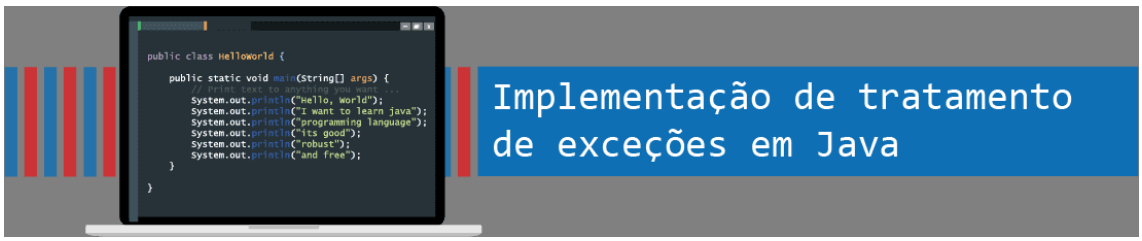


DESCRIÇÃO

Exceções em Java: abordagem de maneira prática dos tipos de exceções e da classe Exception.

Sinalização, lançamento, relançamento e tratamento de exceções.

PROPÓSITO

O tratamento de exceções é um importante recurso que permite criar programas tolerantes a falhas. Trata-se de um mecanismo que permite resolver ou ao menos lidar com exceções, muitas vezes evitando que a execução do software seja interrompida.

PREPARAÇÃO

Para melhor absorção do conhecimento, recomenda-se o uso de computador com o Java Development Kit (JDK) e um IDE (*Integrated Development Environment*) instalados.

OBJETIVOS

MÓDULO 1

Descrever os tipos de exceções em Java

MÓDULO 2

Descrever a classe Exception de Java

MÓDULO 3

Aplicar o mecanismo de tratamento de exceções que Java disponibiliza

INTRODUÇÃO

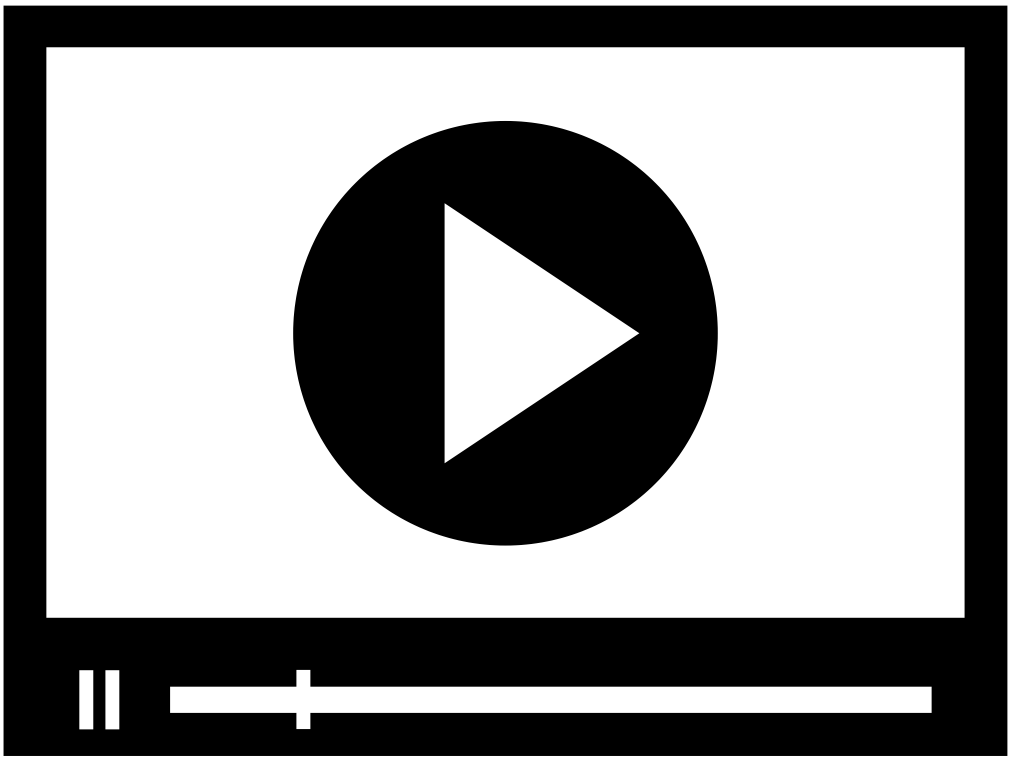
A documentação oficial da linguagem Java explica que o termo exceção é uma abreviatura para a frase **evento excepcional** e o define como “um evento, o qual ocorre durante a execução de um programa, que interrompe o fluxo normal das instruções do programa” (ORACLE AMERICA INC., 2021).

A definição de exceção em software apresentada por Java não é específica da linguagem. Sempre que um evento anormal causa a interrupção no fluxo normal das instruções de um software, há uma exceção. Porém, nem todas as linguagens oferecem mecanismos para lidar com tais problemas. Outras oferecem 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, que buscará lançar as bases para que o futuro profissional de programação seja capaz de explorar os recursos da linguagem Java e produzir softwares de qualidade.

MÓDULO 1

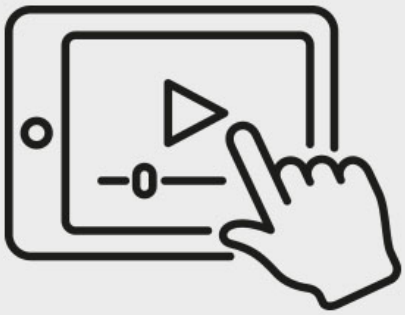
- **Descrever os tipos de exceções em Java**



TRATAMENTO DE EXCEÇÕES EM JAVA

Iniciamos este módulo com uma introdução ao tratamento de exceções em Java. Além disso, apresentamos os tipos de exceções e como criar novas exceções. Assista ao vídeo a seguir.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



CONCEITOS

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, por exemplo. 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.

Esse objeto é entregue para o sistema de execução da Máquina Virtual Java (MVJ), processo esse chamado de lançamento de exceção.

Uma vez que a exceção é lançada por um método, o sistema de execução vai procurar na pilha de chamadas (*call stack*) por 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 tratar. 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.

