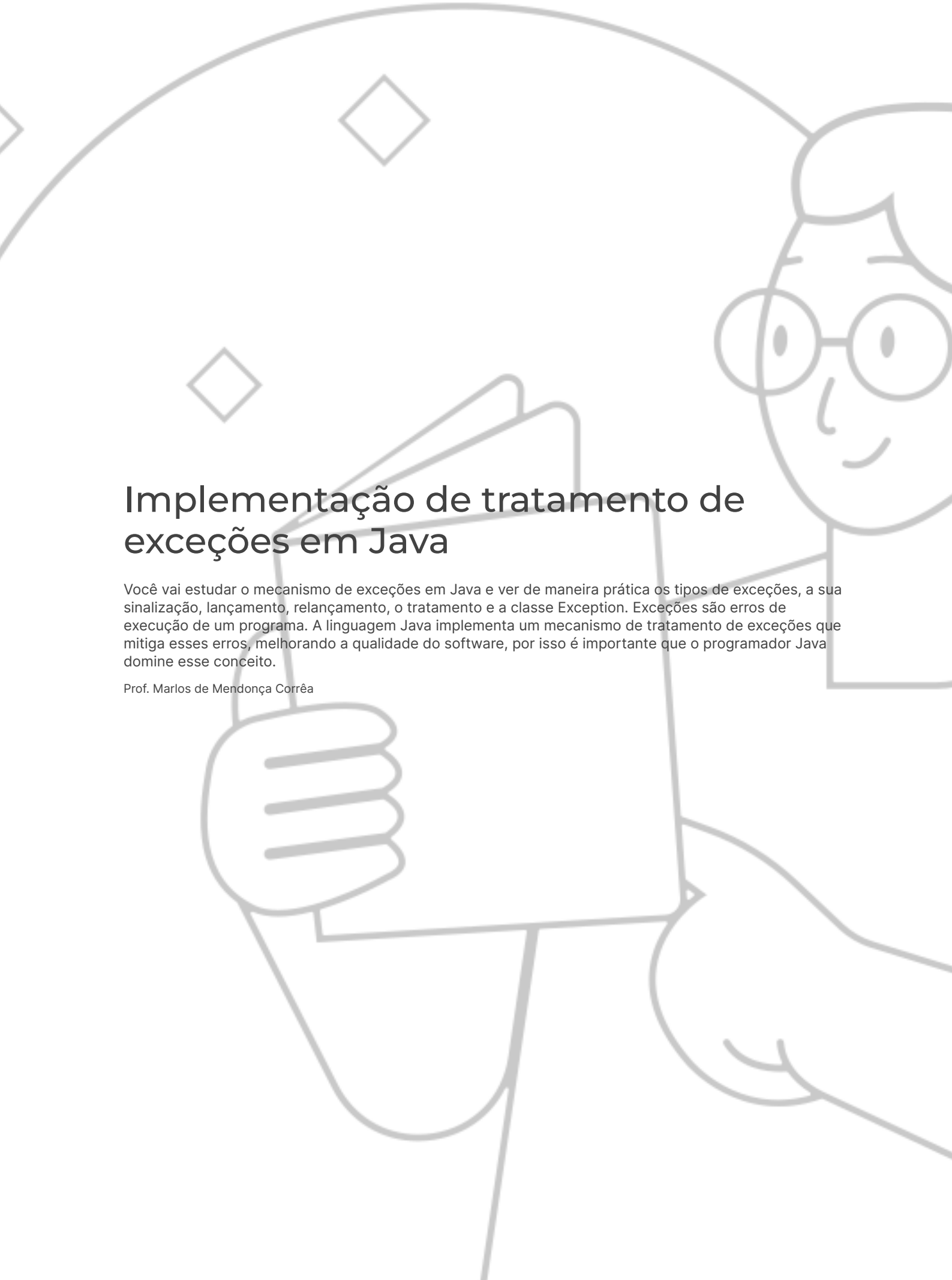


Implementação de tratamento de exceções em Java

Você vai estudar o mecanismo de exceções em Java e ver de maneira prática os tipos de exceções, a sua sinalização, lançamento, relançamento, o tratamento e a classe Exception. Exceções são erros de execução de um programa. A linguagem Java implementa um mecanismo de tratamento de exceções que mitiga esses erros, melhorando a qualidade do software, por isso é importante que o programador Java domine esse conceito.

Prof. Marlos de Mendonça Corrêa

Preparação

Você deve ter em seu computador um kit de desenvolvimento JAVA (JDK). Dois JDK populares e disponíveis para download gratuito são o Java SE Development Kit e o OpenJDK. Uma vez baixados e instalados, embora não seja imprescindível, instalar uma IDE facilita o desenvolvimento, pois é uma ferramenta útil no processo de aprendizagem. O Netbeans e o Eclipse são algumas das IDEs mais utilizadas e estão disponíveis para download gratuito.

Objetivos

- Identificar os tipos de exceções em Java.
- Identificar a classe Exception de Java.
- Analisar o mecanismo de tratamento de exceções da linguagem Java.

Introdução

A documentação oficial da linguagem Java explica que o termo exceção é uma abreviatura para a expressão “evento excepcional” e o define como um evento ocorrido durante a execução de um programa que interrompe o fluxo normal de suas instruções.

Essa definição de exceção não é específica da linguagem Java. Sempre que um evento anormal causa a interrupção no fluxo normal das instruções de um software, há uma exceção. Entretanto, nem todas as linguagens oferecem mecanismos para lidar com tais problemas. Outras disponibilizam mecanismos menos sofisticados, como a linguagem C++.

A linguagem Java foi concebida com o intuito de permitir o desenvolvimento de programas seguros. Assim, não é de se surpreender que disponibilize um recurso especificamente projetado para permitir o tratamento de exceções de software. Esse será o objeto de nosso estudo, a fim de que o futuro profissional de programação seja capaz de explorar os recursos da linguagem Java e produzir softwares de qualidade. Acompanhe o vídeo!

Introdução

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Introdução ao tratamento de exceção em Java

A linguagem Java surgiu com a intenção de facilitar o desenvolvimento de software. É essencial entender, contudo, que essa facilitação tem alcance muito mais amplo do que o conforto do programador. Operações indevidas no código de um programa, como uma divisão por zero, são um caminho para exploração de falhas de segurança.

Assista ao vídeo e entenda que a exceção é um erro em tempo de execução, que não pode ser previsto e evitado por código. Você verá que Java implementa mecanismos de tratamento de exceção e as principais estruturas sintáticas usadas para isso.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Uma exceção é uma condição causada por um erro em **tempo de execução** que interrompe o fluxo normal de execução. Esse tipo de erro pode ter muitas causas, como uma divisão por zero.

Acompanhe a seguir como funciona os mecanismos de tratamento de exceção implementados por Java.

- Quando uma exceção ocorre em Java, é criado um objeto, chamado de exception object, que contém informações sobre o erro, seu tipo e o estado do programa quando o erro ocorreu. Após ser criado, esse objeto é entregue para o sistema de execução da Máquina Virtual Java (MVJ), processo chamado de **lançamento de exceção**.
- Quando a exceção é lançada por um método, o sistema de execução vai procurar na pilha de chamadas (call stack) um método que contenha um código para tratar essa exceção. O bloco de código que tem por finalidade tratar uma exceção é chamado de exception handler (tratador de exceções).
- A busca seguirá até que o sistema de execução da MVJ encontre um exception handler adequado para tratar a exceção, passando-a para ele. Quando isso ocorre, é verificado se o tipo do objeto de exceção lançado é o mesmo que o tratador pode solucionar. Se for, ele é considerado apropriado para aquela exceção. Quando o tratador de exceção recebe uma exceção para tratar, diz-se que ele captura (catch) a exceção.
- Quando fornece um código capaz de lidar com a exceção ocorrida, o programador tem a possibilidade de evitar que o programa seja interrompido. Contudo, se nenhum tratador de exceção apropriado for localizado pelo sistema de execução da MVJ, o programa será terminado.

Um bloco monitorado para o lançamento de exceção tem a forma geral mostrada no **código 1**. Repare que múltiplos blocos catch podem ser encadeados.

```java
try {
//bloco de código monitorado
}
catch ( ExcecaoTipo1 Obj) {
//tratador de exceção para o tipo 1
}
catch ( ExcecaoTipo2 Obj) {
//tratador de exceção para o tipo 2
}
...//“n” blocos catch
finally {
// bloco de código a ser executado após o fim da execução do bloco “try”
}
```

Código 1: Forma geral de declaração de um bloco monitorado para exceção.

Embora o uso de exceções não seja a única forma de se lidar com erros em software, ela oferece algumas vantagens, como:

1. Separar o código destinado ao tratamento de erros do código funcional do software. Isso melhora a organização e contribui para facilitar a depuração do código.
2. Propagar o erro para cima na pilha de chamadas, entregando o objeto da exceção diretamente ao método que tem interesse na sua ocorrência. Tradicionalmente, o código de erro teria de ser propagado método a método, aumentando a complexidade do código.
3. Agrupar e diferenciar os tipos de erros. Java trata as exceções lançadas como objetos que, naturalmente, podem ser classificados com base na hierarquia de classes. Por exemplo, erros definidos mais abaixo na hierarquia são mais específicos, ao contrário dos que se encontram mais próximos do topo, seguindo o princípio da especialização/generalização da programação orientada a objetos.

Atividade 1

Sobre os conceitos básicos de tratamento de exceção em Java, analise as afirmativas:

I. Um erro de sintaxe causa uma exceção.

II. Um desvio no fluxo principal do programa é uma exceção.

III. O tratamento de exceção é feito pela Máquina Virtual Java.

Está correto o que se afirma em

A

I.

B

II.

C

III.

D

I e II.

E

II e III.

A alternativa C está correta.

Uma exceção é causada por um erro em tempo de execução. O erro de sintaxe é um erro que ocorre em tempo de compilação, por isso a afirmativa I é falsa. O erro em tempo de execução desvia o programa de seu fluxo normal (previsto). O desvio do fluxo principal decorre de alguma condição prevista em tempo de programação, e é acionado pela verificação de condições inseridas nas estruturas de desvio da linguagem (if, while etc.). Logo, a afirmativa II também é falsa. Por fim, o tratamento de exceções é um mecanismo da linguagem Java e, portanto, suportado pela Máquina Virtual Java, que é o software que executa os programas em Java. Assim, a afirmativa III é a única verdadeira.

Classificando as exceções em Java

Em Java todos os objetos derivam de um ancestral comum, a classe Object. Incluem-se aí as exceções e todas as suas especializações. Entender essa arquitetura de classes é importante para empregar adequadamente o mecanismo de tratamento de exceções.

Assista ao vídeo e compreenda a hierarquia de classes utilizadas em Java para realizar o tratamento de exceções, assim como as exceções implícitas e explicitas.

