também que existem exceções para as quais não é possível antever um plano de contingência, erros e falhas, são esses tipos. Os primeiros nunca deveriam acontecer, portanto não faz sentido ter planos para os resolver. Os segundos podem até demonstrar que a aplicação está funcionado corretamente ao identificar aquele problema. Por isso `Error` e `RuntimeException` são exceções não-verificadas.

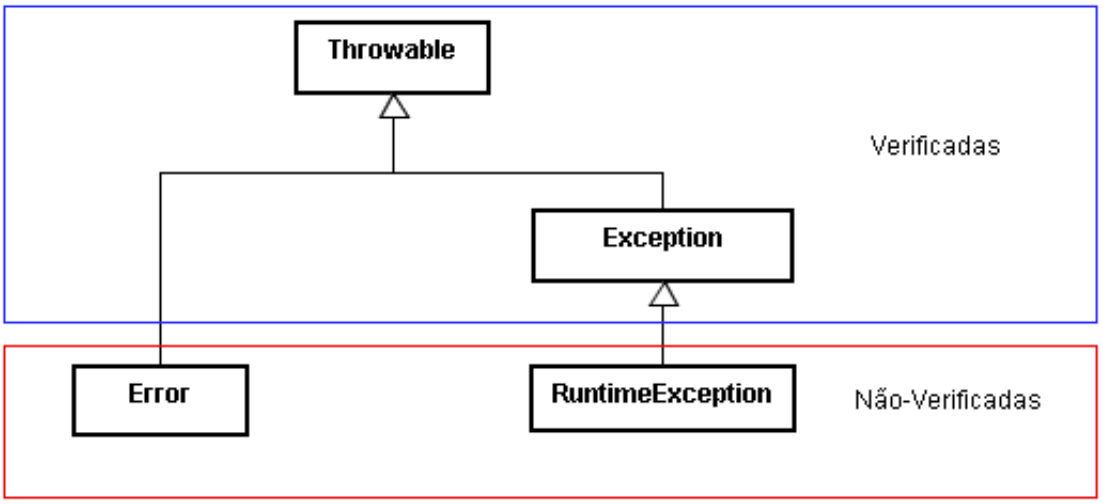

Ilustração 3: Exceções Verificadas e não-Verificadas

O fato de `RuntimeException` herdar de `Exception` confunde muita gente porque parece significar que todas as `RuntimeException` são também `Exception`. Isso significaria que todas as falhas poderiam ser resolvidas se existirem planos de contingência para elas. O que é verdade todas as falhas são possíveis exceções de contingência . Na verdade é por isso que todas as `RuntimeException` são também `Exception`. Elas podem ser resolvidas se o programa tiver como e desse ponto de vista a sua resolução é igual à de `Exception`.

Mas `RuntimeException` herdar de `Exception` parece significar que o comportamento de exceção verificada é também herdado, o que não é verdade. O mesmo argumento poderia ser usado com `Error` e `Throwable`. A verificação é algo que o compilador obriga, como obriga por exemplo que não exista nenhum código depois de um `return` o que seja feito um cast se os dois lados de uma atribuição não são da mesma classe. A verificação é portanto uma característica da linguagem que nada tem a ver com herança. Se este conceito é difícil de entender, pense apenas que as várias classes de exceção são marcadores para o compilador e a JVM saberem como e quando lançar/capturar a exceção que elas representam. Exceções verificadas são uma característica da inteligência da linguagem Java.

Muitos poderão ler a última frase com um sorriso. Hoje parece irônico dizer que exceções verificadas são algo inteligente. Porque as exceções verificadas sempre têm que ser verificadas por todos os métodos por onde passam, tornam-se rapidamente um problema para o programador, que acaba

incorrendo em muitos erros ao tratar todas as exceções verificadas que lhe aparecem. Veremos na segunda parte deste artigo como tratar corretamente as exceções verificadas. Espero que por agora, aceite a afirmação de que é realmente uma característica inteligente.

Lançando Exceções

Como foi dito, o mecanismos de exceções se baseia em lançar e capturar. Veremos agora como lançar e capturar exceções.

Throw e Throws

Para lançar uma exceção simplesmente usamos a clausula `throw` seguida do objeto que representa a exceção que queremos lançar. O conceito é semelhante ao de `return`, mas enquanto `return` está devolvendo um resultado de dentro do método, `throw` está lançando uma exceção. Nunca é possível considerar uma exceção como o resultado de um método, o objetivo do método é obter resultados sem lançar exceções.

```
public double divide (double dividendo , double divisor){ if (divisor==0){ // não podemos dividr por zero.}}
```

Código 2: Lançando uma exceção.

