Unidade IV

7 EXCEÇÕES

Ao longo do desenvolvimento de sistemas, estamos sujeitos a situações que podem gerar erros de execução. Um erro de execução não é detectado na compilação do sistema, já que o código está correto, e pode acontecer a qualquer momento durante a utilização do sistema pelo cliente.

A prática de programação permite que o desenvolvedor preveja muitos dos possíveis problemas que podem ocorrer quando o sistema estiver rodando em produção. O problema está no fato de que caso ocorram alguns desses imprevistos, um erro pode encerrar todas as tarefas executadas no servidor, de modo a termos que reiniciá-lo.

Seguem alguns exemplos de erros de execução de sistema que podem ocorrer, apesar de toda a previsão e experiência do desenvolvedor:

- tentativa de acesso a banco de dados quando ele está fora do ar;
- um cálculo impossível (como uma divisão por zero) executado pelo cliente;
- uma falha de leitura de um determinado hardware ou periférico;
- uma falha física de acesso à memória.

A ocorrência desses erros gera o que chamamos de erros ou exceções no sistema (que vamos chamar a partir de agora apenas de exceções).

O compilador Java está preparado para identificar boa parte dessas exceções, gerando um aviso de exceção na tela da console.

Por exemplo, o programa a seguir não gera interrupção, pois não há exceções ocorrendo durante a sua execução:

```
int a = 12;
int b = 4;
int c = a / b;
System.out.println(c);
// --------------------
System.out.println("FIM DO PROGRAMA");
```

```
// ## Imprimirá na tela:
// 3
// FIM DO PROGRAMA
```

No entanto, se fizermos com que a variável b tenha valor 0, como no exemplo a seguir, o programa sofrerá uma interrupção:

```
int a = 12;
int b = 0;
int c = a / b;
System.out.println(c);
// --------------------
System.out.println("FIM DO PROGRAMA");
```

Neste caso, como não há tratamento desse tipo de ocorrência, essa interrupção provocará uma parada geral de todas as outras execuções de programa do compilador. No caso, se esse sistema estiver sendo rodado em um servidor de aplicação, ele sofrerá uma parada e interromperá todas as outras execuções de programas, necessitando ser reiniciado.

Ao executar este último programa, veremos na console do servidor uma mensagem (geralmente em vermelho):