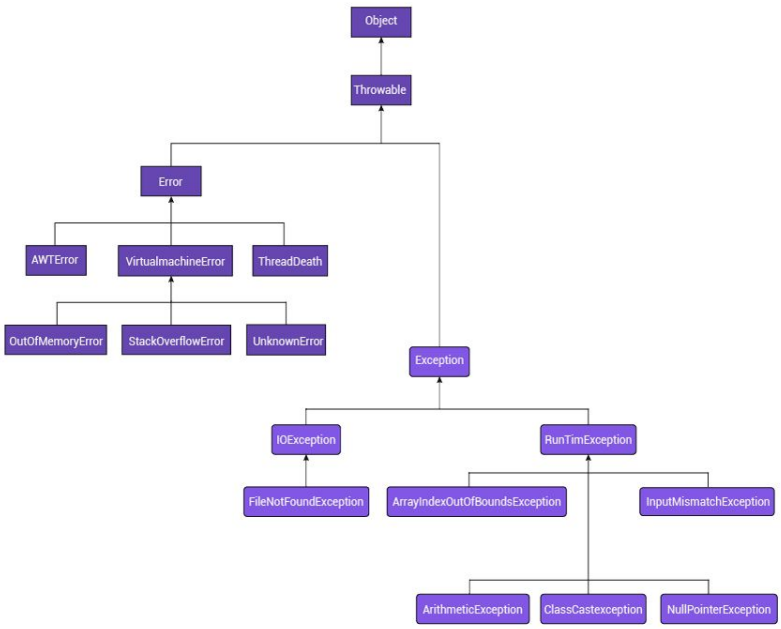
Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Todos os tipos de exceção em Java são subclasses da classe Throwable. Veja algumas dessas subclasses:

Conteúdo interativo

Acesse a versão digital para ver mais detalhes da imagem abaixo.



Hierarquia parcial de classes de exceções em Java.

A primeira separação feita agrupa as exceções em dois subtipos. Vamos conhecê-los!

Error

Agrupa as exceções que, em situações normais, não precisam ser capturadas pelo programa. Um estouro de pilha (stack overflow) é um exemplo de exceção que pertence ao subtipo Error. Em situações normais, não se espera que o programa cause esse tipo de erro, mas ele pode ocorrer e, nesse caso, provoca uma falha catastrófica. Veja que esse erro ocorre durante a execução do software. Os erros que acontecem em tempo de execução são os que a classe Error agrupa. Ela é utilizada para o sistema de execução Java indicar erros relacionados ao ambiente de execução.

Exception

Agrupa as exceções que os programas deverão capturar e permite a extensão pelo programador para criar suas próprias exceções. Uma importante subclasse de Exception é a classe RuntimeException, que corresponde às exceções como a divisão por zero ou o acesso a índice inválido de array e que são automaticamente definidas.

Quando uma exceção não é adequadamente capturada e tratada, o interpretador mostrará uma mensagem de erro com informações sobre o problema ocorrido e encerrará o programa. Vejamos o exemplo mostrado no **código 2**.

```
java

    public class Principal {
        public static void main ( String args [ ] ) throws InterruptedException {
            int divisor , dividendo , quociente = 0;
            String controle = "s";

            Scanner s = new Scanner ( System.in );
            do {
                System.out.println ( "Entre com o dividendo." );
                dividendo = s.nextInt();
                System.out.println ( "Entre com o divisor." );
                divisor = s.nextInt();
                quociente = dividendo / divisor;
                System.out.println ( "O quociente é: " + quociente );
                System.out.println ( "Repetir?" );
                controle = s.next().toString();
            } while ( controle.equals( "s" ) );
            s.close();
    }
}
```

Código 2: Exemplo de código sujeito a exceção em tempo de execução.

A execução desse código é sujeita a erros ocasionados na execução. Nas condições normais, o programa executará indefinidamente até que escolhamos um valor diferente de “s” para a pergunta “Repetir?”. Um dos erros a que ele está sujeito é a divisão por zero. Como não estamos tratando essa exceção, se ela ocorrer teremos o fim do programa. Observe!

```
java

Entre com o dividendo.
10
Entre com o divisor.
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at com.mycompany.teste.Principal.main(Principal.java:26)
Command execution failed.
org.apache.commons.exec.ExecuteException: Process exited with an error: 1 (Exit value: 1)
        at org.apache.commons.exec.DefaultExecutor.executeInternal (DefaultExecutor.java:
404)
        at org.apache.commons.exec.DefaultExecutor.execute (DefaultExecutor.java:166)
        at org.codehaus.mojo.exec.ExecMojo.executeCommandLine (ExecMojo.java:982)
        at org.codehaus.mojo.exec.ExecMojo.executeCommandLine (ExecMojo.java:929)
        at org.codehaus.mojo.exec.ExecMojo.execute (ExecMojo.java:457)
        at org.apache.maven.plugin.DefaultBuildPluginManager.executeMojo
(DefaultBuildPluginManager.java:137)
        at org.apache.maven.lifecycle.internal.MojoExecutor.execute (MojoExecutor.java:
210)
        at org.apache.maven.lifecycle.internal.MojoExecutor.execute (MojoExecutor.java:
156)
        at org.apache.maven.lifecycle.internal.MojoExecutor.execute (MojoExecutor.java:
148)
        at org.apache.maven.lifecycle.internal.LifecycleModuleBuilder.buildProject
(LifecycleModuleBuilder.java:117)
        at org.apache.maven.lifecycle.internal.LifecycleModuleBuilder.buildProject
(LifecycleModuleBuilder.java:81)
        at
org.apache.maven.lifecycle.internal.builder.singlethreaded.SingleThreadedBuilder.build
(SingleThreadedBuilder.java:56)
        at org.apache.maven.lifecycle.internal.LifecycleStarter.execute
(LifecycleStarter.java:128)
        at org.apache.maven.DefaultMaven.doExecute (DefaultMaven.java:305)
        at org.apache.maven.DefaultMaven.doExecute (DefaultMaven.java:192)
        at org.apache.maven.DefaultMaven.execute (DefaultMaven.java:105)
        at org.apache.maven.cli.MavenCli.execute (MavenCli.java:957)
        at org.apache.maven.cli.MavenCli.doMain (MavenCli.java:289)
        at org.apache.maven.cli.MavenCli.main (MavenCli.java:193)
        at jdk.internal.reflect.NativeMethodAccessorImpl.invoke0 (Native Method)
        at jdk.internal.reflect.NativeMethodAccessorImpl.invoke
(NativeMethodAccessorImpl.java:64)
        at jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke
(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke (Method.java:564)
        at org.codehaus.plexus.classworlds.launcher.Launcher.launchEnhanced
(Launcher.java:282)
        at org.codehaus.plexus.classworlds.launcher.Launcher.launch (Launcher.java:225)
        at org.codehaus.plexus.classworlds.launcher.Launcher.mainWithExitCode
(Launcher.java:406)
        at org.codehaus.plexus.classworlds.launcher.Launcher.main (Launcher.java:347)
------------------------------------------------------------------------
BUILD FAILURE
------------------------------------------------------------------------
Total time:  20.179 s
Finished at: 2021-05-22T14:03:57-03:00
------------------------------------------------------------------------
Failed to execute goal org.codehaus.mojo:exec-maven-plugin:3.0.0:exec (default-cli) on
project teste: Command execution failed.: Process exited with an error: 1 (Exit value: 1)
-> [Help 1]

To see the full stack trace of the errors, re-run Maven with the -e switch.
Re-run Maven using the -X switch to enable full debug logging.

For more information about the errors and possible solutions, please read the following
articles:
[Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoExecutionException
```

Código 3: Exemplo de código sujeito a erro divisão por zero.

Vendo a linha número 42, observamos a mensagem “(...) Process exited with an error (...)”, que indica o término abrupto do programa. Veja agora como a captura e o tratamento da exceção beneficia o programa. Vamos substituir a linha 12 do **código 2** pelas instruções mostradas no **código 4** a seguir.

```
java

    try {
        quociente = dividendo / divisor;
    } catch (Exception e)
           {
        System.out.println( "ERRO: Divisão por zero!" );
    }
```

Código 4: Envolvendo a divisão num bloco try-catch.

Confira a saída para o código modificado.

```
java

Entre com o dividendo.
10
Entre com o divisor.
0
ERRO: Divisão por zero!
O quociente é: 0
Repetir?
s
Entre com o dividendo.
10
Entre com o divisor.
5
O quociente é: 2
Repetir?
n
------------------------------------------------------------------------
BUILD SUCCESS
------------------------
```

Código 5: Saída do código modificado.

Note que agora o programa foi encerrado de maneira normal, apesar da divisão por zero provocada na linha 4. Quando o erro ocorreu, a exceção foi lançada e capturada pelo bloco catch, como vemos na linha 5. A linha 6 mostrou o valor de “quociente” sendo zero, conforme inicializamos a variável na linha 3 do **código 2**. Se você tentar compilar o programa sem essa inicialização, verá que sem o bloco try-catch não é gerado erro, mas, quando o utilizamos, o interpretador Java nos obriga a inicializar a variável.

Já compreendemos o essencial do tratamento de erros em Java, então veremos a seguir os tipos de exceções implícitas e explícitas (ORACLE INC., s.d.). Também veremos como declarar novos tipos de exceções.

Exceções implícitas

Definidas nos subtipos Error e RunTimeException e suas derivadas, são exceções ubíquas, isto é, que podem ocorrer em qualquer parte do programa e normalmente não são causadas diretamente pelo programador. Por essa razão, possuem um tratamento especial e não precisam ser manualmente lançadas.

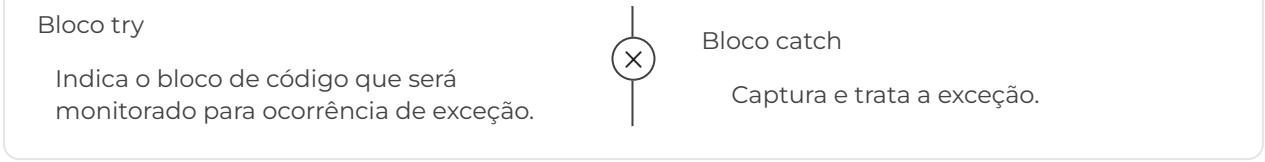
Verificando o **código 2**, perceberemos que a divisão por zero não está presente no código. A linha 12 apenas codifica uma operação de divisão. Ela não realiza a divisão que produzirá a exceção, a menos que durante a execução o usuário entre com o valor zero para o divisor, situação em que o Java runtime detecta o erro e lança a exceção ArithmeticException.

Outra exceção implícita: o problema de estouro de memória (OutOfMemoryError) pode acontecer em qualquer momento, com qualquer instrução sendo executada, pois sua causa não é a instrução em si, mas um conjunto de circunstâncias de execução que a causaram. Também são exemplos as exceções NullPointerException e IndexOutOfBoundsException, entre outras.

> As exceções implícitas são lançadas pelo próprio Java Runtime, não sendo necessária a declaração de um método throw para ela. Quando uma exceção desse tipo ocorre, é gerada uma referência implícita para a instância lançada.

A saída do **código 2** mostra o lançamento da exceção que nessa versão não foi tratada, gerando o encerramento anormal do programa.

O **código 4** nos mostra um bloco try-catch. Considere suas funções a seguir.