Se o divisor é zero, isso não é correto.

Aqui temos um método que tentará dividir dois double . O Java vai deixar dividir esses números mesmo que o divisor seja zero, mas como regra do nosso programa não queremos deixar isso acontecer. Então, testamos se o divisor é zero e se for, lançamos uma exceção. `ArithmeticException` é uma exceção não-verificada padrão que representa que há um problema na hora de fazer algum calculo.

Lançar exceções é o inicio do processo, mas quando a exceção for lançada o método que chamou este terá que saber que este método pode lançar este tipo de exceção. Para informar isso usamos a palavra reservada `throws` na assinatura do método, indicando a quem usar o método que ele pode lançar uma `ArithmeticException.`

```
public double divide (double dividendo , double divisor)
```

Código 3: Usando throws

Lembre-se que exceções só podem ser lançadas de dentro de métodos e que uma vez lançadas elas passam por toda a cadeia de métodos que estão na pilha de chamadas. Ou seja, passam pelo método que chamou o método que lançou a exceção. Pelo método que chamou esse método, e pelo método que chamou este outro e assim sucessivamente até que sejam apanhadas. Se o programa não apanhar a exceção a JVM o fará. Por padrão, a JVM exibirá uma mensagem no console.

Documentando o lançamento

Quando a exceção é não-verificada, não é obrigatório indicar o seu lançamento na clausula `throws` mas sempre é necessário documentar a razão que irá lançar essa exceção. A forma mais simples de documentar é usar o Javadoc e a tag `@throws`. O código final seria então:

```
/** Devide dois numeros double. O divisor não pode ser zero. @param dividendo o numero a dividir @param divisor o numero pelo qual dividir. Não pode ser zero. @return o quociente da divisão.

@throw ArithmeticException se o divisor for zero.

*/

public double divide (double dividendo , double divisor){

if (divisor==0){

// não podemos dividr por zero.

throw new ArithmeticException(“Divisor não pode ser zero”);

}

return dividendo / divisor;

}
```

Código 4: Código com exceção e javadoc

É sempre importante documentar as exceções , verificadas ou não, que o método pode lançar e as condições em que elas serão lançadas. Isso dá informação a quem usar o método do tipo de condições em que o método pode ser usado.

Capturando Exceções

Exceções são lançadas de dentro de métodos. Para entender como capturar a exceção temos que

entender com usar o bloco `Try-Catch-Finally`. Na realidade estamos falando de 3 blocos diferentes `try-cacth`, `try-finally` e `try-catch-finally`.

Try-Catch

Esta é a forma mais usada. Todas as chamadas a métodos que sabemos que podem lançar exceções colocamos dentro de chaves com a palavra reservada `try` antes. Isso significa o seguinte: "JVM, tenta executar o seguinte código. Se uma exceção acontecer em qualquer método deixa-me tratá-la."

Uma vez capturada temos que dar tratamento à exceção, ou seja, fazer alguma coisa para resolver o problema que ela representa, ou tomar ações alternativas. O código para fazer isso colocamos dentro de chaves com a palavra reservada `catch` atrás.

Muitos tipos de exceções podem ser lançadas e nem sempre o mesmo código de tratamento serve para todos os tipos. Para informar qual o tipo de exceção que o código se destina a resolver colocamos uma declaração a seguir ao `catch` que indica o tipo de exceção que podemos tratar naquele código. O tipo é definido declarando uma classe especifica. Essa classe pode ser qualquer uma que herde implícita ou explicitamente de `Throwable`.

```
try { // aqui executamos um, ou mais, métodos// que podem lançar execções.}catch (Throwable e) { // aqui a execção aconteceu e tentamos evitar o problema

}
```

Código 5: Exemplo de uso do bloco try-catch

Podemos declarar mais do que um bloco `catch`. Isso é importante porque podemos ter vários tipos diferentes de exceção sendo lançados e necessitar de um tratamentos especifico para cada um. Por outro lado, não é sensato usar `Throwable` para definir a classe a ser capturada. Isso significa que queremos capturar todos os tipos de exceção. Como vimos, não há muito o que fazer quando acontece um erro. Por isso não estamos normalmente interessados em capturar exceções do tipo de `Error`. Eis um exemplo mais realista do uso de `try-catch`.

```
try { // aqui executamos um método que tenta ler um arquivo }catch (FileNotFoundException e) { // se o arquivo não existir esta exceção é lançada. // aqui colocamos a resolução

}catch (EOFException e) {
```

```
// quando esta exceção aoontece significa que aconteceu

// um problema na leitura do arquivo.

// aqui colocamos a resolução

} catch (IOException e) {

// uma outra exceção de I/O aconteceu.

// aqui colocamos a resolução

}
```

Código 6: Uso mais realista de try-cacth

Pode acontece que durante a tentativa de resolução do problema, cheguemos à conclusão que não podemos fazer mais nada e o problema é irresoluvel. Nesse caso, é possível usar `throw,` dentro do bloco `catch` para relançar a exceção que capturámos. Tudo se passa como se ela nunca tivesse sido apanhada.