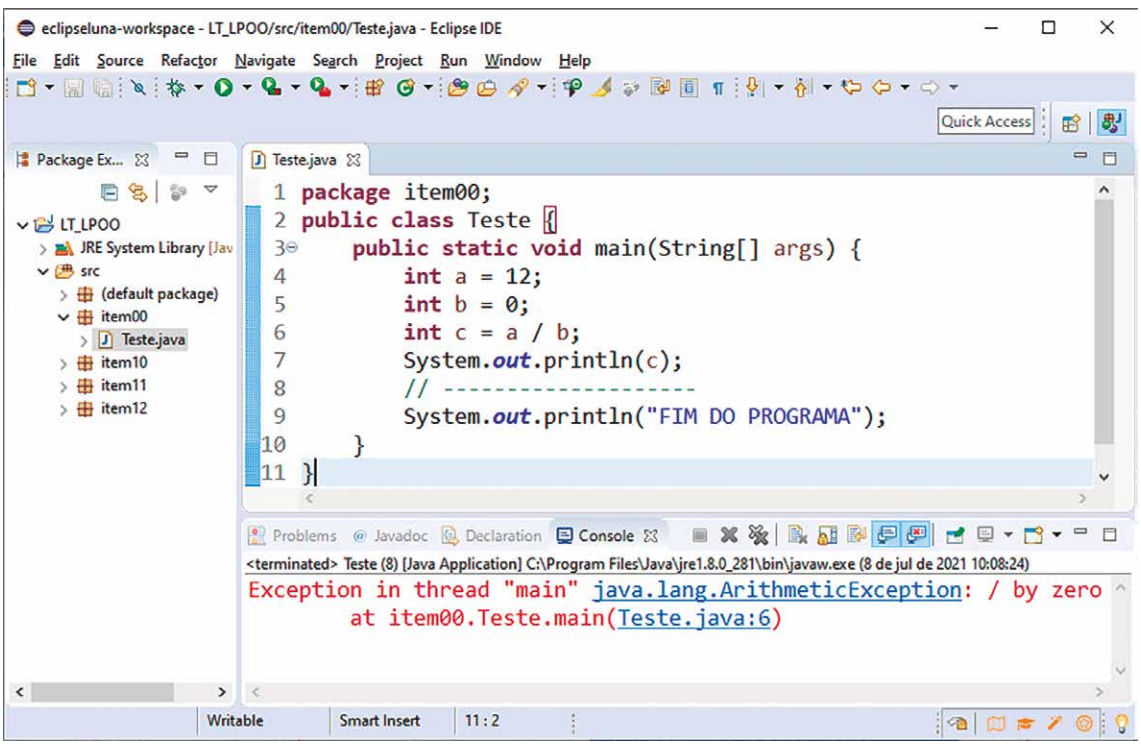

Figura 30 – Tela do Eclipse mostrando a localização do arquivo, o programa e a console com o resultado da execução

A mensagem da imagem mostra em que ponto (e de qual programa) ocorreu a exceção. Nela, temos o número da linha de código (6), de qual método e classe (método main da classe Teste do pacote item00), além da identificação da exceção ocorrida (java.lang.ArithmeticException) e uma explicação textual do erro (/ by zero, ou divisão por zero).

Observação

Na aritmética com números inteiros, não é possível realizarmos uma divisão de um número por 0.

Percebe-se nessa tela da console que o programa nem imprimiu o texto "FIM DO PROGRAMA", já que foi interrompido na linha 6 de execução e, consequentemente, não executou nenhuma linha de comando abaixo dela.

Essa mensagem, deixada pelo compilador na tela da console, procura dar uma pista aos administradores de rede e, também, à equipe de desenvolvedores Java, para identificar qual sistema gerou a exceção, para que ele possa ser acertado pela equipe de desenvolvimento, como um meio de facilitar a sua manutenção, e não gerar mais problemas ao servidor.

7.1 Hierarquia das exceções

Sabemos que na linguagem Java a base de construção de sistemas é a criação de classes (com seus atributos e métodos). Dessa forma, as exceções são classes da biblioteca do Java e, portanto, ao se manifestarem (ao serem instanciadas) se transformam em objetos (em memória) que também possuem atributos, métodos e construtores, como qualquer outra classe que vimos até agora.

As exceções se utilizam dos conceitos de orientação a objetos e compartilham de suas possibilidades e facilidades, como herança, polimorfismo, encapsulamento etc. Seguindo essa ideia, existe uma árvore hierárquica das exceções composta de todas as classes da biblioteca do Java, que podem ser acionadas assim que o problema é identificado. Para cada tipo de problema existe uma classe (uma exceção) específica a ser acionada.

A hierarquia das exceções está relacionada a superclasses e subclasses que compõem todas as possíveis classes que podem ser acionadas, dependendo do problema ocorrido na execução dos programas.

Lembrete

Uma superclasse (classe mãe) é uma classe que foi herdada por outra classe, enquanto a subclasse (classe filha) é a classe que herdou a superclasse.

A imagem a seguir mostra uma pequena parte dessa hierarquia, já que existem muitas outras classes na biblioteca além daquelas que podem ser criadas por qualquer desenvolvedor, e que são específicas para determinados sistemas.