📣 ATENÇÃO

Ao fornecer um código capaz de lidar com a exceção ocorrida, o programador tem a possibilidade de evitar que o programa seja interrompido. Todavia, se nenhum tratador de exceção apropriado for localizado pelo sistema de execução da MVJ, o programa irá terminar.

Um bloco de exceção tem a forma geral mostrada no Código 1, a seguir.

```
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”
}
```

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

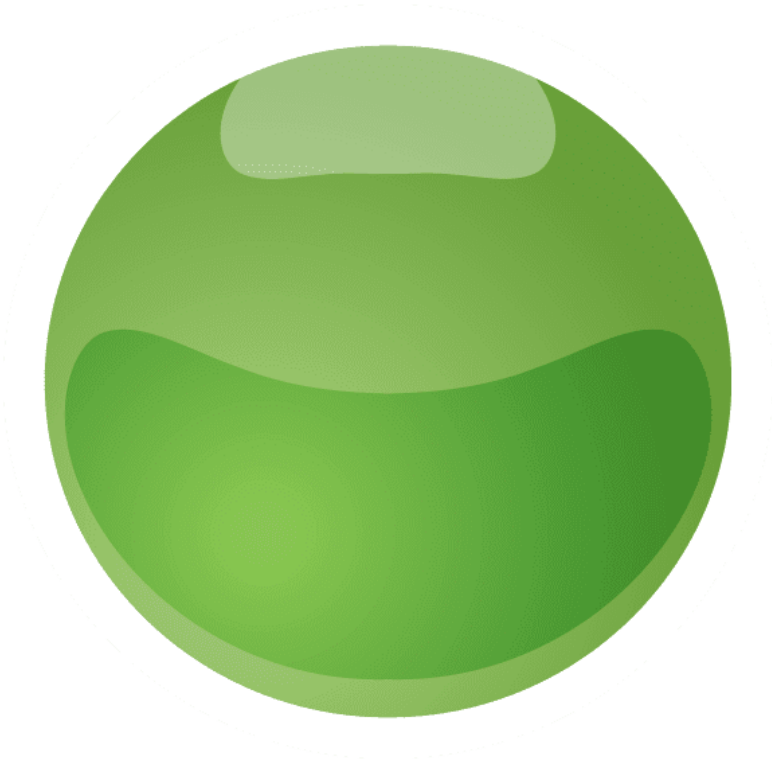


Imagem: Shutterstock.com

A separação do código destinado ao tratamento de erros do código funcional do software. Isso melhora a organização e contribui para facilitar a depuração de problemas.

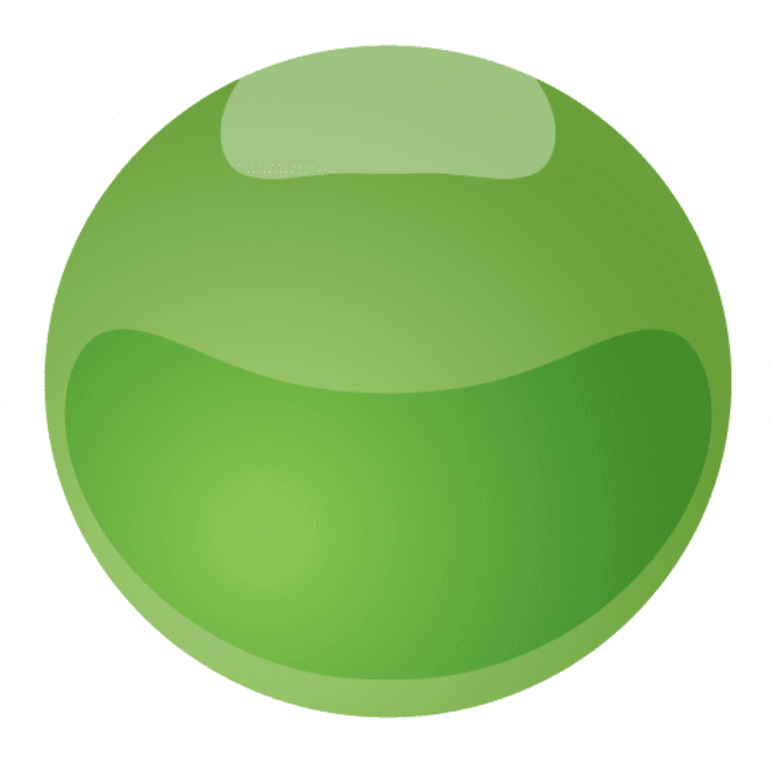


Imagem: Shutterstock.com

A possibilidade de se 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.

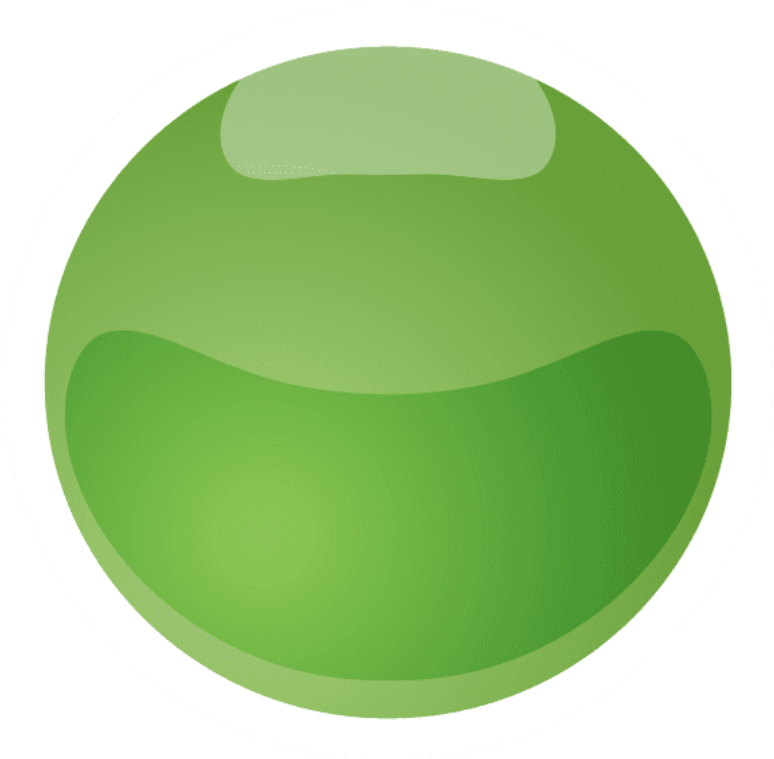


Imagem: Shutterstock.com

A possibilidade de 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.