A ordem pela qual devemos colocar os blocos `catch` não é aleatória. Se usarmos classes de exceção de uma mesma hierarquia, a classe mais genérica tem que ser capturada depois das outras da sua descendência. No exemplo anterior, `FileNotFoundException` e `EOFException` são filhas de `IOException` por isso ela é capturada depois das outras duas. Isto é assim porque a JVM irá comparar a classe da exceção que aconteceu com a classe declarada em `catch` e usar o primeiro bloco que for compatível. Se a classe mais genérica estiver antes, ela seria sempre a escolhida nunca dando chance de usar os outros blocos. Se você tentar fazer isso, o compilador irá reclamar, protegendo a lógica do mecanismo de tratamento.

Try-Finally

Por vezes, mesmo sabendo que os métodos que estamos usando lançam exceções, sabemos também que não podemos fazer nada para as resolver. Nesse caso, simplesmente não usamos o bloco try-`catch` e simplesmente declaramos as exceções com `throws` na assinatura do método. Mas, e se, mesmo acontecendo uma exceção existe um código que precisamos executar ? É neste caso que usamos o bloco `finally`.

Este tipo de problema é mais comum do que possa parecer. Por exemplo, se você está escrevendo num arquivo e acontece um erro, o arquivo tem que ser fechado mesmo assim. Ou se você está usando uma conexão a banco de dados e acontece algum problema a conexão tem que ser fechada.

Para usar o bloco `try-finally`, começamos como envolver os métodos que podem lançar exceções como vimos antes, mas usamos um bloco `finally` em vez de um `catch`.

```
try { // aqui executamos um método que pode lançar uma exceção que não // sabemos resolver}
finally { // aqui executamos código que tem que ser executado, mesmo que um problema aconteça.

}
```

Código 7: Uso de bloco finally

Isto é muito útil, mas pense o que acontece se dentro do bloco `try` colocamos um `return`.

Isso significa que algo tem que ser retornado para fora do método, mas significa também que o método acaba aí. Nenhum código pode ser executado depois de um `return` (o compilador vai-se queixar dizendo que o código seguinte é inalcançável). Isso é tudo verdade, exceto se esse código suplementar estiver dentro de um bloco `finally`. O código dentro do bloco `finally` não apenas é executado se uma exceção acontecer, mas também se o método for interrompido. É garantido que o código dentro do bloco `finally` sempre será executado, aconteça o que acontecer. Este é um outro uso importante deste bloco.

Try-Catch-Finally

Este bloco é apenas a conjunção dos anteriores. Apenas é necessário deixar claro que o bloco `finally` tem que ser declarado depois de todos os blocos `catch`. A Listagem seguinte mostra o uso de todos os conceitos e palavras chave relacionados ao mecanismo de exceções.

`SQLException` é uma exceção de contingência, e portanto verificada, mas nem sempre é claro como tratar esse tipo de exceção. Isso acontece porque na realidade essa exceção representa uma imensidão de exceções diferentes. A especificação JDBC 4.0 vem melhorar este cenário definindo classes filhas mais especificas.

```
// faz uma consulta SQL ao banco retornando todos os produtos public List<Produto>
queryAllProducts () throws SQLException { // Para podermos usar o objecto con dentro do try e
do finally // precisamos declará-lo for de ambos os blocos. Connection con = null;

try {

// obtém conexão. Não nos importa muito como.

con = this.getConnection();

// executa comando SQL

ResultSet rs = con.createStament().executeQuery(“ SELECT * FROM PRODUTOS”);

// mapeia o ResultSet para uma lista de objetos
```

```
List<Produto> resultado = mapResultSet(rs,Produto.class);
// retorna o rsultado.
// O código no bloco finally ainda será executado.
return resultado;
} catch (SQLException e) {
// descobre se a falha se deve à tabela não existir no banco
if (this.exceptionMeansTableMissing(e)){
// realmente a tabela não exite no banco.
// retorna uma lista vazia.
return Collection.emptyList();
} else {
// não conseguimos resolver o problema.
// relançamos a exceção
throw e;
}
} finally {
con.close();
}
}
```

Código 8: Exemplo completo do uso de try-catch-finally

Resumo

O mecanismo de exceções é baseado no lançamento e captura de objetos da classe `Throwable`. Este mecanismo é diferente do mecanismo de retorno de resultados invocado quando usamos `return` e por isso existe a palavra `throw`, que lança a exceção e dá inicio ao mecanismo de lançamento e captura de exceções.