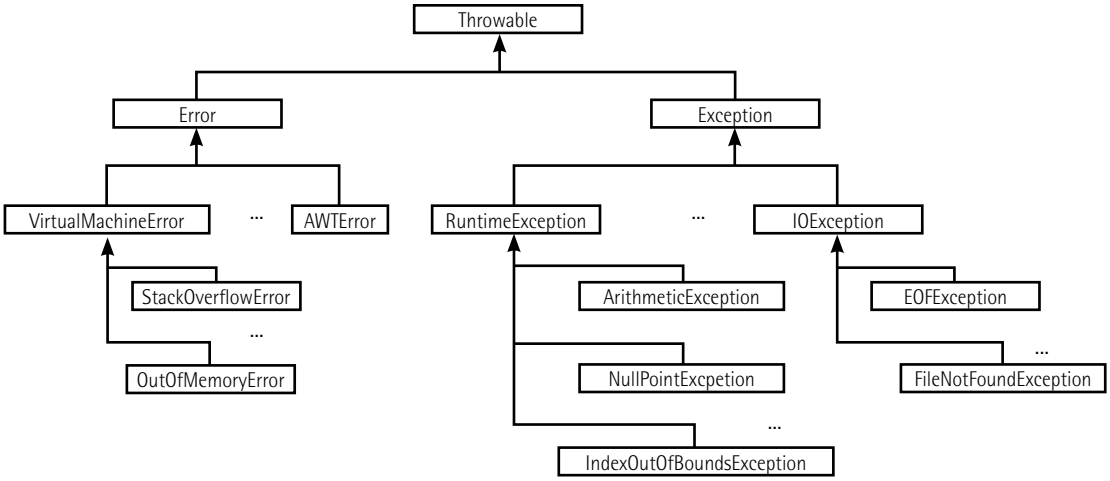

Figura 31 – Estrutura da classe java.lang.Throwable, raiz (mãe) de todas as classes que provocam interrupções

Na imagem, pode-se perceber que a classe Throwable é a classe mãe de todo tipo de interrupção que pode ocorrer em um sistema.

A hierarquia que tem como base a classe Error, uma das subclasses da classe Throwable, equivale mais a problemas que podem ocorrer na parte física de processamento, como na memória, por exemplo, além de outras situações. Esses são problemas que não costumam ocorrer normalmente. Já a hierarquia que tem como base a classe Exception, outra subclasse da classe Throwable, equivale a problemas de funcionalidade que podem ser previstas, porém nunca se sabe quando efetivamente poderão ocorrer.

Para facilitar o trabalho dos desenvolvedores, existe na linguagem Java um recurso que obriga e auxilia a "previsão" desses problemas, de forma a exigir o seu tratamento (que veremos no item a seguir).

Algumas classes de exceções obrigam sempre o seu tratamento, enquanto outras classes não o fazem. A utilização das classes de exceções derivadas da classe Exception – **exceto as derivadas de RuntimeException** – obrigam o tratamento e representam condições previsíveis, ou seja, que podem acontecer ao longo da execução de um método.

Observação

Como observado no parágrafo anterior, a classe RuntimeException e suas subclasses não obrigam o seu tratamento. No entanto, se elas ocorrerem na execução de um programa em que não foi feito o seu tratamento, uma interrupção será acionada, e o servidor em que o sistema estiver rodando sofrerá uma parada de funcionamento, precisando ser acionado (ou "subido") manualmente.

7.2 Tratamento de exceções

Uma exceção é um objeto gerado para indicar a ocorrência de algum tipo de condição excepcional durante a execução de um método. Uma vez detectada essa condição anormal, é necessário ativar mecanismos para corrigi-la e permitir o prosseguimento da execução.

Esse tipo de procedimento é chamado de tratamento (ou manipulação) de exceções, o que, no código, se faz por meio da utilização de uma estrutura própria, um bloco de tratamento de exceções (exception handler). O programador deverá prever os pontos do programa suscetíveis a algum tipo de exceção e definir, antecipadamente, blocos de tratamento de exceções para cada um deles.

O tratamento de exceções consiste em duas etapas:

1 – "tentativa" da execução do código;

2 – captura do tipo da exceção.

Quando ocorre uma exceção, o sistema (ou a máquina virtual do Java) está preparado para verificar se há um tratamento previsto (codificado no programa) para aquela exceção e o executa. No entanto, se isso não foi previsto, e o programador não programou uma "saída" para aquele problema específico, o sistema gerará uma falha e bloqueará a execução dos códigos subsequentes, provocando uma parada (uma interrupção) em todas as eventuais execuções da máquina virtual.

Percebe-se que a importância de se capturar e tratar uma exceção é impedir que haja uma parada nas execuções da máquina virtual, impedindo com isso que haja um travamento das execuções do servidor de aplicações do Java.

A estrutura de captura e tratamento de exceções (try-catch-finally) apresenta a seguinte sintaxe:

```
try {
  // códigos que podem gerar uma exceção (tenta-se executar)
} catch (Excecao01 e) {
  // bloco de tratamento realizado a partir da
  // identificação de uma exceção do tipo Excecao01
} catch (Excecao02 e) {
  // bloco de tratamento realizado a partir da
  // identificação de uma exceção do tipo Excecao02
...
} catch (ExcecaoN e) {
  // bloco de tratamento para exceção mais genérica
  // (geralmente Exception ou Throwable)
} finally {
  // bloco opcional mas que sempre será executado
  // independente da ocorrência ou não de uma exceção
}
```

Uma estrutura de captura e tratamento de exceções (try-catch-finally) deve ter:

- apenas um bloco try (no início da estrutura);
- um ou mais blocos catch (cada bloco capturando uma exceção específica).