TIPOS DE EXCEÇÕES

Todos os tipos de exceção em Java são subclasses da classe Throwable. A primeira separação feita agrupa as exceções em dois subtipos:

ERROR

Agrupa as exceções que, em situações normais, não precisam ser capturadas pelo programa. Em situações normais, não se espera que o programa cause esse tipo de erro, mas ele pode ocorrer e, quando ocorre, causa uma falha catastrófica.

A subclasse Error agrupa erros que ocorrem em **tempo de execução**, ela é utilizada para o

sistema de execução Java indicar erros relacionados ao ambiente de execução. Um estouro de pilha (*stack overflow*) é um exemplo de exceção que pertence à subclasse Error.

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.

TEMPO DE EXECUÇÃO

Trata-se do período em que um software permanece em execução.

A figura a seguir mostra algumas subclasses da classe Throwable.

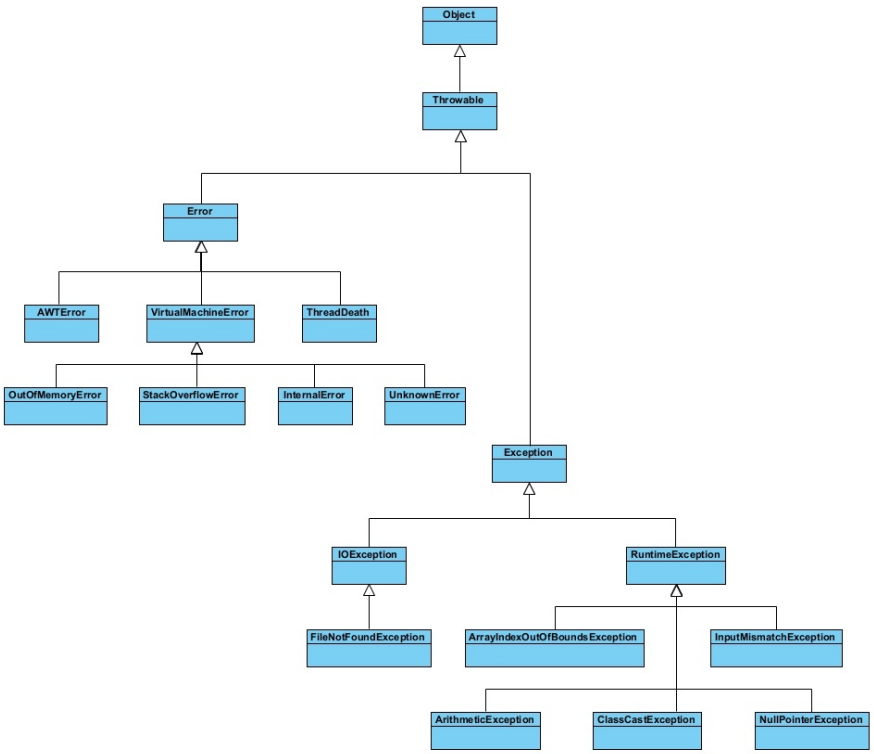


Imagem: Marlos de Mendonça Corrêa.

📷 Hierarquia parcial de classes de exceções em Java.

Quando uma exceção não é adequadamente capturada e tratada, o interpretador irá mostrar uma mensagem de erro com informações sobre o problema ocorrido e encerrará o programa.

Vejamos o exemplo mostrado no Código 2.

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

```
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();
}
}
```

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, como podemos observar a seguir.

```
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

Vendo a linha número 42, observamos a mensagem “(...) *Process exited with an error* (…)”, que indica o término abrupto do programa. Vamos, agora, observar como a captura e o tratamento da exceção beneficia o programa. Vamos substituir a linha 12 do Código 2 pelas instruções mostradas no Código 3.

Código 3: Envolvendo a divisão num bloco try-catch.

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

Eis a saída para o código modificado:

```
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
------------------------------------------------------------------------
```

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*, conforme vemos na linha 5. A linha 6 mostrou o valor de “quociente” como 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.

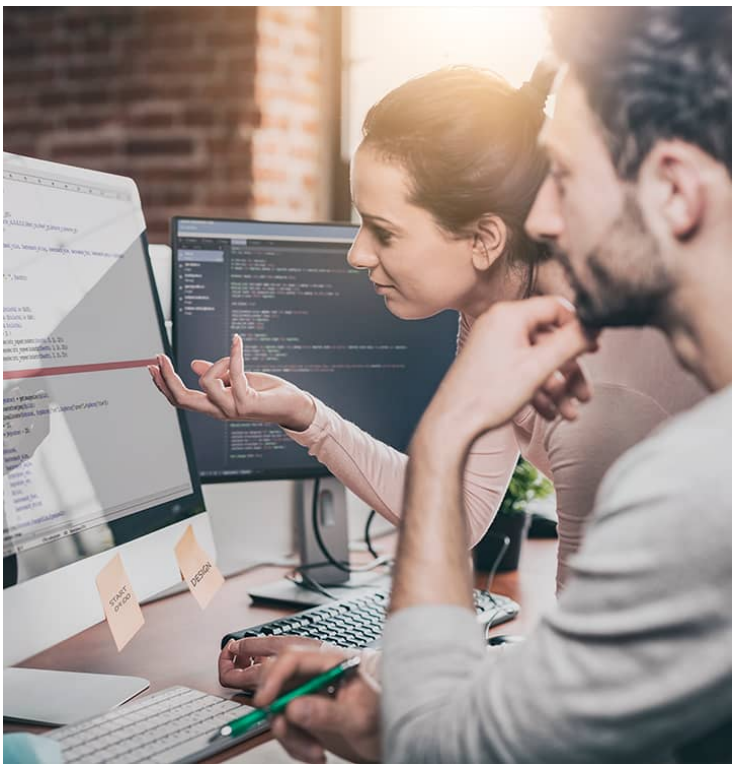


Foto: Shutterstock.com

AGORA QUE JÁ COMPREENDEMOS O ESSENCIAL DO TRATAMENTO DE ERROS EM JAVA, VEREMOS A SEGUIR OS TIPOS DE EXCEÇÕES IMPLÍCITAS, EXPLÍCITAS E THROWABLES.

EXCEÇÕES IMPLÍCITAS

Exceções implícitas são aquelas definidas nos subtipos Error e RuntimeException e suas derivadas. Trata-se de exceções ubíquas, quer dizer, 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.

★ EXEMPLO

Se nos remetermos ao 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 irá produzir a exceção a menos que o usuário, durante a execução, 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.

📢 ATENÇÃO

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 3 nos mostra um bloco *try-catch*. O bloco *try* indica o bloco de código que será monitorado para ocorrência de exceção e o bloco catch captura e trata essa exceção. Mas mesmo nessa versão modificada, o programador não está lançando a exceção, apenas capturando-a. A exceção continua sendo lançada automaticamente pelo Java runtime. Cabe

dizer, contudo, que não há impedimento a que o programador lance manualmente a exceção. Veja o Código 4, que modifica o Código 3, passando a lançar a exceção.

Código 4: Lançando manualmente uma exceção implícita.

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

A execução dessa versão modificada produz como saída:

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

DICA

Embora seja possível ao programador lançar manualmente a exceção, os chamadores do método que a lançam não serão forçados 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, pois podemos simplesmente ignorar seu tratamento. Isso, porém, não tende a ser uma boa ideia, já que elas serão lançadas e terminarão por 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 dito incontornável.

📢 ATENÇÃO

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 onde 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 que o método irá lançar se um erro ocorrer. Nesse caso, os chamadores precisarão envolver a chamada em um bloco *try-catch*.

O Código 5 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.

Código 5: Exemplo de exceção explícita.

```
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];
}
}
```

Como é uma exceção explícita, ela precisa ser tratada em algum momento. No caso mostrado no Código 5, ela está abarcada por um bloco try-catch, o que livra os métodos chamadores de terem de lidar com a exceção gerada.

E quais seriam as consequências se o programador optasse por não a tratar 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, lembremos, é uma exceção explícita. Ao fazer isso, o interpretador Java obrigaria a que uma cláusula throws fosse acrescentada na assinatura do método, informando aos chamadores as exceções que o método pode lançar.

O Código 5 pode ser invocado simplesmente fazendo-se como no Código 6. Observe que a invocação, vista na linha 5, prescinde do uso de bloco try-catch.

Código 6: Invocando um método que lança uma exceção explícita.

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

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

SAIBA MAIS

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.

DECLARANDO NOVOS TIPOS DE EXCEÇÕES

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

DICA

Se você for 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 com o uso do mecanismo de herança.

Para criar uma classe de exceção deve-se, obrigatoriamente, estender-se 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 a seguir (DEITEL; DEITEL, [s.d.]):

Um que não recebe argumentos e passa uma String – mensagem de erro padrão – para o construtor da superclasse.

Um que recebe uma String – mensagem de erro personalizada – e a passa para o construtor da superclasse.

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.

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. Numa 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.

DICA

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

O Código 7 mostra um exemplo de exceção criada para indicar problemas na validação do CNPJ. Repare seu uso nas linhas 22 e 61 da classe Juridica (Código 8) e logo após a saída provocada por uma exceção. Veja também que a exceção é lançada e encadeada, ficando o tratamento a cargo de quem invoca atualizarID ().

Código 7: Nova classe de exceção.

```
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 8: Classe Jurídica com lançamento de exceção.

```
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
{
throw new ErroValidacaoCNPJ ( "DV inválido." );
```

```
//return false;
}
} catch (InputMismatchException erro) {
return false;
}
}
public String retornaTipo ( ) {
return "Juridica";
}
}
```

Saída.

ErroValidacaoCNPJ: Entrada invalida!

BUILD SUCCESS

VERIFICANDO O APRENDIZADO

1. CONSIDERE AS AFIRMAÇÕES A SEGUIR.

I – UM DESVIO NO FLUXO PRINCIPAL DE UM PROGRAMA É UMA EXCEÇÃO.
II – SE UMA EXCEÇÃO É LANÇADA PELA INSTRUÇÃO “THROW”, ENTÃO ELA É UMA EXCEÇÃO EXPLÍCITA.
III – UMA EXCEÇÃO EXPLÍCITA NÃO PRECISA SER TRATADA LOCALMENTE NO MÉTODO NO QUAL É LANÇADA.

É (SÃO) VERDADEIRA(S) APENAS A:

A) I

B) II

C) III

D) I e II

E) II e III

2. SOBRE EXCEÇÕES IMPLÍCITAS, É CORRETO AFIRMAR-SE APENAS QUE:

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) Uma exceção implícita é sempre consequência direta da programação.

E) Exceções implícitas ocorrem sempre em tempo de compilação.

GABARITO

1. Considere as afirmações a seguir.

I – Um desvio no fluxo principal de um programa é uma exceção.
II – Se uma exceção é lançada pela instrução “throw”, então ela é uma exceção explícita.
III – Uma exceção explícita não precisa ser tratada localmente no método no qual é lançada.

É (são) verdadeira(s) apenas a:

A alternativa **"C "** está correta.

Uma exceção é um desvio não previsto no fluxo do programa. Desvios no fluxo principal para fluxos alternativos são condições normais e previstas e não são exceções. Além disso, uma exceção implícita também pode ser lançada pela instrução “throw”. Mesmo uma instrução explícita pode ser lançada e propagada, não sendo obrigatório seu tratamento no método onde ocorre.

2. Sobre exceções implícitas, é correto afirmar-se apenas que:

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.

MÓDULO 2

Descrever a classe Exception de Java

CONCEITOS

O tratamento de exceções em Java é um mecanismo flexível que agrega ao programador um importante recurso. Para extrair tudo que ele tem a oferecer, faz-se necessário conhecer mais detalhadamente o seu funcionamento. Já dissemos que as exceções provocam um desvio na lógica de execução do programa. Mas esse desvio pode trazer consequências indesejadas. A linguagem, porém, oferece uma maneira de remediar tal situação.