Entretanto, mesmo nessa versão modificada o programador não está lançando a exceção, apenas a captura. A exceção continua sendo lançada automaticamente pelo Java Runtime. Cabe dizer, contudo, que não há impedição ao lançamento manual da exceção. Veja agora o **código 6**, que modifica o **código 4**, passando a lançar a exceção.

```
java

try {
    if ( divisor ==0 )
        throw new ArithmeticException ( "Divisor nulo." );
    quociente = dividendo / divisor;
}
catch (Exception e)
{
    System.out.println( "ERRO: Divisão por zero! " + e.getMessage() );
}
```

Código 6: Lançando manualmente uma exceção implícita.

Veja a saída produzida pela execução dessa versão modificada:

```
java

Entre com o dividendo.
12
Entre com o divisor.
0
ERRO: Divisão por zero! Divisor nulo.
O quociente é: 0
Repetir?
n
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Código 7: Saída do código modificado.

Apesar de o programador ter a capacidade de lançar manualmente uma exceção, os chamadores do método que realiza o lançamento não são obrigados pelo interpretador a tratar a exceção. Apenas exceções que não são implícitas, isto é, lançadas pelo Java Runtime, devem obrigatoriamente ser tratadas. Por isso também dizemos que exceções implícitas são contornáveis, podemos simplesmente ignorar seu tratamento. Isso, porém, não tende a ser uma boa ideia, já que elas serão lançadas e vão provocar a saída anormal do programa.

Exceções explícitas

Todas as exceções que não são implícitas são consideradas explícitas. Esse tipo de exceção, de maneira oposta às implícitas, é considerado incontornável. Quando um método usa uma exceção explícita, ele obrigatoriamente deve usar uma instrução throw no corpo do método para criar e lançar a exceção.

O programador pode escolher não capturar essa exceção e tratá-la no método em que ela é lançada. Ou fazê-lo e ainda assim desejar propagar a exceção para os chamadores do método. Em ambos os casos, deve ser usada uma cláusula throws na assinatura do método que declara o tipo de exceção a ser lançada se um erro ocorrer. Nesse caso, os chamadores precisarão envolver a chamada em um bloco try-catch.

O **código 8** mostra um exemplo de exceção explícita (IllegalAccessException). Essa exceção é uma subclasse de ReflectiveOperationException, que, por sua vez, descende da classe Exception. Logo, não pertence aos subtipos Error ou RuntimeException que correspondem às exceções explícitas.

```
java

public class Arranjo {
    int [] vetor = { 1 , 2 , 3, 4 };
    int getElemento ( int i ) {
        try {
            if ( i < 0 || i > 3 )
                throw new IllegalAccessException ();
        } catch ( Exception e ) {
            System.out.println ( "ERRO: índice fora dos limites do vetor." );
        }
        return vetor [i];
    }
}
```

Código 8: Exemplo de exceção explícita.

Como é uma exceção explícita, ela precisa ser tratada em algum momento. No caso mostrado no **código 8**, essa exceção explícita está abarcada por um bloco try-catch, o que livra os métodos chamadores de terem de lidar com a exceção gerada.

Quais seriam as consequências se o programador optasse por não tratar a exceção explícita localmente?

Nesse caso, o bloco try-catch seria omitido com a supressão das linhas 4, 7, 8 e 9. A linha 6, que lança a exceção, teria de permanecer, pois é uma exceção explícita. Ao fazer isso, o interpretador Java obrigaria o acréscimo de uma cláusula throws na assinatura do método, informando aos chamadores as exceções que o método pode lançar.

O **código 8** pode ser invocado simplesmente fazendo-se como no **código 9** a seguir. Observe que a invocação, vista na linha 5, prescinde do uso de bloco try-catch.

```java
public class Chamador {
    Arranjo arrj = new Arranjo ( );

    int invocaGetElemento ( int i ) {
        return arrj.getElemento ( i );
    }
}
```

Código 9: Invocando um método que lança uma exceção explícita.

A situação, porém, é diferente se o método getElemento (i) propagar a exceção por meio de uma cláusula throws. Nesse caso, mesmo que a exceção seja tratada localmente, os chamadores deverão envolver a chamada ao método num bloco try-catch ou também propagar a exceção.

Atividade 2

Sobre exceções, assinale a alternativa correta.

A

Exceções implícitas não podem ser propagadas.

B

Exceções definidas na classe UnknownError são implícitas.

C

Se uma exceção implícita não for capturada, será gerado erro de compilação.

D

Se uma exceção é lançada pela instrução “throw”, então ela é uma exceção explícita.

E

Exceções implícitas ocorrem sempre em tempo de compilação.

A alternativa B está correta.

Exceções são implícitas quando definidas nas classes Error e RunTimeException e suas derivadas. A classe UnknownError é uma subclasse de Error.

Declarando novos tipos de exceção

Até o momento, todas as exceções que vimos são providas pela própria API Java. Essas exceções cobrem os principais erros que podem ocorrer num software, como:

- Erros de operações de entrada e saída (E/S)
- Problemas com operações aritméticas
- Operações de acesso de memória ilegais

Adicionalmente, fornecedores de bibliotecas costumam prover suas próprias exceções para cobrir situações particulares de seus programas. Então, no desenvolvimento de um software simples, possivelmente isso deve bastar. Mas nada impede que o programador crie sua própria classe de exceções para lidar com situações específicas.

Assista ao vídeo e conheça os mecanismos da linguagem Java para customizar e criar novos tipos de exceções de acordo com as necessidades do que está sendo implementado.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Ao prover software para ser usado por outros, como bibliotecas, motores ou outros componentes, declarar seu próprio conjunto de exceções é uma boa prática de programação e agrega qualidade ao produto. Felizmente, isso é possível utilizando o mecanismo de herança.

Para criar uma classe de exceção, deve-se obrigatoriamente estender uma classe de exceção existente, pois isso irá legar à nova classe o mecanismo de manipulação de exceções. Uma classe de exceção não possui qualquer membro a não ser os 4 construtores (DEITEL; DEITEL, 2017). Vamos conhecê-los!

1. Um que não recebe argumentos e passa uma String — mensagem de erro padrão — para o construtor da superclasse.
2. Um que recebe uma String — mensagem de erro personalizada — e a passa para o construtor da superclasse.
3. Um que recebe uma String — mensagem de erro personalizada — e um objeto Throwable — para encadeamento de exceções — e os passa para o construtor da superclasse.
4. Um que recebe um objeto Throwable — para encadeamento de exceções — e o passa para o construtor da superclasse.

Embora o programador possa estender qualquer classe de exceções, é preciso considerar qual superclasse se adequa melhor à situação.

Relembrando

Estamos nos valendo do mecanismo de herança, um importante conceito da programação orientada a objetos.

Em uma hierarquia de herança, os níveis mais altos generalizam comportamentos, enquanto os mais baixos especializam. Logo, a classe mais apropriada será aquela de nível mais baixo cujas exceções ainda representem uma generalização das exceções definidas na nova classe. Em última instância, podemos estender diretamente da classe Throwable.

Como qualquer classe derivada de Throwable, um objeto dessa subclasse conterá um instantâneo da pilha de execução de sua thread no momento em que foi criado. Ele também poderá conter uma String com uma mensagem de erro, como vimos antes, a depender do construtor utilizado. Isso também abre a possibilidade

para que o objeto tenha uma referência para outro objeto throwable que tenha causado sua instanciação, caso em que há um encadeamento de exceções.

> Ao se criar uma classe de exceção, deve-se considerar a reescrita do método toString (). Além de fornecer uma descrição para a exceção, essa reescrita permite ajustar as informações que são impressas pela invocação desse método, potencialmente melhorando a legibilidade das informações mostradas na ocorrência das exceções.

O **código 10** mostra um exemplo de exceção criada para indicar problemas na validação do CNPJ. Veja!

```
java

public class ErroValidacaoCNPJ extends Throwable {
    private String msg_erro;

    ErroValidacaoCNPJ ( String msg_erro ) {
        this.msg_erro = msg_erro;
    }

    @Override
    public String toString ( ) {
        return "ErroValidacaoCNPJ: " + msg_erro;
    }
}
```

Código 10: Nova classe de exceção.

Repare o uso do **código 10** nas linhas 22 e 61 da classe Juridica (**Código 11**). Veja também que a exceção é lançada e encadeada, ficando o tratamento a cargo de quem invoca atualizarID ().

java

```
public class Juridica extends Pessoa {
    public Juridica ( String razao_social , Calendar data_criacao, String CNPJ , Endereco
endereco , String nacionalidade , String sede ) {
        super ( razao_social , data_criacao, CNPJ , endereco , nacionalidade ,
sede);
    }
    @Override
    public boolean atualizarID ( String CNPJ ) throws ErroValidacaoCNPJ {
        if ( validaCNPJ ( CNPJ ) ) {
            this.identificador = CNPJ;
            return true;
        }
        else {
            System.out.println ( "ERRO: CNPJ invalido!" );
            return false;
        }
    }
    private boolean validaCNPJ ( String CNPJ ) throws ErroValidacaoCNPJ {
        char DV13, DV14;
        int soma, num, peso, i, resto;
        //Verifica sequência de dígitos iguais e tamanho (14 dígitos)
        if ( CNPJ.equals("00000000000000") || CNPJ.equals("11111111111111") ||
CNPJ.equals("22222222222222") || CNPJ.equals("33333333333333") ||
CNPJ.equals("44444444444444") || CNPJ.equals("55555555555555") ||
CNPJ.equals("66666666666666") || CNPJ.equals("77777777777777") ||
CNPJ.equals("88888888888888") || CNPJ.equals("99999999999999") || (CNPJ.length() != 14) )
        {
            throw new ErroValidacaoCNPJ ( "Entrada invalida!" );
            //   return(false);
        }
        try {
            //1º Dígito Verificador
            soma = 0;
            peso = 2;
            for ( i = 11 ; i >= 0 ; i-- ) {
                num = (int)( CNPJ.charAt ( i ) - 48 );
                soma = soma + ( num * peso );
                peso++;
                if ( peso == 10 )
                peso = 2;
            }
            resto = soma % 11;
            if ( ( resto  == 0 ) || ( resto == 1 ) )
                DV13 = '0';
            else
                DV13 = (char)( ( 11 - resto ) + 48 );
            //2º Dígito Verificador
            soma = 0;
            peso = 2;
            for ( i = 12 ; i >= 0 ; i-- ) {
                num = (int) ( CNPJ.charAt ( i ) - 48 );
                soma = soma + ( num * peso );
                peso++;
                if ( peso == 10 )
                peso = 2;
            }
            resto = soma % 11;
            if ( ( resto == 0 ) || ( resto == 1 ) )
                DV14 = '0';
            else
                DV14 = (char) ( ( 11 - resto ) + 48 );
            //Verifica se os DV informados coincidem com os calculados
            if ( ( DV13 == CNPJ.charAt ( 12 ) ) && ( DV14 == CNPJ.charAt ( 13 ) ) )
                return true;
            else
            {
```

Código 11: Classe Juridica com lançamento de exceção.

Veja agora a saída:

```
java

ErroValidacaoCNPJ: Entrada invalida!
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Código 12: Saída provocada por uma exceção.