Opcionalmente, pode ter:

- apenas um bloco finally (ao final da estrutura).

Observação

O bloco finally, quando existente, será sempre executado, tendo ocorrido uma exceção ou não. De um modo geral, esse bloco inclui comandos de liberação de recursos que possam ter sido alocados no processamento do bloco try e que precisam ser liberados tendo sua execução sido concluída com sucesso ou não.

De uma forma geral, uma exceção, quando ocorre, gera a seguinte mensagem na console:

```
Exception in thread "main" java.lang.ExceptionName: general description
    at Class.method01(Class.java:line)
    at Class.method02(Class.java:line)
    ...
    at Class.main(Class.java:line)
```

Dessa forma, é possível rastrear uma exceção, já que na mensagem aparece o nome do método e de sua classe, além do número da linha de comando que gerou a exceção.

Por exemplo, a linha de comando a seguir gerará uma exceção de cálculo:

```
int a = 3, b = 0;
int c = a / b;
System.out.println("FIM DO PROGRAMA");
```

A segunda linha desse código (na verdade, a linha 9 do programa do qual esse código faz parte) gerará a seguinte mensagem de exceção:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at TesteA.main(TesteA.java:9)
```

Assim, neste caso, para se criar uma estrutura de captura e tratamento de exceções, basta **envolver** a linha que gerou a exceção com um bloco try-catch, copiar o nome da exceção e colocá-lo como parâmetro de captura:

```
int a = 3, b = 0;
try {
  int c = a / b;
} catch (ArithmeticException e) {
  // lógica de tratamento
}
System.out.println("FIM DO PROGRAMA");
```

Executando este último programa-exemplo, veremos que não estamos impedindo de gerar o erro na sua execução, no entanto, perceberemos que dessa forma o sistema como um todo não será interrompido, já que existe um tratamento para o erro ocorrido.

7.2.1 Tratando várias exceções

Muitas vezes, num mesmo bloco de código, podemos ter várias situações de erros diversos, podendo haver a necessidade de tratarmos de forma diferente cada uma delas.

Podemos citar como exemplo uma situação em que estamos lidando com dados em banco de dados (BD), na qual, ao tentarmos acessar determinada informação nele guardada, podemos nos deparar com os seguintes problemas:

- o BD está temporariamente fora do ar;
- o BD está no ar, mas excedeu o número máximo de sua capacidade de receber solicitações de acesso;
- foi possível acessar o BD, mas o dado solicitado não foi encontrado.

Perceba que num conjunto de ações sequenciais, vários problemas podem necessitar de tratamentos diferentes. Por exemplo:

- com o BD fora do ar, pode-se tentar acessá-lo mais aproximadamente quatro vezes, automaticamente via sistema; caso o erro persista, deve-se gerar um aviso ao usuário para que ele se comunique com o help-desk ou, ainda, gerar um e-mail à equipe de help-desk avisando-a do problema;
- para o problema de excesso de solicitações de acesso, pode-se mostrar uma mensagem ao usuário para que ele tente novamente mais tarde (pois aquele pode ser um horário de pico);
- para o problema da informação não encontrada, deve-se simplesmente avisar sobre o ocorrido e sugerir uma nova busca.

Tratar cada uma das exceções significa justamente dar um encaminhamento diferente para cada tipo de problema que ocorrer (se ocorrer). Dessa forma, quando diversos tipos de exceções podem ser gerados na execução de um mesmo bloco de código, então é importante que se identifique a maioria das exceções que podem ocorrer, gerando um bloco de tratamento específico para alguns deles, ou todos eles, ou genérico para boa parte deles. Isso é muito importante pois caso ocorra alguma exceção não prevista, e ela não for devidamente capturada, essa exceção poderá interromper o curso de execuções de serviços do servidor, finalizando-o.

Como exemplo, podemos analisar o programa apresentado na imagem a seguir:

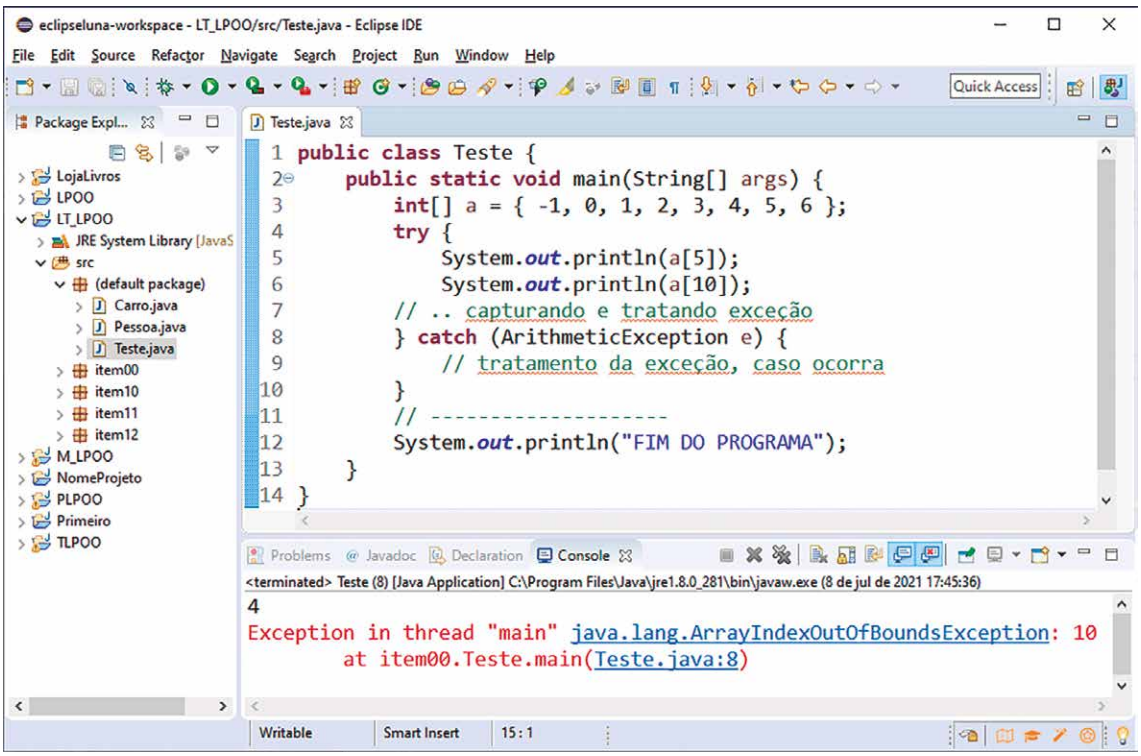

Figura 32 – Programa em que o tratamento da exceção não condiz com a exceção que ocorrerá efetivamente

Percebam que a primeira impressão na tela da console ocorreu sem problema (mostrando o valor da 5ª posição do vetor ocupada pelo número 4). No entanto, o sistema foi interrompido antes de chegar à última impressão (a impressão "FIM DO PROGRAMA" não foi executada como deveria).

Segundo a mensagem de exceção (em vermelho na imagem), o problema aconteceu na linha 8 e a exceção ocorrida foi:

`ArrayIndexOutOfBoundsException`

Decerto, se verificarmos o código da linha 8, veremos que ele tenta ler o valor da posição 10 do vetor (array) a, posição esta que não existe, já que ele contém apenas 8 valores. Essa tentativa incorreta de leitura gerou a exceção da mensagem.

Lembrete

Um array é uma variável que pode conter mais de um valor de um determinado tipo. Seus índices (posição de cada valor) iniciam em 0 e vão até o valor de seu tamanho (quantidade de valores) "menos um".

Contudo, por que o programa travou a continuidade da sua execução se havia um tratamento de erro previsto na sua codificação?