Imagem: Shutterstock.com

Três cláusulas se destacam quando se busca compreender o mecanismo de tratamento de exceções de Java:

FINALLY

A instrução finally oferece uma maneira de lidar com certos problemas gerados pelo desvio indesejado no fluxo do programa.

THROW

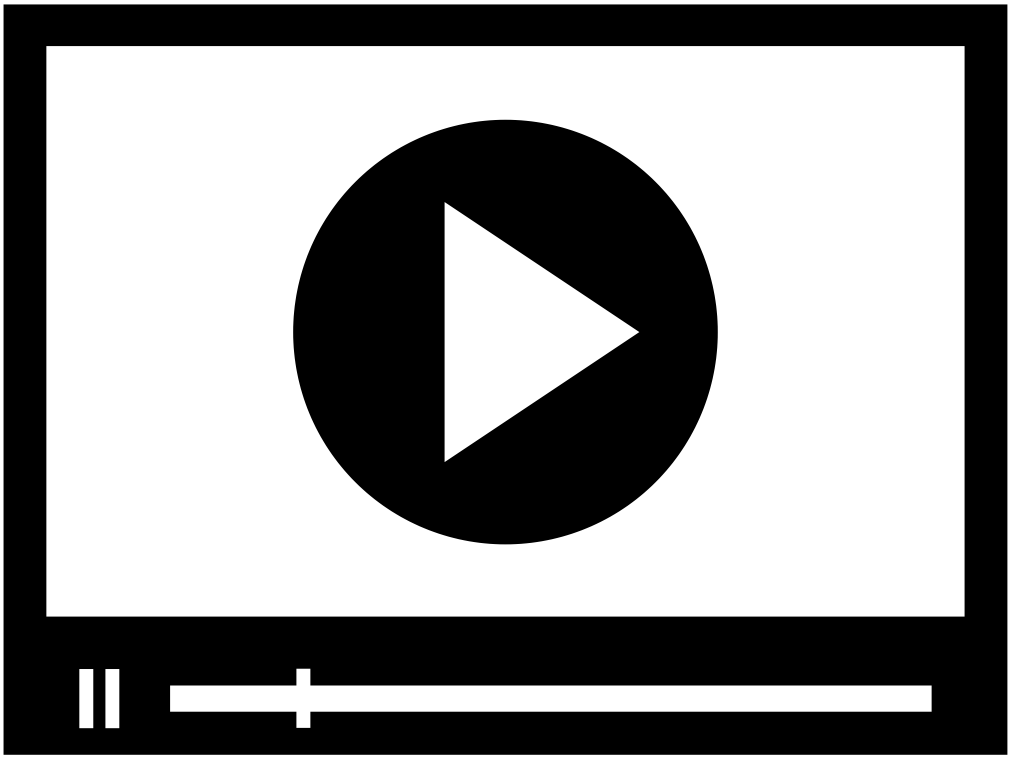
A instrução throw é básica, por meio dela podemos lançar as exceções explícitas e, manualmente, as implícitas.

THROWS

A instrução throws pode ser conjugada para alterar a abordagem no tratamento das exceções lançadas.

Nas seções a seguir, abordaremos cada uma dessas instruções e, na última seção, analisaremos um mecanismo adicionado a partir da versão 1.4 da Java 2 chamado Encadeamento de Exceção.

TIPOS DE COMANDO



OS COMANDOS FINALLY, THROW E THROWS

Acompanhe os comandos finally, throw e throws e o encadeamento de exceções apresentados no vídeo a seguir.

COMANDO FINALLY

★ EXEMPLO

Imagine que um programador está manipulando um grande volume de dados. Para isso, ele está usando 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?

Em linguagens como o C++, que não implementam um coletor de lixo, a porção de memória reservada para uso permanecerá alocada e indisponível até que o sistema seja reiniciado. Como a Máquina Virtual Java implementa o coletor de lixo, o mesmo não ocorre. A porção de memória permanecerá alocada somente até a próxima execução do coletor. Entretanto, vazamentos ainda podem ocorrer se o programador, equivocadamente, 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. Podemos ter outros vazamentos, como conexões não encerradas, acessos a arquivos não fechados, dispositivos não liberados etc. Infelizmente, nesses casos, o coletor de lixo não resolve a questão.

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 9 mostra um exemplo de uso de finally na linha 21.

Código 9: Exemplo de uso de finally.

```
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();
}
}
```

Vamos observar a saída.

```
--- 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
------------------------------------------------------------------------
```

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 finally. Colocando de outra forma, 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 9, mas mantivéssemos o bloco finally, o programa compilaria sem problema.

RELEMBRANDO

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, sempre, 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. Tais elementos não podem ser usados como exceção. Uma instância throwable pode ser obtida por meio da passagem de parâmetro na cláusula catch ou com o uso do operador new.

O efeito quando o compilador encontra uma cláusula throw é de desviar a execução do programa. Isso significa que nenhuma instrução existente após a cláusula será executada. Nesse ponto:

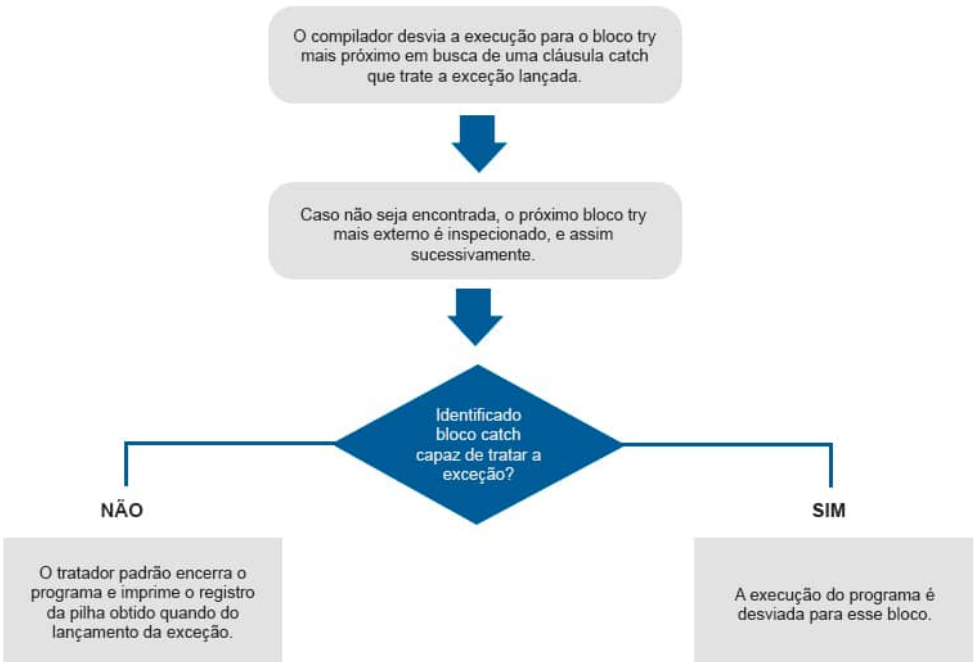


Imagem: Haline Santiago

Vamos modificar o Código 9 movendo a lógica do programa para a classe Calculadora, conforme mostrado no Código 10 e Código 11.

Código 10: Classe Calculadora.

```
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 11: Classe Principal modificada.

```
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();
}
}
```

Nessa versão modificada, a exceção é lançada na linha 6 do Código 10. É possível ver que a criação da exceção se dá por meio do 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 no caso está na linha 4. Esse bloco define um contexto de exceção que, como vemos na linha 8 (bloco catch correspondente), é capaz de tratar a exceção lançada. 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.

★ EXEMPLO

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.

Uma outra forma de lidar com isso seria definir um contexto de exceção no chamador (linha 13 do Código 11) e propagar a exceção para que fosse tratada externamente. O mecanismo de propagação de exceções é justamente o que veremos na próxima seção.

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.

📢 ATENÇÃO

Mas quando um método pode lançar uma exceção e não a trata, ele deve informar isso aos seus chamadores explicitamente. Isso permite 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. Não listar tais exceções irá 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 { …}

Observe que, 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 mostrado no Código 12. 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. Em consequência, precisamos ajustar a classe Principal como mostrado no Código 13. É possível ver, como sugerimos no final da seção anterior, a definição de um bloco try-catch (linha 13) para capturar e tratar a exceção lançada pelo método “divisão ()”.

Código 12: Classe Calculadora com método "divisão" modificado para propagar a exceção.

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

Código 13: Classe Principal com contexto de tratamento de exceção definido para o método "divisão ()".