Atividade 3

A declaração de novos tipos de exceção é um importante recurso para o programador. Sobre isso, indique a alternativa correta.

A

Todas as exceções em Java herdam diretamente da classe Throwable.

B

A declaração de uma nova exceção em Java deve estender diretamente a superclasse Throwable.

C

Ao declarar uma nova exceção a partir da extensão direta de Throwable, a classe derivada precisa implementar o mecanismo de tratamento da exceção.

D

Apenas as exceções nativas da linguagem podem ser encadeadas.

E

Mesmo exceções declaradas pelo programador que usem um construtor vazio possuem o contexto de execução em que ocorreram.

A alternativa E está correta.

Todo objeto instanciado a partir de uma classe derivada de Throwable possui um instantâneo da pilha de execução no momento de sua criação. Se o construtor for vazio, isso significa que a exceção não pode receber um objeto do tipo Throwable, ou seja, ela não é capaz de receber o contexto de execução de exceções anteriores que a causaram. Mesmo assim, ela possui seu próprio contexto de execução e pode passá-lo adiante.

Usando exceções em Java

Apresentaremos agora um caso prático para examinar a utilidade das exceções no desenvolvimento de software, usando o cálculo recursivo do fatorial como exemplo. O sistema é composto por duas classes, a Principal e a Fatorial, e funciona solicitando ao usuário um número para o qual o fatorial será calculado. O programa realiza o cálculo por meio de uma chamada recursiva ao método calcularFatorial() da classe Fatorial, contabilizando quantas chamadas recursivas são feitas. O objetivo do estudo é explorar exceções, forçando duas exceções da classe Error: IOException e StackOverflowError.

Assista ao vídeo e confira o código fonte de um programa em Java que valida uma senha usando exceções.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Agora você apresentará tratamento semelhante para o código que calcula o n-ésimo termo da sequência de Fibonacci. A seguir, exibimos o código completo das classes Principal e Fibonacci. A classe Principal é a que contém o método main (String args [ ]), responsável por implementar a interatividade do programa.

```
java

//imports
import java.io.IOException;
import java.util.Scanner;

public class Principal {
    private static Fibonacci fib;
    private static Scanner entrada;

    public static void main ( String args [ ] ) {
        double num = 0;
        entrada = new Scanner ( System.in );
        fib = new Fibonacci ();
        do {
            System.out.println ( "Entre com um numero não negativo ou \"-1\" para sair:
");
            num = entrada.nextDouble();
            if ( num == -1 )
                System.exit ( 0 );
            System.out.println ( "O " + num + “-esimo termo de Fibonacci eh: " +
fib.CalcularFibonacci( num ) );
        } while ( num >= 0 );
    }
}
```

Código 13: Classe Principal.

```java
//imports
// Não há

public class Fibonacci {
    //Atributos
    private static int conta_chamada = 0; //conta o número de chamadas recursivas

    public double CalcularFibonacci ( double num ) {
        conta_chamada++;
        System.out.println ( "Chamada recursiva nr: " + conta_chamada );
        if ( num != 2 && num != 1 )
            return CalcularFibonacci ( num - 1 ) + CalcularFibonacci (num – 2);
        else
            return 1;
    }
}
```

Código 14: Classe Fibonacci.

Para verificar o funcionamento, execute o programa e insira qualquer valor entre 0 e 100. O programa realizará o cálculo sem problemas todas as vezes que você repetir o procedimento.

Feito o teste inicial, vamos explorar as situações das exceções. A primeira exceção que queremos forçar é a de estouro de pilha (StackOverflowError). Esse é um caso interessante, porque o estouro de pilha depende, entre outros fatores, da configuração da máquina virtual. Ou seja, pode ser que o mesmo valor de entrada gere a exceção em uma MVJ e não a gere em outra. Você pode explorar esse limite. Nesse caso, sugerimos que você comece com 999 e sempre aumente em potências de 10 (9999, 99999 etc.). Mas não precisamos fazer isso. Pela forma como o método está implementado, qualquer número negativo diferente de -1 (valor de escape) provocará uma recursividade ilimitada que irá estourar a pilha. A seguir mostramos a parte da saída para o cálculo do termo de ordem -2 (não é definido).

Repare que a exceção foi lançada independentemente da inclusão do comando para isso pelo programador. Como vimos, as exceções da classe Error têm esse comportamento.

Vamos agora fazer uma ligeira modificação para tratar a exceção de estouro de pilha.

Faça você mesmo!

Considere apenas o código mostrado na listagem a seguir.

```java
//imports
import java.io.IOException;
import java.util.Scanner;

public class Principal {
    //Atributos
    private static Scanner entrada;

    public static void main ( String args [ ] ) {
        double num = 0;
        entrada = new Scanner ( System.in );

        do {
            System.out.println ( "Entre com um numero de 4 dígitos ou \"-0001\" para
sair: ");
            if ( num == -1 )
                System.exit ( 0 );
            num = Double.parseDouble( lerEntrada ( 4 ) );
            System.out.println ( "O numero lido eh: " + num );
        } while ( num >= 0 );
    }

    private static String lerEntrada ( int tam_entrada ) throws IOException {
        String entrada = null;

            entrada = new String ( System.in.readNBytes ( tam_entrada ) );
        return entrada;
    }
}
```

A

É suficiente colocar a linha 17 dentro de um bloco try e capturar a exceção lançada com catch (Error e).

B

É suficiente colocar a linha 17 dentro de um bloco try e capturar a exceção lançada com catch (IOException e).

C

É necessário e suficiente colocar somente a chamada lerEntrada (4) da linha 17 dentro de um bloco try e capturar a exceção lançada com catch (Error e).

D

É necessário e suficiente colocar somente a chamada lerEntrada (4) da linha 17 dentro de um bloco try e capturar a exceção lançada com catch (IOException e).

E

É necessário colocar as chamadas das linhas 17 e 22 dentro de um bloco try e capturar a exceção lançada em ambos os casos com catch (IOException e).

A alternativa B está correta.

A exceção IOException precisa ser capturada, isto é, estar num bloco try-catch, ou ser informado ao chamador que ela não será tratada. Entretanto, a exceção não pode ser capturada com um comando catch (Error e), embora IOException seja um subtipo de Error. Se isso fosse possível, a informação de qual foi a exceção gerada se perderia, anulando o propósito desse mecanismo. Como nada impede que outras instruções estejam dentro de um bloco try-catch além daquela que se quer monitorar para o lançamento de exceção, não é necessário que apenas a chamada lerEntrada (4) da linha 17 seja inserida no bloco try-catch. Da mesma forma, colocar as linhas 17 e 22 dentro de um bloco try-catch resolve o problema, mas não é necessário.

Comandos relativos ao tratamento de exceções em Java

Estudaremos agora o tratamento de exceções em Java destacando sua flexibilidade e importância para os programadores. Seu uso, entretanto, exige conhecimento detalhado de seu funcionamento para evitar consequências indesejadas.

Veja a seguir as três cláusulas principais:

Throw

Permite lançar exceções explicitamente.

Throws

Altera a abordagem no tratamento das exceções lançadas.

Finally

Oferece uma maneira de lidar com problemas gerados pelo desvio no fluxo do programa.

Assista ao vídeo e conheça os comandos finally, throw e throws, bem como sua estrutura sintática e semântica.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Comando finally

Imagine que um programador está manipulando um grande volume de dados. Para isso, ele usa estruturas de dados que ocupam significativo espaço de memória. De repente o software sofre uma falha catastrófica, por um acesso ilegal de memória por exemplo, e encerra anormalmente. O que ocorre com toda a memória alocada? Considere as seguintes linguagens:

C++

Quando não implementa um coletor de lixo, a porção de memória reservada para uso permanecerá alocada e indisponível até que o processo seja terminado.

Máquina Virtual Java

Quando implementa o coletor de lixo, isso não ocorre. A porção de memória permanecerá alocada somente até a próxima execução do coletor. Vazamentos, porém, ainda podem ocorrer se o programador mantiver referências para objetos não utilizados.

O problema que acabamos de descrever é chamado de vazamento de memória e é um dos diversos tipos de vazamento de recursos. Infelizmente, nesses casos o coletor de lixo não resolve a questão. Confira outros exemplos de vazamentos:

- Conexões não encerradas
- Acessos a arquivos não fechados

- Dispositivos não liberados, entre outros

Java oferece, porém, um mecanismo para lidar com tais situações: o **bloco finally**. Esse bloco será executado independentemente de uma exceção ser lançada no bloco try ao qual ele está ligado. Aliás, mesmo que haja instruções return, break ou continue no bloco try, finally será executado. A única exceção é se a saída se der pela invocação de System.exit, o que força o término do programa.

Por essa característica, podemos utilizar o bloco finally para liberar os recursos, evitando o vazamento. Quando nenhuma exceção é lançada no bloco try, os blocos catch não são executados e o compilador segue para a execução do bloco finally. Alternativamente, uma exceção pode ser lançada. Nessa situação, o bloco catch correspondente é executado e a exceção é tratada. Em seguida, o bloco finally é executado e, se possuir algum código de liberação de recursos, o recurso é liberado. Uma última situação se dá com a não captura de uma exceção. Também nesse caso o bloco finally será executado e o recurso poderá ser liberado.

O **Código 1** mostra um exemplo de uso de finally na linha 21.

```
java

public class Principal {
    public static void main ( String args [ ] ) throws InterruptedException {
        int divisor , dividendo , quociente = 0;
        String controle = "s";

        Scanner s = new Scanner ( System.in );
        do {
            System.out.println ( "Entre com o dividendo." );
            dividendo = s.nextInt();
            System.out.println ( "Entre com o divisor." );
            divisor = s.nextInt();
            try {
                if ( divisor ==0 )
                    throw new ArithmeticException ( "Divisor nulo." );
                quociente = dividendo / divisor;
            }
            catch (Exception e)
            {
                System.out.println( "ERRO: Divisão por zero! " + e.getMessage() );
            }
            finally
            {
                System.out.println("Bloco finally.");
            }
            System.out.println ( "O quociente é: " + quociente );
            System.out.println ( "Repetir?" );
            controle = s.next().toString();
        } while ( controle.equals( "s" ) );
        s.close();
    }
}
```

Código 1: Exemplo de uso de finallyA.