Se prestarmos atenção, a captura e o tratamento de exceção estavam focados sobre a exceção ArithmeticException. Porém, a exceção gerada foi outra, relacionada à leitura de posição inexistente em um vetor, e por isso o sistema entendeu que, como essa exceção não foi encontrada entre as opções de captura, não conseguiu seguir em frente, finalizando todas as execuções em andamento. Podemos, contudo, realizar várias capturas de exceções numa mesma estrutura try-catch, gerando um tratamento específico para cada uma delas:

Desse modo, a partir do exemplo anterior, podemos fazer:

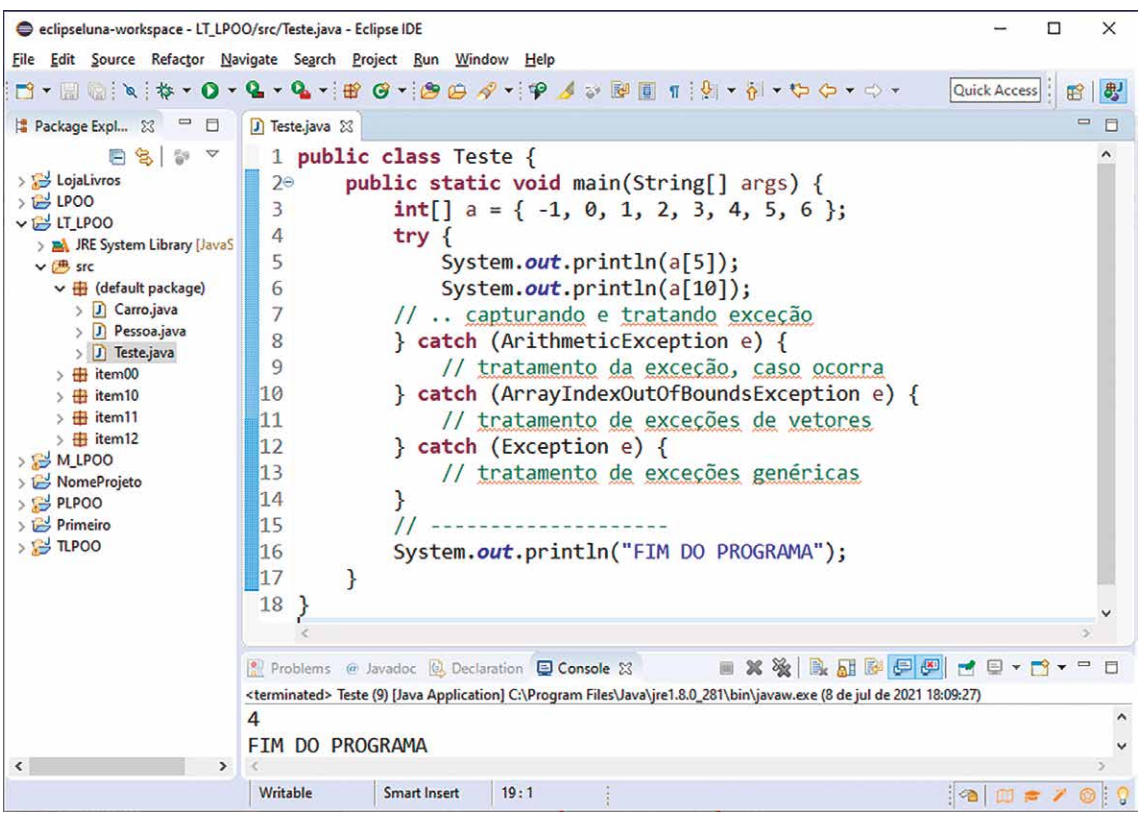

Figura 33 – Programa em que observamos múltiplos tratamentos de exceções a fim de garantir a integridade do sistema e do servidor

No exemplo que acabamos de apresentar, a exceção que ocorre na execução do código da linha 6 está devidamente capturada e tratada pelo sistema (nas linhas 10 e 11), de forma que, ao ser acionado, o programa executa o primeiro print (da linha 5), porém sofre uma interrupção no print da linha 6, passando a executar o tratamento que foi programado para rodar na ocorrência de sua captura.

Percebe-se que a última captura (linhas 12 e 13) é sobre um tipo de exceção mais genérico. Caso ao longo do bloco try se execute algum comando que provoque uma exceção não especificamente capturada, então cairá sobre uma captura genérica, e o tratamento programado a partir da linha 13 será acionado sem que o servidor seja finalizado, dando continuidade inclusive à execução das linhas que existem após o final do bloco try-catch.

Isso acontece porque, como dito no exemplo hipotético do parágrafo anterior, caso ocorra algum outro tipo de exceção, assim como explicado no item sobre polimorfismo de classes, o objeto e, parâmetro da captura da linha 13, por ser um objeto do tipo Exception (classe mãe das exceções), receberá e se comportará como um objeto relativo à exceção ocorrida (que certamente é uma classe filha da classe Exception) e, portanto, tem um tratamento definido nesse programa.

Observação

Sempre que se criar uma estrutura de captura e tratamento de exceções, caso venha a ser possível o sistema gerar uma exceção não prevista, deve-se inserir como última captura uma exceção genérica (Exception ou Throwable), garantindo assim a integridade de funcionalidade do seu sistema.

Essa captura deverá ser sempre a última, pois na ocorrência de uma exceção, o sistema sempre irá testar as exceções na ordem sequencial em que foram codificadas, de forma que caso a captura da Exception seja a primeira, sempre cairá nela, pois todas as exceções são subclasses de Exception.

7.3 Lançamento de exceções

Ao trabalharmos em um sistema em que muitas pessoas programam simultaneamente, ou ainda quando implementamos um sistema já existente no cliente, devemos sempre observar as regras de negócio desse cliente, gerando um sistema que esteja de acordo com elas. Neste caso, em vez de utilizarmos apenas as estruturas condicionais para definir e controlar essas regras, utilizamos também o lançamento de exceções nesse contexto, de modo a garantirmos que o programador se preocupe com as regras sempre que for utilizar aqueles métodos condicionados às exceções.

Lançar uma exceção é provocar uma interrupção em um ponto determinado do programa, que foi definido segundo as regras de negócio daquele sistema. Para fazer o lançamento da exceção e provocarmos essa interrupção em determinadas circunstâncias, utilizamos a palavra-chave throw, que gera uma exceção na JVM bloqueando a execução dos códigos existentes a partir do lançamento, caso ele efetivamente ocorra.

A sintaxe de um lançamento de exceção é:

```
throw new <tipo de exceção> (parâmetro);
```

O <tipo de exceção> é uma classe de exceção existente (na biblioteca Java ou alguma que tenha sido criada para o sistema), e o parâmetro (que pode existir ou não), que deve ser uma string, representará a mensagem de erro que será mostrada na console quando o erro (a exceção) ocorrer. Podemos usar como exemplo uma classe Conta (de um banco) em que existe um método de nome sacar(...), que recebe como parâmetro o valor que será acrescido na conta corrente.

Neste caso, sempre que algum programador venha a realizar alguma manutenção ao sistema e porventura se utilizar do método sacar(...), ele será forçado a envolver o método em um bloco try-catch para evitar que o servidor trave e, com isso, garantir que ele trabalhe na lógica de tratamento dessa ação em seu sistema.

Vejamos como fica a codificação do sistema desse exemplo:

```
public class Conta {
   private double saldo;
   public void sacar(double valor) {
     if (valor < 0) {
              String msg = "Este valor não pode ser negativo!!";
              System.out.println(msg);
              //.. e eventualmente alguma outra lógica necessária
              //.. para não deixar seguir com a transação
              //.. caso isto ocorra.
     } else {
              saldo -= valor;
     }
   }
}
```

Com o programa codificado que acabamos de apresentar, o criador do método sacar(...) será obrigado a definir toda a lógica de tratamento do problema quando for sacado um valor negativo.

```
public class TesteConta {
   public static void main(String[] args) {
     Conta c1 = new Conta();
     c1.sacar(-40);
   }
}
// A execução deste programa resultará na seguinte mensagem:
// Este valor não pode ser negativo!!
// ..de forma que este valor não será "retirado" no saldo.
```

Porém, esse código pode ser necessário ao funcionamento de vários outros sistemas criados muito tempo depois, não sendo possível se pensar em todos os problemas que poderão ocorrer em sistemas futuros com a utilização errada desse método.

Para isso, em vez de a lógica de tratamento estar codificada no método sacar(...), fazemos com que sempre que algum desenvolvedor (futuro) for se utilizar desse método, chamando o método sacar(...), será ele quem deve se preocupar em tratar o problema. Assim, o programador pode "lançar uma exceção" sempre que o problema ocorrer, para que a obrigação do tratamento do problema (o que fazer caso ele ocorra) fique a cargo do autor do outro programa que está chamando aquele método, sendo que para cada tipo de situação, o tratamento pode ser diferente.

No exemplo, vamos alterar apenas a 6ª linha de programação, lançando a exceção IllegalArgumentException(...).