```
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();
}
}
```

ENCADEAMENTO DE EXCEÇÕES

Pode acontecer de, ao responder a uma exceção, um método lançar 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.

Ocorre que se uma exceção for lançada durante o tratamento de outra, as informações como o registro da pilha de execução e o tipo de exceção da exceção anterior são perdidas. Nas primeiras versões, a linguagem Java não provia um método capaz de manter essas informações, agregando as pertencentes à nova exceção. A consequência dessa limitação é 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 provêm um mecanismo chamado **Encadeamento de Exceção**. 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.

O encadeamento de exceção permite associar uma exceção com a exceção corrente, de maneira que a exceção que ocorreu no contexto atual da execução pode registrar a exceção que lhe deu causa. Além da utilização dos construtores que admitem um objeto Throwable como parâmetro, isso também é feito utilizando dois outros métodos da classe Throwable:

GETCAUSE ()

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

INITCAUSE ()

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

Assim, utilizando o método **initCause ()**, cria-se a associação entre uma exceção e outra que lhe originou após a sua criação. Em outras palavras, permite fazer a mesma associação que o construtor, mas após a exceção ter sido criada.

📢 ATENÇÃO

Após uma exceção ter sido associada a outra, não é possível modificar tal associação. Logo, não se pode invocar mais de uma vez o método initCause () e nem o chamar se o construtor já tiver sido empregado para criar a associação.

VERIFICANDO O APRENDIZADO

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

2. SOBRE O COMANDO THROW, SÃO FEITAS AS SEGUINTES AFIRMAÇÕES:

I – NÃO PODE SER USADO COM O COMANDO THROWS.
II – SÓ PODE SER UTILIZADO DENTRO DO BLOCO TRY OU DO BLOCO CATCH.
III – PODE SER UTILIZADO PARA LANÇAR EXCEÇÕES DEFINIDAS NAS SUBCLASSES DE ERROR.

É CORRETO APENAS O QUE SE AFIRMA EM:

A) I

B) II

C) III

D) I e II

E) II e III

GABARITO

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 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.

2. Sobre o comando throw, são feitas as seguintes afirmações:

I – Não pode ser usado com o comando throws.
II – Só pode ser utilizado dentro do bloco try ou do bloco catch.
III – Pode ser utilizado para lançar exceções definidas nas subclasses de Error.

É correto apenas o que se afirma em:

A alternativa **"C "** está correta.

O comando throw pode ser usado para lançar exceções fora do bloco try-catch e pode ser usado em combinação com throws, cujo objetivo é notificar o chamador de que uma exceção pode ser lançada, mas não será tratada no local. As exceções derivadas de Error e suas subclasses são implícitas, mas throw pode ser usado para lançá-las manualmente.

MÓDULO 3

- **Aplicar o mecanismo de tratamento de exceções que Java disponibiliza**

CONCEITOS

Vamos agora refletir um pouco mais sobre o mecanismo de tratamentos de exceção de Java. Até o momento, temos falado 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.

Certamente os erros causados quando se tenta ler além do fim de um array, quando ocorre uma divisão por zero ou mesmo uma tentativa falha de abrir um arquivo que já não existe mais geram problemas que precisam ser enfrentados. Mas, para todos esses casos, técnicas tradicionais podem ser empregadas.

Imagem: Shutterstock.com

Um ponto que também deve ser considerado é o impacto que o tratamento de exceções traz ao desempenho do software. Num artigo intitulado *Efficient Java Exception Handling in Just-in-Time Compilation* (SEUNGIL *et al.*, 2000), os autores argumentam que o tratamento de exceções impede o compilador Java de fazer algumas otimizações, impactando o desempenho do programa.

DICA

A lição aqui é que tudo possui prós e contras que devem ser pesados segundo a situação específica que se está enfrentando. Não se questiona que o mecanismo de tratamento de exceções possua diversas vantagens. Mas em situações nas quais o desempenho seja crítico, deve-se refletir se seu uso é a abordagem mais adequada.

Outra 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.

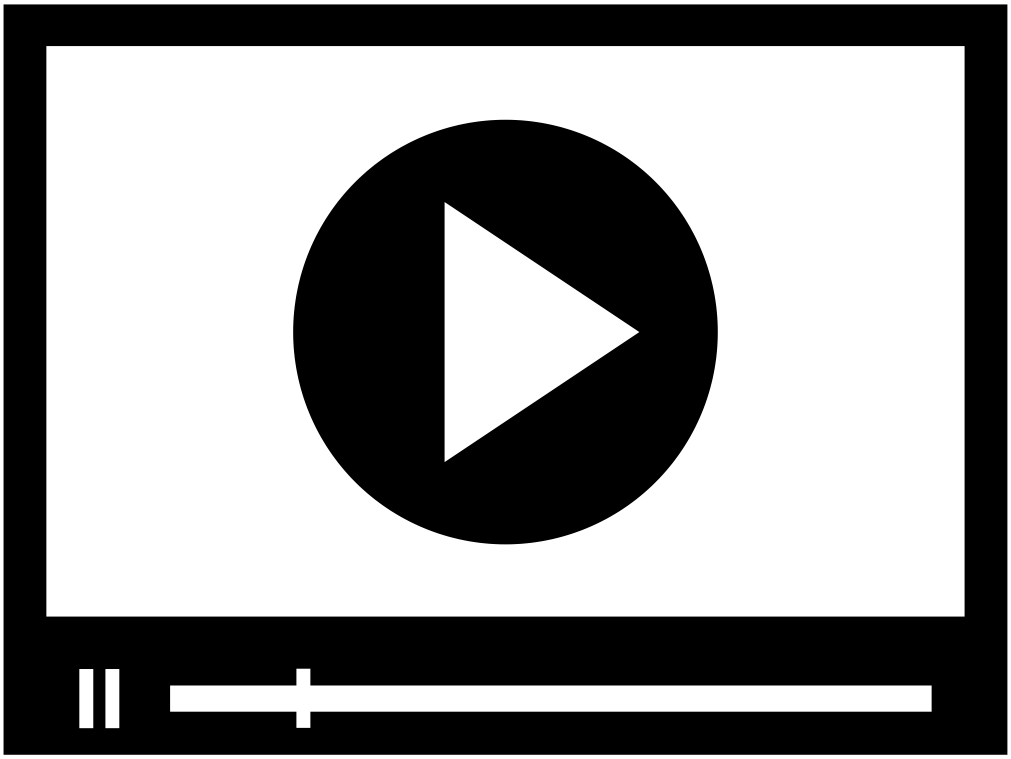
Para a disciplina de Linguagem de Programação, sinalizar uma exceção é um conceito genérico que indica o mecanismo pelo qual uma exceção é criada. Em Java, é o que se chama de lançamento de exceção. Outros autores consideram que a sinalização é o mecanismo de instanciação de exceções implícitas, diferenciando-o do mecanismo que cria uma exceção explícita.

PARA EVITAR A CONFUSÃO QUE ESSA MESCLA DE CONCEITOS PROVOCA, VAMOS DEFINIR A SINALIZAÇÃO DE UMA EXCEÇÃO COMO O MECANISMO PELO QUAL UM MÉTODO CRIA UMA INSTÂNCIA DE UMA EXCEÇÃO. OU SEJA, MANTEMO-NOS ALINHADOS COM O CONCEITO DE LINGUAGEM DE PROGRAMAÇÃO SOBRE O MECANISMO DE TRATAMENTO DE EXCEÇÃO. SINALIZAÇÃO E LANÇAMENTO DE EXCEÇÃO TORNAM-SE SINÔNIMOS.

A razão de fazermos isso é porque alguns profissionais chamam ao mecanismo pelo qual um método Java notifica o chamador das exceções que pode lançar como sinalização de exceções. Isso, contudo, é uma abordagem predisposta à confusão. Tal mecanismo, para fins de nossa discussão, será referenciado como mecanismo de notificação de exceção.

Veremos nas sessões seguintes, em mais detalhes, os processos que formam o tratamento de exceções em Java. Abordaremos a notificação de exceção, o seu lançamento e como relançar uma exceção, e concluiremos falando sobre o tratamento.

TRATAMENTO DE EXCEÇÕES



ENTENDENDO O TRATAMENTO DE EXCEÇÕES EM JAVA

No vídeo a seguir, apresentamos notificação, lançamento, relançamento e tratamento de exceções em Java.

NOTIFICAÇÃO DE 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: AS EXCEÇÕES EXPLÍCITAS PRECISAM SER TRATADAS PELO PROGRAMADOR, ENQUANTO AS IMPLÍCITAS PODEM SER SIMPLESMENTE DEIXADAS PARA O TRATADOR DE EXCEÇÕES PADRÃO DE JAVA. QUANDO UMA EXCEÇÃO IMPLÍCITA É LANÇADA, SE NENHUM TRATADOR ADEQUADO FOR LOCALIZADO, A EXCEÇÃO É PASSADA PARA O TRATADOR PADRÃO, QUE FORÇA O ENCERRAMENTO DO PROGRAMA.

Pela mesma razão, ainda que sejam notificadas pelo método, as exceções implícitas não obrigam a que o chamador defina um contexto de tratamento de exceções para invocar o método. Isso não se passa com as exceções implícitas, pois como o compilador impõe que o programador as trate, uma de duas situações deverá ocorrer: ou se define um contexto para tratamento de exceções ou o chamador notifica, por sua vez, outro método que venha a lhe chamar de que pode lançar uma exceção.

💡 DICA

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.

LANÇAMENTO DE 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 14. 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.

Código 14: Exemplo de código que não compila devido a erro no lançamento de exceção.