Observe a seguinte saída:

```
java

--- exec-maven-plugin:3.0.0:exec (default-cli) @ teste ---
Entre com o dividendo.
10
Entre com o divisor.
2
Bloco finally.
O quociente é: 5
Repetir?
s
Entre com o dividendo.
10
Entre com o divisor.
0
ERRO: Divisão por zero! Divisor nulo.
Bloco finally.
O quociente é: 5
Repetir?
n
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Código 2: Saída do uso de finallyA.

Fica claro que finally foi executado independentemente de a exceção ter sido lançada (linhas 13, 14 e 15) ou não (linhas 5 e 6).

O bloco finally é opcional. Entretanto, um bloco try exige pelo menos um bloco catch ou um bloco try. Ou seja, se usarmos um bloco try, podemos omitir o bloco catch desde que tenhamos um bloco finally. No entanto, diferentemente de catch, um try pode ter como correspondente uma, e somente uma, cláusula finally. Assim, se suprimíssemos o bloco catch da linha 17 do **código 1**, mas mantivéssemos o bloco finally, o programa compilaria sem problema.

Como dissemos, mesmo que um desvio seja causado por break, return ou continue, finally é executado. Logo, se inserirmos uma instrução return após a linha 15, ainda veremos a saída “Bloco finally.” sendo impressa.

Comando throw

Já vimos que Java lança automaticamente as exceções da classe Error e RunTimeException, as chamadas exceções implícitas. Mas para lançarmos manualmente uma exceção, precisamos nos valer do comando throw. Por meio dele lançamos as chamadas exceções explícitas. A sintaxe de throw é bem simples e consiste no comando seguido da exceção que se quer lançar.

O objeto lançado por throw deve ser um objeto da classe Throwable ou de uma de suas subclasses. Ou seja, deve ser uma exceção. Não é possível lançar tipos primitivos — int, string, char — ou mesmo objetos da classe Object, pois esses elementos não podem ser usados como exceção. Uma instância Throwable pode ser obtida pela passagem de parâmetro na cláusula catch ou com o uso do operador new.

Quando o compilador encontra uma cláusula throw, desvia a execução do programa. Isso significa que nenhuma instrução existente após a cláusula será executada. Nesse ponto, o compilador desvia a execução para o bloco try mais próximo em busca de uma cláusula catch que trate a exceção lançada. Caso não seja encontrada, o próximo bloco try mais externo é inspecionado e assim sucessivamente. Se nenhum bloco catch tratar a exceção lançada, então o tratador padrão encerra o programa e imprime o registro da pilha obtido quando do lançamento da exceção. Ao contrário, se for identificado um bloco catch capaz de tratar a exceção, a execução do programa é desviada para esse bloco.

Vamos modificar o **código 1** movendo a lógica do programa para a classe Calculadora. Veja os **códigos 3** e **4** correspondentes.

```java
public class Calculadora {
    public int divisao ( int dividendo , int divisor )
    {
        try {
            if ( divisor == 0 )
                throw new ArithmeticException ( "Divisor nulo." );
        }
        catch (Exception e)
        {
            System.out.println( "ERRO: Divisão por zero! " + e.getMessage() );
            return 999999999;
        }
        return dividendo / divisor;
    }
}
```

Código 3: Classe Calculadora.

```java
public class Principal {
    public static void main ( String args [ ] ) {
        int dividendo, divisor;
        String controle = "s";

        Calculadora calc = new Calculadora ( );
        Scanner s = new Scanner ( System.in );
        do {
            System.out.println ( "Entre com o dividendo." );
            dividendo = s.nextInt();
            System.out.println ( "Entre com o divisor." );
            divisor = s.nextInt();
            System.out.println ( "O quociente é: " + calc.divisao ( dividendo ,
divisor ) );
            System.out.println ( "Repetir?" );
            controle = s.next().toString();
        } while ( !controle.equals( "n" ) );
        s.close();
    }
}
```

Código 4: Classe Principal modificada.

Nessa versão modificada, a exceção é lançada na linha 6 do **código 3**. É possível ver que a criação da exceção se dá mediante o operador new, que instancia a classe ArithmeticException. Uma vez que a cláusula throw é executada e a instrução lançada, o interpretador Java busca o bloco try-catch mais próximo, que está na linha 4. Esse bloco define um contexto de exceção que é capaz de tratar a exceção lançada, como vemos na linha 8 (bloco catch correspondente). A execução do programa é, então, desviada para o bloco catch que recebe a referência do objeto da exceção como parâmetro. A linha 10 imprime, por meio do método getMessage () definido na classe Throwable, a mensagem passada como parâmetro para o construtor na linha 6.

O que aconteceria se removêssemos o bloco try-catch, mantendo apenas o lançamento de exceção?

Se não houver um bloco catch que trate a exceção lançada, ela será passada para o tratador padrão de exceções, que encerrará o programa e imprimirá o registro da pilha de execução. Outra forma de lidar com isso seria definir um contexto de exceção no chamador (linha 13 do **código 4**) e propagar a exceção para que fosse tratada externamente.

O mecanismo de propagação de exceções é o que veremos a seguir.

Comando throws

Vimos que um método pode lançar uma exceção, mas ele não é obrigado a tratá-la. Ele pode transferir essa responsabilidade para o chamador, que também pode transferir para o seu próprio chamador e assim repetidamente. Mas quando um método pode lançar uma exceção e não a trata, ele deve informar isso aos seus chamadores explicitamente, a fim de permitir que o chamador se prepare para lidar com a possibilidade do lançamento de uma exceção. Essa notificação é feita pela adição da cláusula throws na assinatura do método, após o parêntese de fechamento da lista de parâmetros e antes das chaves de início de bloco.

A cláusula throws deve listar todas as exceções explícitas que podem ser lançadas no método, mas que não serão tratadas.

Atenção

Não listar as exceções explícitas que não serão tratadas vai gerar erro em tempo de compilação.

Exceções das classes Error e RuntimeException, bem como de suas subclasses não precisam ser listadas. A forma geral de throws é:

<modificadores> <nome do método> <lista de parâmetros> throws <lista de exceções explícitas não tratadas> { <corpo do método> }

A lista de exceções explícitas não tratadas é uma lista formada por elementos separados por vírgulas. Eis um exemplo de declaração de métodos com o uso de throws:

public int acessaDB (int arg1 , String aux) throws ArithmeticException , IllegalAccessException { ...}

> Embora não seja necessário listar uma exceção implícita, isso não é um erro e nem obrigará o chamador a definir um contexto de exceção para essa exceção especificamente. No entanto, ele deverá definir um contexto de exceção para tratar da exceção explícita listada.

Vamos modificar a classe Calculadora conforme o **código 5** a seguir. Observando a linha 2, constatamos a instrução throws. Notamos também que o método divisão () ainda pode lançar uma exceção, mas agora não há mais um bloco try-catch para capturá-la.

```
java

public class Calculadora {
    public int divisao ( int dividendo , int divisor ) throws ArithmeticException
    {
        if ( divisor == 0 )
            throw new ArithmeticException ( "Divisor nulo." );
        return dividendo / divisor;
    }
}
```

Código 5: Classe Calculadora com método divisão modificado para propagar a exceção.

Em consequência, precisamos ajustar a classe Principal. É possível ver a definição de um bloco try-catch (linha 13) para capturar e tratar a exceção lançada pelo método divisão (). Veja no **código 6**:

```java
public class Principal {
    public static void main ( String args [ ] ) {
        int dividendo, divisor;
        String controle = "s";

        Calculadora calc = new Calculadora ( );
        Scanner s = new Scanner ( System.in );
        do {
            System.out.println ( "Entre com o dividendo." );
            dividendo = s.nextInt();
            System.out.println ( "Entre com o divisor." );
            divisor = s.nextInt();
            try {
                System.out.println ( "O quociente é: " + calc.divisao ( dividendo ,
divisor ) );
            } catch ( ArithmeticException e ) {
                System.out.println( "ERRO: Divisão por zero! " + e.getMessage() );
            }
            System.out.println ( "Repetir?" );
            controle = s.next().toString();
        } while ( !controle.equals( "n" ) );
        s.close();
    }
}
```

Código 6: Classe Principal com contexto de tratamento de exceção definido para o método divisão ().

Atividade 1

Um programador criou seu conjunto de exceções por meio da extensão da classe Error, com a finalidade de tratar erros de socket em conexões com bancos de dados. O comando que pode ser empregado para garantir o fechamento correto da conexão, mesmo em caso de ocorrência de exceção é

A

try.

B

catch.

C

throw.

D

throws.

E

finally.

A alternativa E está correta.

A ocorrência de exceção causa um desvio não desejado no fluxo do programa, podendo impedir o encerramento da conexão e causando um vazamento de recurso. O comando finally, porém, é executado independentemente do desvio causado pela ocorrência de exceção, o que o torna adequado para a tarefa.

Encadeamento de exceções

Em Java, quando uma exceção causa outra, ocorre o que chamamos de encadeamento de exceções. Ou seja, há uma relação de causa e efeito entre as exceções. Identificar essa situação é importante. E não é muito difícil entender a razão pela qual isso é necessário. Sabemos que, quando uma exceção ocorre, o objeto criado possui o contexto de execução em que a exceção foi gerada. Mas se outra exceção é lançada antes de o tratamento da exceção original finalizar, perdemos acesso ao contexto original e o tratamento da exceção anterior pode ser prejudicado.

Assista ao vídeo e veja que o surgimento de uma exceção na execução de um programa pode não ser único, isto é, mais de uma exceção pode ser gerada por um erro.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

É possível que, ao responder a uma exceção, um método lance outra exceção distinta. É fácil entender como isso ocorre. Em nossos exemplos, o bloco catch sempre se limitou a imprimir uma mensagem. Mas isso se deu por motivos didáticos. Nada impede que o tratamento exija um processamento mais complexo, o que pode levar ao lançamento de exceções implícitas ou mesmo explícitas.

Há um problema, então, se uma exceção for lançada durante o tratamento de outra, pois as informações como o registro da pilha de execução e o tipo de exceção da exceção anterior são perdidos. Nas primeiras versões, a linguagem Java não provia um método capaz de manter essas informações e cada programador desenvolvia sua abordagem.

> Não identificar o encadeamento de exceções tem uma limitação óbvia: a depuração do código se torna mais difícil, já que as informações sobre a causa original do erro se perdem.

Felizmente, as versões mais recentes da linguagem oferecem um mecanismo padrão para lidar com encadeamento de exceções. Por meio dele, um objeto de exceção pode manter todas as informações relativas à exceção original. A chave para isso são os dois construtores, dos quatro que vimos antes, que admitem a passagem de um objeto Throwable como parâmetro.

Esse encadeamento permite associar uma exceção com a exceção corrente, de maneira que a exceção ocorrida no contexto atual da execução possa registrar a exceção que lhe deu causa. Como já foi dito, isso é feito mediante uso dos construtores que admitem um objeto Throwable como parâmetro, pois passando um objeto Throwable pelo construtor, permite-se que a exceção corrente tenha acesso à exceção que a originou.

Além dos construtores que recebem um objeto Throwable como parâmetro, confira dois outros métodos utilizados também da classe Throwable.

getCause ()

Retorna a exceção subjacente à exceção corrente, se houver; caso contrário, retorna null.

initCause (Throwable causeExc)

Permite associar o objeto Throwable com a exceção que o invoca, retornando uma referência.

Os métodos getCause () e initCause () também permitem criar uma associação entre a exceção corrente e outra que lhe originou após a sua criação. Em outras palavras, permite fazer a mesma associação feita quando se usa o construtor, mas, nesse caso, a associação é estabelecida após a exceção ter sido criada.

Uma vez que a exceção tenha sido associada a outra, não é possível modificar tal associação. Logo, não se pode invocar mais de uma vez o método initCause () nem o chamar se o construtor já tiver sido empregado para criar a associação. Agora temos um conjunto de métodos padronizados que implementam com simplicidade o mecanismo de encadeamento de exceções. Mas um ponto pode escapar à primeira vista: o impacto desse mecanismo para a própria **programação OO (orientada ao objeto)**, suportada por Java.

Já falamos diversas vezes sobre a generalização e a especialização de classes em uma hierarquia de classes. Usualmente, quando subimos nessa árvore, lidamos com classes que são generalizações, o que implica diversos níveis de abstração.

Exemplo

Considere o caso de empregados de uma empresa. Podemos definir a classe Empregado e dela derivar os subtipos Estagiário, Diretor e Temporário. Nesse caso, temos dois níveis distintos de abstração, cujas operações envolvidas podem ensejar exceções específicas.

O encadeamento de exceções possibilita a criação de exceções com o mesmo nível de abstração do método que as originou, pois agora é possível mapear exceções de tipos distintos.

Atividade 2

Assinale a alternativa correta sobre o encadeamento de exceções em Java.

A

É o mecanismo pelo qual múltiplas exceções podem ser lançadas.

B

Permite recuperar as exceções sucessivamente na cadeia de lançamento.

C

Permite encadear apenas exceções explicitamente não tratadas e listadas na cláusula throws.

D

Se uma exceção for instanciada com o construtor recebendo null para o objeto Throwable, ela pode ser posteriormente vinculada a outra exceção com o método initCause ().

E

As exceções definidas pelo usuário não podem ser encadeadas com as exceções definidas nas bibliotecas da linguajem Java e vice-versa.

A alternativa B está correta.

O encadeamento de exceções cria uma ligação entre a exceção lançada e a exceção que é instanciada. Isso pode ser feito pelo construtor — dois dos quatro construtores padrão permitem passar um objeto de uma exceção instanciada — ou do método initCause (). Em qualquer dos casos, o novo objeto terá referência para o objeto da exceção precedente. Assim, o encadeamento cria uma lista encadeada de exceções que pode ser varrida até o primeiro objeto. Essas referências são recuperadas pela chamada getCause ().

Trabalhando com encadeamento de exceções em Java

Apresentaremos agora um caso prático relacionado a exceções em Java, utilizando duas classes: **Principal** e **ErroValidacao**. A classe Principal implementa o comportamento interativo do programa e realiza o cálculo do fatorial de um número, impondo duas condições essenciais: a entrada deve ser um número inteiro e o cálculo deve respeitar o maior valor que uma variável long consegue acumular. Caso essas condições sejam violadas, a classe ErroValidacao modela uma exceção de validação, que pode ser gerada. Destaca-se que a classe ErroValidacao estende a classe Throwable e fornece dois construtores para definir a mensagem da exceção e associá-la diretamente ao construtor.

Assista ao vídeo e confira o código fonte que mostra a abertura de um arquivo, tratando as exceções geradas no processo de abertura por não encontrar o arquivo.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Com base no próximo código fornecido, construa as classes que recebam um número n inteiro não negativo e calcule o n-ésimo termo da série de Fibonacci. Como recordação, $F_n = F_{n-1} + F_{n-2}, F_1 = 1$ e $F_2 = 1$. Vamos ao código!

java