Exceções podem ser verificadas ou não. As exceções verificadas obrigam o código a verificar se podem ser resolvidas ou evitadas. Exceções verificadas não são o padrão e não existem em outras linguagens. Exceções verificadas são normalmente usadas em código que acesse recursos fora da

memória da máquina, como o sistema de arquivos ou a rede. Java parte do principio de que ambientes que ele não controla não são confiáveis e que deve sempre haver pelo menos um plano de contingência no caso de algum problema acontecer.

Podemos capturar e tratar exceções usando uma conjunção do blocos `try-catch-finally`. É garantido que o código no bloco `finally` sempre é executado, mesmo que exista uma exceção e mesmo que o bloco `try` contenha a instrução `return`. Esta funcionalidade especial do bloco `finally` é importante quando temos que fazer operações de limpeza, como fechar conexões, antes de sair do método.

Se queremos apanhar uma, ou mais exceções, que sabemos que podem ser originadas pelos métodos que estamos usando basta envolver esses métodos dentro de um bloco de execução com palavra `try`. Isto indica ao Java que os métodos dentro do bloco podem lançar exceções. Se isso realmente acontecer, então a exceção será passado para o bloco `catch`, por fim, o código dentro de `finally` será executado.

Outros artigos

O asssunto de execções em Java é vasto. Em outro artigo falo sobre as boas e más práticas de tratamento de execções.

Referências

[1] **The Java Tutorials – Lesson Exception**
Sun Microsystems, Inc.
URL:http://java.sun.com/docs/books/tutorial/essential/exceptions/

[2] **Does Java need Checked Exceptions?**
Kevlin Henney
URL:http://www.mindview.net/Etc/Discussions/CheckedExceptions

[3] **JDBC 4.0 Enhancements in Java SE 6**
Srini Penchikala
URL:http://www.onjava.com/pub/a/onjava/2006/08/02/jjdbc-4-enhancements-in-java-se-6.html

Comentários (4) Trackbacks (1) Deixe o seu comentário Trackback

1. 
Tarso
2008/04/28 às 11:20
Responder

Olá Sérgio.

Venho acompanhado seus artigos, são excelentes.

Tenho uma dúvida: como proceder no caso de precisarmos executar código dentro de um bloco finally que gere uma checked exception? Colocar um try-catch dentro dele é recomendável?

Abraços

2. 
sergiotaborda
2008/04/28 às 13:31
Responder

Obrigado pela atenção.

Nesse caso só temos duas hipoteses: colocar outro try/catch no finally ou colocar um try/catch em torno do primeiro. A segunda opção é ruim porque pode nos obriga a tratar execções que o primeiro try/catch lançou relançando-as. Isso viola o principio de não apanhar o que não pode tratar. contudo é uma melhor opção quando as execções lançadas no finally são diferentes das lançadas no try/catch normal.
A necessidade de try/catch dentro do finally é comum quando se fecha uma Connection. Tem que ser feito no finally mas connection.close() lança SQLException (tal como todo o resto dos metodos JDBC). Não ha opção senão declarar isso no throws do metodo ou capturar. Mas se por exemplo o resultSet foi lido corretamente e tudo o mais e existe um return o finally não pode inutilizar isso. Aqui ha um trade-off. Cada caso é um caso. Neste caso especifico normalmente eu apenas logo a exceção consumindo-a. Isto porque se o trabalho deu certo não quero inutilizá-lo por causa de problemas de conexão. Mas como disse, cada caso é um caso. Em particular pode ser necessário testar o codigo da exeção SQL para dicidir o que fazer. E isso é dependente do banco.

3. 
André Bandera
2009/10/27 às 15:50
Responder

Parabéns pelo artigo, muito bom.

4. 
Jéssica
2010/12/12 às 14:33
Responder

Ótimo o artigo… Valew, foi de grande ajuda =D

1. 2010/05/01 às 18:10
Artigo – Exceções: Conceitos « Diário Java

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 obervações 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 JavaBuiliding. 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 Disponivel
- 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.
- Create a free website or blog at WordPress.com.

Páginas