```
public class Calculadora {
public int divisao ( int dividendo , int divisor ) throws ArithmeticException ,
FileNotFoundException
{
throw new FileNotFoundException ( "Divisor nulo." );
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;
}
}
```

Situações assim podem ser resolvidas, porém, por meio do aninhamento de blocos try. Quando múltiplos blocos try são aninhados, cada contexto de tratamento de interrupção é empilhado até o mais interno. Quando uma exceção ocorre, os blocos são inspecionados em busca de um

bloco catch adequado para tratar a exceção. Esse processo começa no contexto no qual 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.

SAIBA MAIS

Uma maneira menos óbvia em que o aninhamento se dá é quando há um encadeamento de chamadas a métodos onde cada método tenha definido seu próprio contexto de tratamento de exceção. Esse é exatamente o caso se combinarmos o Código 10 e o Código 13. Veja que, na linha 4 do Código 10, 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 13, 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!

PRÉ-CONDIÇÕES

São condições estabelecidas pelo programador que devem ser satisfeitas a fim de que o método possa começar 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 dado que as pré-condições tenham sido satisfeitas, 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.

Podemos pensar num método que une (concatena) dois vetores (array) recebidos como parâmetro e retorna uma referência para o vetor resultante como um exemplo didático desse uso de exceções. Podemos estabelecer como pré-condição que nenhuma das referências para os vetores que serão unidos pode ser nula. Se alguma delas o for, então uma exceção NullPointerException é lançada. Não sendo nulas, então 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.

RELANÇAMENTO DE EXCEÇÃO

As situações que examinamos até o momento sempre caíram em uma de duas situações: ou as exceções lançadas eram tratadas, ou elas 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 seja tratada parcialmente. Essa situação difere das anteriores, como veremos.

QUANDO PROPAGAMOS UMA EXCEÇÃO LANÇADA AO LONGO DA CADEIA DE CHAMADORES, FAZEMOS ISSO ATÉ QUE SEJA ENCONTRADO UM BLOCO CATCH ADEQUADO (OU, COMO JÁ DISSEMOS, 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, SERÁ PRECISO QUE A BUSCA POR OUTRO BLOCO CATCH ADEQUADO SEJA REINICIADA, POIS, COMO A EXCEÇÃO FOI CAPTURADA, ESSA BUSCA TERMINOU. FELIZMENTE, É POSSÍVEL REINICIAR O PROCESSO RELANÇANDO-SE A EXCEÇÃO CAPTURADA.

Relançar uma exceção permite postergar seu tratamento, ou parte dele. Assim, um novo bloco catch adequado, associado a um bloco try mais externo, será buscado. É claro que, uma vez relançada a exceção, o procedimento transcorre da mesma forma como ocorreu no

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 15 mostra o relançamento da exceção.

Código 15: Exemplo de relançamento de exceção na classe Calculadora.

```
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;
}
}
```

📢 ATENÇÃO

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 9. O bloco finally não recebe a referência para a exceção e esta é uma variável local do bloco catch.

CAPTURA DE EXCEÇÃO

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. Quer dizer que, 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, contudo, não podem ocorrer de maneira independente no código. Eles precisam, sempre, estar associados a um bloco try. E também não podem ser aninhados, como os blocos try. Então, como lidar com o Código 16?

Código 16: Possibilidade de lançamento de tipos distintos de exceções.

```
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;
}
```

Observe que se tamnh for zero, a linha 3 irá gerar uma exceção ArithmeticException (divisão por zero), e 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. Mas a linguagem Java nos dá uma solução mais elegante. Podemos empregar múltiplas cláusulas catch, como mostrado no Código 17.

Código 17: Múltiplas cláusulas catch.

```
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;
}
```

📢 ATENÇÃO

Um cuidado que se precisa ter é que 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.

VERIFICANDO O APRENDIZADO

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 relançado.

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

2. UM PROGRAMADOR CRIOU UMA EXCEÇÃO (NOTREFERENCEDEXCEPTION) A PARTIR DA CLASSE RUNTIMEEXCEPTION. CONSIDERANDO SEUS CONHECIMENTOS DE TRATAMENTO DE EXCEÇÃO EM JAVA, MARQUE A ÚNICA ALTERNATIVA CORRETA QUANDO 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.

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.

GABARITO

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

A alternativa **"E "** está correta.

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

2. Um programador criou uma exceção (NotReferencedException) a partir da classe RuntimeException. Considerando seus conhecimentos de tratamento de exceção em Java, marque a única alternativa correta quando 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.

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 este bloco catch e nunca chegará ao bloco destinado a tratar exceções do tipo NotReferencedException.

CONCLUSÃO

CONSIDERAÇÕES FINAIS

O tratamento de exceções não é uma exclusividade da linguagem Java. Outras linguagens de programação a implementam. Java inspirou sua implementação no tratamento de exceções da linguagem C++, mas não se limitou a copiá-lo, oferecendo uma abordagem própria.

Iniciamos nosso estudo abordando os tipos de exceções. Compreendemos o que são exceções implícitas e explícitas e discorremos sobre a criação de novos tipos de exceção. Em seguida, nos detivemos na análise das instruções finally, throw e throws, importantes peças do tratamento de exceção em Java, e finalizamos com um estudo mais detido na notificação, no lançamento, relançamento e tratamento de exceções.

Ao longo do nosso estudo, pudemos ver as diversas formas em que o tratamento de exceções contribui para um código mais robusto, legível e organizado. É, certamente, uma ferramenta a ser empregada por profissionais capacitados e sérios. Mas, para tirar todo o proveito que pode ser obtido, é preciso conhecer as nuances desse mecanismo!

Para ouvir um *podcast* sobre o assunto, acesse a versão online deste conteúdo.

AVALIAÇÃO DO TEMA

REFERÊNCIAS

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

ORACLE AMERICA INC. **What Is an Exception?** (The Java™ Tutorials > Essential Classes > Exceptions). Consultado na internet em: 01 jun. 2021.

SEUNGIL, L. *et. al.* **Efficient Java Exception Handling in Just-in-Time Compilation.** Publicado em: 03 jun. 2000. Consultado na internet em: 01 jun. 2021.

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.

CONTEUDISTA

Marlos de Mendonça Corrêa

🔗 CURRÍCULO LATTES