```
//imports
import java.util.InputMismatchException;
import java.util.Scanner;

public class Principal {
    private static Scanner entrada;
    private static long res = 0;
    private static long fat = 0;

    public static void main ( String args [ ] ) {
        long num = 0;
        entrada = new Scanner ( System.in );
        do {
            System.out.println ( "Entre com um numero inteiro ou \"-1\" para sair: " );
            try {
                num = lerEntrada ( entrada );
            } catch ( ErroValidacao erro ) {
                System.out.println ( "Entrada inválida!" );
                System.out.println ( "Causa: " + erro.getCause ( ) );
                erro.printStackTrace( System.out );
                System.exit ( -1 );
            }
            if ( num == -1 )
                System.exit ( 0 );
            else
                try {
                    System.out.println ( "O fatorial de " + num + " eh: " +
calcularFatorial( num ) );
                } catch ( ErroValidacao erro ) {
                    erro.printStackTrace( System.out );
                }
        } while ( num >= 0 );
    }
    private static long lerEntrada ( Scanner entrada ) throws ErroValidacao {
        try {
            return entrada.nextLong();
        } catch ( InputMismatchException erro_entrada ) {
            ErroValidacao erro = new ErroValidacao ( "A entrada " + entrada.next() + "
nao eh um numero inteiro!" );
            erro.atribuirCausa ( erro_entrada );
            throw erro;
        }
    }
    private static long calcularFatorial ( long num ) throws ErroValidacao {
        if ( num > 0 ) {
            res = calcularFatorial ( num - 1 );
            fat = num * res;
            if ( ( fat / res ) != num ) {
                throw new ErroValidacao ( "Overflow!");
            }
            else
                return fat;
        }
        else
            return 1;
    }
}
```

Código 7: Classe Principal.

Confira o código da Classe ErroValidacao a seguir.

```
java

public class ErroValidacao extends Throwable {
    //private String msg_erro;

    ErroValidacao ( String msg_erro ) {
        super ( msg_erro );
    }

    ErroValidacao ( String msg_erro , Throwable causa ) {
        super ( msg_erro , causa );
    }
    public void atribuirCausa ( Throwable causa ) {
        initCause ( causa );
    }

    @Override
    public String toString ( ) {
        return "ErroValidacao: " + this.getMessage();
    }
}
```

Código 8: Classe ErroValidacao.

Fique agora com o passo a passo da atividade!

1. Abra a IDE e crie a classe Principal sem o tratamento de exceções.
2. Teste o código calculando termos da sequência de Fibonacci válidos.
3. Insira o tratamento de exceções, análogo ao exemplo, que trata as exceções de entrada inválida e estouro de pilha.
4. Teste o código.

Faça você mesmo!

Um programador criou uma exceção, como mostra o trecho de código a seguir.

```
java

public class ErroValidacaoCPF extends Throwable {
    ErroValidacaoCPF ( String msg_erro ) {
        super ( msg_erro );
    }
    ErroValidacaoCPF ( String msg_erro , Throwable causa ) {
        super ( msg_erro , causa );
    }
     public void atribuiCausa (Throwable causa ) {
        initCause (causa);
    }
//Código ocultado
…
}
```

Considerando o que você conhece sobre encadeamento de exceções em Java, qual(is) da(s) opção(ões) a seguir mostra(m) um lançamento correto dessa exceção? Considere que todas as opções são sintaticamente corretas e que E_causa é um objeto do tipo Throwable.

I.

```
java

ErroValidacaoCPF eCPF = new ErroValidacaoCPF ( "Entrada invalida!" );
            eCPF.atribuiCausa(E_causa);
            throw eCPF;
```

II.

```
java

ErroValidacaoCNPJ eCNPJ = new ErroValidacaoCNPJ ( "Entrada invalida!" , null );
            eCNPJ.atribuiCausa(E_causa);
            throw eCNPJ;
```

III.

```
java

ErroValidacaoCPF eCPF = new ErroValidacaoCPF ( "Entrada invalida!"  , E_causa );
            throw eCNPJ;
```

A

I.

B

II.

C

III.

D

I e II.

E

I e III.

A alternativa E está correta.

Tanto I quanto III criam o objeto exceção fazendo o encadeamento e passando apenas um objeto Throwable como causa da exceção. No entanto, a II instancia a exceção passando null como valor do objeto Throwable causador e, na linha seguinte, tenta rescrever esse objeto. Ainda que utilize null novamente, uma vez que a exceção foi associada ao objeto do tipo Throwable causador, essa associação não pode ser modificada.

Procedimentos do tratamento de exceções em Java

Agora vamos nos concentrar nos procedimentos envolvidos no tratamento de exceções em Java, isto é, os procedimentos que o mecanismo disponibiliza para que o programador faça o tratamento de exceções.

Assista ao vídeo e fique por dentro dos conceitos de notificação, lançamento e relançamento de exceções na linguagem Java.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Uma consideração a ser feita é a suposta distinção entre a sinalização e o lançamento de exceções. Uma rápida pesquisa na internet mostra que, em português, o conceito de sinalização de exceções aparece pouco. Em inglês, praticamente inexiste.

Estudando um pouco mais o assunto, podemos ver que o uso do termo sinalização de exceção aparece com mais frequência na disciplina de linguagem de programação, que estuda não uma linguagem específica, mas os conceitos envolvidos no projeto de uma linguagem. Para essa disciplina, sinalizar uma exceção é um conceito genérico que indica o mecanismo pelo qual uma exceção é criada. Em Java, é o que se chama de lançamento de exceção. Há autores que consideram a sinalização como o mecanismo de instanciação de exceções implícitas, diferenciando-o do mecanismo que cria uma exceção explícita.

> Para evitar a confusão que a mescla de conceitos provoca, vamos definir a sinalização de uma exceção como o mecanismo pelo qual um método cria uma instância de uma exceção. Dessa forma, mantemos coerência com o conceito de linguagem de programação em relação ao mecanismo de tratamento de exceções. Portanto, sinalização e lançamento de exceção se tornam sinônimos.

Alguns profissionais chamam ao mecanismo pelo qual um método Java notifica o chamador das exceções que pode lançar como sinalização de exceções. Contudo, isso é uma abordagem predisposta à confusão. Esse mecanismo, para fins de nossa discussão, será referenciado como mecanismo de notificação de exceção.

Veremos em mais detalhes os processos que formam o tratamento de exceções em Java. Abordaremos a notificação de exceção, o seu lançamento e a forma como relançar uma exceção.

Notificando uma exceção

A notificação é o procedimento pelo qual um método avisa ao chamador das exceções que pode lançar. Ela é obrigatória se estivermos lançando uma exceção explícita e sua ausência irá gerar erro de compilação. Contudo, mesmo se lançarmos uma exceção implícita manualmente, ela não é obrigatória. Fazemos a notificação utilizando a cláusula throws ao fim da assinatura do método.

A razão para isso é simples. Vamos conferir!

Exceções explícitas

São invocadas pelo programador por uma chamada explícita que interrompe a execução do procedimento que gerou a exceção, trata a exceção por meio de um procedimento explícito e em seguida completa a execução normal.

Exceções implícitas

São invocadas implicitamente, isto é, sem interferência do programador. As entidades invocadas no tratamento de exceções são os tratadores de exceções. Essas exceções funcionam de forma similar à explícita, pois suspende o procedimento que causou a exceção. No caso das exceções implícitas, o tratador (entidade implícita não definida pelo programador) trata a exceção e duas ações diferentes podem acontecer: a continuação da unidade que causou a exceção ou o término forçado da execução dessa unidade.

Uma coisa interessante a se considerar na propagação de exceções é que ela pode ser propagada até o tratador padrão Java, mesmo no caso de ser uma exceção explícita. Basta que ela seja propagada até o método “main ()” e que este, por sua vez, faça a notificação de que pode lançar a exceção. Isso fará com que ela seja passada ao tratador padrão.

Lançando uma exceção

O lançamento de uma exceção pode ocorrer de maneira implícita ou explícita, caso em que é utilizada a instrução throw. Como já vimos, quando uma exceção é lançada, há um desvio indesejado no fluxo de execução do programa. Isso é importante, pois o lançamento de uma exceção pode se dar fora de um bloco try-catch. Nesse caso, o programador deve ter o cuidado de inserir o lançamento em fluxo alternativo, pois do contrário a instrução throw estará no fluxo principal e o compilador irá acusar erro de compilação, pois todo o código abaixo dela será inatingível. Um exemplo desse caso pode ser visto no **código 1**. Observe que uma exceção é lançada na linha 4, no fluxo principal do programa. Ao atingir a linha 4, a execução é desviada e todo o restante não pode ser alcançado.

```
java

public class Calculadora {
    public int divisao ( int dividendo , int divisor ) throws ArithmeticException ,
FileNotFoundException
    {
        throw new FileNotFoundException  ( "Arquivo nao encontrado." );
        try {
            if ( divisor == 0 )
                throw new ArithmeticException ( "Divisor nulo." );
        }
        catch (Exception e)
        {
            System.out.println( "ERRO: Divisão por zero! " + e.getMessage() );
            //throw e;
            return 999999999;
        }
        return dividendo / divisor;
    }
}
```

Código 1: Exemplo de código que não compila devido a erro no lançamento de exceção.

Situações assim podem ser resolvidas pelo aninhamento de blocos try. Quando múltiplos blocos try são aninhados, cada contexto de tratamento de interrupção é empilhado até o mais interno. Ao ocorrer uma exceção, os blocos são inspecionados em busca de um bloco catch adequado para tratar a exceção. Esse processo começa no contexto em que a exceção foi lançada e segue até o mais externo. Se nenhum bloco catch for adequado, a exceção é passada para o tratador padrão e o programa é encerrado.

O aninhamento também acontece, de maneira menos óbvia, quando há um encadeamento de chamadas a métodos em que cada método tenha definido seu próprio contexto de tratamento de exceção. Esse é exatamente o caso se combinarmos o **código 2** e o **código 3**. Veja o resultado!

```java
public class Calculadora {
    public int divisao ( int dividendo , int divisor )
    {
        try {
            if ( divisor == 0 )
                throw new ArithmeticException ( "Divisor nulo." );
        }
        catch (Exception e)
        {
            System.out.println( "ERRO: Divisão por zero! " + e.getMessage() );
            return 999999999;
        }
        return dividendo / divisor;
    }
}
```

Código 2: Classe Calculadora.

```java
public class Principal {
    public static void main ( String args [ ] ) {
        int dividendo, divisor;
        String controle = "s";

        Calculadora calc = new Calculadora ( );
        Scanner s = new Scanner ( System.in );
        do {
            System.out.println ( "Entre com o dividendo." );
            dividendo = s.nextInt();
            System.out.println ( "Entre com o divisor." );
            divisor = s.nextInt();
            try {
                System.out.println ( "O quociente é: " + calc.divisao ( dividendo ,
divisor ) );
            } catch ( ArithmeticException e ) {
                System.out.println( "ERRO: Divisão por zero! " + e.getMessage() );
            }
            System.out.println ( "Repetir?" );
            controle = s.next().toString();
        } while ( !controle.equals( "n" ) );
        s.close();
    }
}
```

Código 3: Classe Principal com contexto de tratamento de exceção definido para o método divisão ().

Veja que na linha 4 do **código 2** há um bloco try, que forma o contexto de tratamento de exceção do método (linha 2). Quando esse método é invocado na linha 14 do **código 3**, ele está dentro de outro bloco try (linha 12) que define outro contexto de tratamento de exceção.

Exceções também podem ser lançadas quando o programador estabelece pré-condições e pós-condições para o método. Trata-se de uma boa prática de programação que contribui para um código bem escrito. Vamos conhecer essas condições:

Pré-condições

São condições estabelecidas pelo programador que devem ser satisfeitas para que o método comece sua execução. Quando não são satisfeitas, o comportamento do método é indefinido. Nesse caso, uma exceção pode ser lançada para refletir essa situação e permitir o tratamento adequado ao problema.

Pós-condições

São restrições impostas ao retorno do método e a seus efeitos colaterais possíveis. Elas são verdadeiras se o método, após a sua execução e a satisfação das pré-condições, tiver levado o sistema ao estado previsto, ou seja, se os efeitos da execução daquele método correspondem ao projeto. Caso contrário, as pós-condições são falsas e o programador pode lançar exceções para identificar e tratar o problema.

Como um exemplo didático desse uso de exceções, podemos pensar num método que une (concatena) dois vetores (array) recebidos como parâmetro e retorna uma referência para o vetor resultante. Podemos estabelecer como pré-condição que nenhuma das referências para os vetores que serão unidos podem ser nulas. Se alguma delas o forem, então uma exceção NullPointerException é lançada. Não sendo nulas, há de fato dois vetores para serem concatenados. O resultado dessa concatenação não pode ultrapassar o tamanho da heap. Se o vetor for maior do que o permitido, uma exceção OutOfMemoryError é lançada. Caso contrário, o resultado é válido.

Relançando uma exceção

As situações que examinamos até o momento sempre caíram em uma de duas situações:

- As exceções lançadas eram tratadas.
- As exceções lançadas eram propagadas, podendo ser, em último caso, tratadas pelo tratador padrão da Java.

Mas há outra possibilidade: é possível que uma exceção capturada não seja tratada, ou tratada parcialmente.

Quando propagamos uma exceção lançada ao longo da cadeia de chamadores, fazemos isso até que seja encontrado um bloco catch adequado ou ela seja capturada pelo tratador padrão. Uma vez que um bloco catch captura uma exceção, ele pode decidir tratá-la, tratá-la parcialmente ou não a tratar. Nos dois últimos casos, a busca por outro bloco catch adequado deve ser reiniciada, pois como a exceção foi capturada, essa busca terminou. Felizmente é possível reiniciar o processo relançando a exceção capturada.

Relançar uma exceção permite postergar o tratamento dela ou parte dela. Assim, um novo bloco catch adequado, associado a um bloco try mais externo, será buscado. Uma vez relançada a exceção, o procedimento transcorre semelhante ao lançamento. Um bloco catch adequado é buscado e, se não encontrado, a exceção será passada para o tratador padrão de exceções da Java, que forçará o fim do programa.

O relançamento da exceção é feito pela instrução throw, dentro do bloco catch, seguida pela referência para o objeto exceção capturado. A linha 11 do **código 4** mostra o relançamento da exceção.

```java
public class Calculadora {
    public int divisao ( int dividendo , int divisor ) throws ArithmeticException
    {
        try {
            if ( divisor == 0 )
                throw new ArithmeticException ( "Divisor nulo." );
        }
        catch (Exception e)
        {
            System.out.println( "ERRO: Divisão por zero! " + e.getMessage() );
            throw e;
        }
        return dividendo / divisor;
    }
}
```

Código 4: Exemplo de relançamento de exceção na classe Calculadora.

Exceções, contudo, não podem ser relançadas de um bloco finally, pois nesse caso a referência ao objeto exceção não está disponível. Observe a linha 21 do **código 5**. O bloco finally não recebe a referência para a exceção e essa é uma variável local do bloco catch.

```java
public class Principal {
    public static void main ( String args [ ] ) throws InterruptedException {
        int divisor , dividendo , quociente = 0;
        String controle = "s";

        Scanner s = new Scanner ( System.in );
        do {
            System.out.println ( "Entre com o dividendo." );
            dividendo = s.nextInt();
            System.out.println ( "Entre com o divisor." );
            divisor = s.nextInt();
            try {
                if ( divisor ==0 )
                    throw new ArithmeticException ( "Divisor nulo." );
                quociente = dividendo / divisor;
            }
            catch (Exception e)
            {
                System.out.println( "ERRO: Divisão por zero! " + e.getMessage() );
            }
            finally
            {
                System.out.println("Bloco finally.");
            }
            System.out.println ( "O quociente é: " + quociente );
            System.out.println ( "Repetir?" );
            controle = s.next().toString();
        } while ( controle.equals( "s" ) );
        s.close();
    }
}
```

Código 5: Exemplo de uso de finallyA.

Atividade 1

Sobre o relançamento de exceções em Java, assinale a única alternativa correta.

A

O relançamento é feito com a instrução throws.

B

Blocos try aninhados dispensam o relançamento de exceção.

C

O relançamento é feito dentro de um bloco try.

D

Usa-se a instrução throw seguida de new e do tipo de exceção a ser relançada.

E

É feito dentro de um bloco catch e usando a referência da exceção capturada.

A alternativa E está correta.

O relançamento utiliza a referência da exceção capturada junto da instrução throw para lançar novamente essa exceção. Esse relançamento é feito dentro de um bloco catch.

Tratando uma exceção

Vamos refletir um pouco mais sobre o mecanismo de tratamento de exceção de Java. Até o momento falamos sobre exceções e mencionamos que elas são anormalidades — erros — experimentados pelo software durante sua execução. Não falamos, contudo, dos impactos que o tratamento de exceção produz. Vamos estudá-los agora.

Assista ao vídeo e veja como o tratamento de exceções colabora para aumentar a confiabilidade e a segurança de um software.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

A instrução catch define qual tipo de exceção aquele bloco pode tratar. Assim, quando uma exceção é lançada, uma busca é feita do bloco try local para o mais externo, até que um bloco catch adequado seja localizado. Quando isso ocorre, a exceção é capturada por aquele bloco catch que recebe como parâmetro a referência para o objeto exceção que foi lançado.

A captura de uma exceção também significa que os demais blocos catch não serão verificados. Portanto, uma vez que um bloco catch adequado seja identificado, a exceção é entregue a ele para tratamento e os demais

blocos são desprezados. Aliás, é por isso que precisamos relançar a exceção se desejarmos transferir seu tratamento para outro bloco.

Blocos catch, entretanto, não podem ocorrer de maneira independente no código. Eles precisam estar associados a um bloco try. Eles também não podem ser aninhados, como os blocos try. Então como lidar com o **código 6**? Confira!

```
java

public char[] separa ( char[] arrj , int tamnh ) {
    int partes;
    partes = arrj.length / tamnh;
    if ( partes < 1 )
    char[] prim_parte = new char [tamnh];
    System.arraycopy ( arrj , 0 , prim_parte , 0 , tamnh );
    return prim_parte;
}
```

Código 6: Possibilidade de lançamento de tipos distintos de exceções.

Observe que se tamnh for zero, a linha 3 irá gerar uma exceção ArithmeticException (divisão por zero); já se tamnh for maior do que o tamanho de arrj, a linha 6 irá gerar uma exceção ArrayIndexOutOfBoundsException. Esse é um exemplo de um trecho de código que pode lançar mais de um tipo de exceção.

Uma solução é aninhar os blocos try, porém a linguagem Java nos dá uma solução mais elegante: podemos empregar múltiplas cláusulas catch, como mostra o **código 7**.

```
java

public char[] separa ( char[] arrj , int tamnh ) {
    int partes;
    try {
        partes = arrj.length / tamnh;
        if ( partes < 1 )
        {
            char[] prim_parte = new char [tamnh];
            System.arraycopy ( arrj , 0 , prim_parte , 0 , tamnh );
            return prim_parte;
        }
    } catch (ArithmeticException e) {
        System.out.println ( "Divisão por zero: " + e );
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println ( "Indice fora dos limites: " + e );
    }
        return null;
}
```

Código 7: Múltiplas cláusulas catch.

Um cuidado é necessário: as exceções de subclasses devem vir antes das exceções da respectiva superclasse. O bloco catch irá capturar as exceções que correspondam à classe listada e a todas as suas subclasses. Isso se dá porque um objeto de uma subclasse é também um tipo da superclasse. Uma vez que a exceção seja capturada, as demais cláusulas catch não são verificadas.

Atividade 2

Um programador criou uma exceção (NotReferencedException) a partir da classe RunTimeException. Considere que as exceções são empregadas em múltiplas cláusulas catch associadas ao mesmo bloco try

para lidar com o lançamento de tipos diferentes de exceção pelo mesmo trecho de código. Sobre essa situação, marque a única alternativa correta.

A

RunTimeException deve vir antes de NotReferencedException.

B

É indiferente a ordem em que as exceções são usadas.

C

NotReferencedException deve vir antes de RunTimeException.

D

Como NotReferencedException é uma exceção implícita, ela não pode ser usada dessa forma.

E

Exceções implícitas não são capturadas quando há mais de uma cláusula catch associada ao mesmo bloco try.

A alternativa C está correta.

A classe NotReferencedException é uma subclasse de RunTimeException. Logo, se RunTimeException estiver antes de NotReferencedException, a exceção será capturada por esse bloco catch e nunca chegará ao bloco destinado a tratar exceções do tipo NotReferencedException.

Explorando o mecanismo de tratamento de exceções em Java

O caso prático que usaremos é um pouco mais complexo, embora tenha apenas três classes. Vamos implementar uma operação sobre arrays, a concatenação de dois arrays. Será apresentada a implantação da exceção que trata a falta de memória para concatenação. Em seguida, você irá implementar a exceção que trata a manipulação de um array não alocado.

Assista ao vídeo e veja o código fonte de um programa em Java que implementa os principais conceitos estudados aqui, com destaque para o lançamento e o encadeamento de exceções.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Com base nas classes usadas no exemplo, modifique o código fonte para que seja lançada uma exceção caso um dos parâmetros seja um vetor nulo. Considere o próximo código!

```java
public class ErroValidacao extends Throwable {
    ErroValidacao ( String msg_erro ) {
        super ( msg_erro );
    }
    ErroValidacao ( String msg_erro , Throwable causa ) {
        super ( msg_erro , causa );
    }
    public void atribuirCausa ( Throwable causa ) {
        initCause ( causa );
    }
    @Override
    public String toString ( ) {
        return "ErroValidacao: " + this.getMessage();
    }
}
```

Código 8: Classe ErroValidacao.

A seguir, o código da classe principal.

```java
public class Principal {
    public static void main ( String args [ ] ) {
        OperacaoArray calc = new OperacaoArray ();
        char[] op1 = null;
        char[] op2 = null;
        try {
            op1 = new char [Short.MAX_VALUE];
            op2 = new char [Short.MAX_VALUE];
        } catch ( OutOfMemoryError e ) {
            Runtime runtime = Runtime.getRuntime ();
            System.out.println ( "Memoria insuficiente!" );
            System.out.println ( "A memória total da MVJ eh " + runtime.totalMemory() + "
e o máximo eh " + runtime.maxMemory () );
            System.out.println ( "Reconfigure a MVJ usando o parametro -Xmx. Você precisa
de " + 16*Short.MAX_VALUE + " soh para os vetores.");
            System.exit ( -1 );
        }

            calc.concatenarArray ( op1 , op2 );

        }
    }
}
```

Código 9: Classe Principal.

Por fim, observe a classe OperacaoArray.

```java
public class OperacaoArray {

    public char[] concatenarArray ( char[] op1 , char[] op2 ) throws ErroValidacao {
        int tamnh_res;

        tamnh_res = op1.length + op2.length;

        return copiarArray ( op1 , op2 , tamnh_res , op2.length );

    }
    private char[] copiarArray ( char[] op1 , char[] op2 , int tamnh_res , int n ){
        char[] resultado = new char [ tamnh_res ];
        System.arraycopy ( op1 , 0 , resultado , 0 , op1.length );
        System.arraycopy ( op2 , 0 , resultado , op1.length , n );
        return resultado;
    }
}
```

Código 10: Classe OperacaoArray.

Veja agora o roteiro da prática com os passos a serem seguidos.

1. Na IDE, copie as classes apresentadas criando um novo projeto.
2. Teste o código executando-o.
3. Altere o método concatenarArray de forma que uma exceção seja gerada caso um dos parâmetros for um array não alocado.
4. Teste o código.

Faça você mesmo!

Um programador implementou uma rotina de cópia de vetores do tipo char. A fim de dar mais qualidade ao seu código, ele empregou o mecanismo de tratamento de exceções de Java. Ele então fez o código de teste mostrado a seguir, junto de uma exceção definida por ele.

java

```
public class Principal {
    public static void main ( String args [ ] ) {
        char[] op1 = { 'J' , 'A' , 'V' , 'A' , '.' };
        char[] op2 = new char [4];
        System.out.println ( copiarArray ( op1 , op2 ) );
    }
    private static char[] copiarArray ( char[] op1 , char[] op2 ) { //copia o vetor op1
para op2
        try {
            if ( verificarOperandos ( op1 , op2 ) && verificarTamanhoOperandos ( op1 ,
op2 ) ) {
                System.arraycopy ( op1 , 0 , op2 , 0 , op1.length );
                return op2;
            } else
                System.out.println( "A operacao nao pode ser realizada!" );
        } catch ( NullPointerException e ) {
            System.out.println ( "Ponteiro para objeto nulo!" );
            System.out.println ( e.getMessage() );
            System.exit ( -1 );
        } catch ( ArrayIndexOutOfBoundsException e ) {
            System.out.println ( "Tentativa de extrapolar o limite do vetor!" );
            System.out.println ( e.getMessage() );
            System.exit ( -1 );
        }
        return null;
    }
    private static boolean verificarOperandos ( char[] op1 , char[] op2 ) throws
ErroOperando , NullPointerException {
        boolean check = false;
        if ( ( op1 == null ) && ( op2 == null ) )
            throw new ErroOperando ( "Ambos operandos sao nulos!" );
        else if ( op1 == null )
            throw new ErroOperando ( "Primeiro operando eh nulo!" );
        else if ( op2 == null )
            throw new ErroOperando ( "Segundo operando eh nulo!" );
        else
            check = true;
        return check;
    }
    private static boolean verificarTamanhoOperandos ( char[] op1 , char[] op2 ) {
        if ( op1.length > op2.length ) {
            System.out.println ( "Os tamanhos dos vetores são incompativeis! ");
            throw new ErroOperando ( new ArrayIndexOutOfBoundsException ( "ESPACO
INSUFICIENTE NO SEGUNDO OPERANDO!" ) );
        } else
            return true;
    }
}
```

Classe principal

```java
public class ErroOperando extends NullPointerException {
    ErroOperando ( String msg_erro ) {
        super ( msg_erro );
    }
    ErroOperando ( Throwable causa ) {
        initCause ( causa );
    }
    @Override
    public String toString ( ) {
        return "Operando nulo: " + this.getMessage();
    }
}
```

Exceção definida pelo operador.

Ao rodar o programa para testar seu código, ele observou que o programa não imprime a mensagem “ESPACO INSUFICIENTE NO SEGUNDO OPERANDO!”. Entretanto, o programador deseja que essa mensagem seja impressa quando o segundo vetor for menor do que o primeiro. Assinale a única opção que mostra a alteração do código capaz de provocar o comportamento que o operador deseja.

A

Substituir as linhas 16 e 17 da classe Principal pela linha “throw (ArrayIndexOutOfBoundsException) e.getCause();”.

B

Inverter os blocos catch das linhas 14 e 18 da classe Principal.

C

Acrescentar a instrução “throws ArrayIndexOutOfBoundsException” na assinatura do método verificarTamanhoOperandos () (linha 37 da classe Principal).

D

Acrescentar a instrução “throws ArrayIndexOutOfBoundsException” na assinatura do método copiarArray () (linha 7 da classe Principal).

E

Substituir as linhas 16 e 17 da classe Principal pela linha “throw (ArrayIndexOutOfBoundsException) e.getCause();”.

A alternativa A está correta.

Na linha 40 da classe Principal, vê-se que uma exceção do tipo ArrayIndexOutOfBoundsException é instanciada e colocada como causa da exceção ErroValidacao que é lançada. Vendo a linha 1 da classe ErroValidacao, percebe-se que ela herda de NullPointerException. Assim, a exceção lançada na linha 40 é capturada pela instrução catch da linha 14 (ErroValidacao é um subtipo de NullPointerException).

Lembremos que essa exceção tem atrelada como causa uma exceção do tipo ArrayIndexOutOfBoundsException.

Logo, a instrução e.getCause() retorna o objeto do tipo ArrayIndexOutOfBoundsException definido como causador. Para que o compilador saiba o tipo de exceção que estamos relançando, faz-se um cast explícito “(ArrayIndexOutOfBoundsException)” que leva o compilador a interpretar corretamente aquele objeto como sendo do tipo ArrayIndexOutOfBoundsException. Assim, conseguimos relançar essa exceção que, capturada pelo tratador default do Java, imprime a mensagem almejada.

Considerações finais

O que você aprendeu neste conteúdo?

- Conceitos básicos de tratamento de exceções em Java.
- Os tipos de exceções implícitas e explícitas de Java.
- A forma como declarar novos tipos de exceções em Java (exceções definidas pelo programador).
- Os comandos do mecanismo de tratamento de exceções de Java.
- O mecanismo de encadeamento de exceções.
- Os processos de notificação, lançamento e relançamento de exceções.
- A forma como empregar o tratamento de exceções.

Explore +

O mecanismo de tratamento de exceções possui muitas nuances interessantes que são úteis para o desenvolvimento de software de qualidade. O encadeamento de exceções é uma das características que podem ajudar, sobretudo no desenvolvimento de bibliotecas, motores e componentes que manipulam recursos. Por isso, estudar a documentação Java sobre o assunto e conhecer melhor a classe throwable certamente contribuirá para o seu aprendizado.

Referências

DEITEL, P.; DEITEL, H. **Java - How to program**. 11th. ed. [*S.l.*]: Pearson, 2017.

ORACLE AMERICA INC. **What is an exception?** Oracle Java Documentation, s. d. Consultado na internet em: 10 abr. 2023.

ORACLE INC. **Programming with exceptions**. Oracle Help Center, s. d. Consultado na internet em: 27 abr. 2023.

SEUNGIL, L. *et al*. **Efficient Java exception handling in just-in-time compilation**. JAVA '00, 2000.