```
public class Conta {
   private double saldo;
   public void sacar(double valor) {
     if (valor < 0) {
              String msg = "Este valor não pode ser negativo!!";
              throw new IllegalArgumentException(msg);
     } else {
              saldo -= valor;
     }
   }
}
```

No entanto, ao se chamar esse método, como a Classe IllegalArgumentException é uma subclasse da RuntimeException, ela não obriga o tratamento com a estrutura try-catch e, dessa forma, o programa a seguir, ao ser acionado, irá travar o servidor:

```
public class TesteConta {
   public static void main(String[] args) {
     Conta c1 = new Conta();
     c1.sacar(-40);
   }
}
```

Ele mostrará na console a seguinte mensagem:

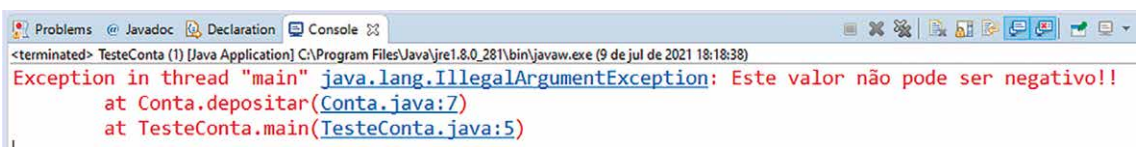

Figura 34 – Mensagem de exceção mostrada na console do Eclipse

No entanto, se tratamos a exceção ao chamarmos o método sacar(...), assim como no exemplo da classe TesteConta a seguir, evitaremos o travamento da JVM (do servidor) e, caso o problema efetivamente ocorra, garantimos a continuidade da execução do sistema e de todos os outros sistemas existentes em execução naquele servidor.

```
public class TesteConta {
   public static void main(String[] args) {
     Conta c1 = new Conta();
     try {
              c1.sacar(-40);
     } catch (Exception e) {
              // Lógica de Tratamento da Exceção
     }
   }
}
```

Mas a pergunta é: como o desenvolvedor iria saber que o método sacar(...) deveria ser envolto em uma estrutura try-catch para tratamento da exceção?

Na verdade, da forma com que o sistema está construído, não tem como o desenvolvedor saber que deveria tratar o método sacar() com uma estrutura try-catch, e a não ser que ele esteja a par do lançamento da exceção existente, o sistema estaria em risco de travar o servidor caso essa situação aconteça, e ele só ficaria sabendo da necessidade do tratamento da exceção após ocorrer o problema.

Porém, existe uma forma de obrigarmos o desenvolvedor a envolver a chamada do método sacar(...) numa estrutura try-catch. Para que isso seja possível, utilizamos a palavra reservada throws na linha de declaração do método, arremessando uma exceção que obriga o tratamento (uma exceção que **não** seja subclasse da RuntimeException, podendo ser, por exemplo, a própria classe Exception).

```
public void sacar(double valor) throws Exception {
   ...
}
```

Assim, a linha de código que chamar esse método sacar(...) será obrigada a ser tratada com uma estrutura de tratamento de exceção, envolvendo-a com o bloco try-catch.

Dessa forma, vimos que para o lançamento de uma exceção usamos duas palavras reservadas:

- **Throw**: palavra-chave utilizada dentro do código de um método, com a finalidade de interromper a execução de uma sequência de códigos por meio do arremesso de uma exceção:

  ```
  throw new TipoDeException(...);
  ```

- **Throws**: palavra-chave utilizada na declaração de um método e que vai arremessar aquele tipo de exceção, obrigando o tratamento a quem se utilizar daquele método.

```
public tipoRetorno metodo(param) throws TipoDaException { ... }
```

7.4 Criação de exceções

Além das exceções disponibilizadas na biblioteca do Java (em java.lang.Exception e java.lang.Error), é possível criar novas exceções de acordo com a necessidade do programador.

À medida que queremos impor as regras de negócio da empresa cliente no sistema que se está desenvolvendo, é importante que a equipe esteja ciente dessas regras. Com isso, podemos gerar interrupções personalizadas com exceções criadas a partir dessas regras. Essas exceções personalizadas podem ser úteis para se associar determinadas falhas de execução ou regras de negócio da empresa que exigem determinadas situações que definem como ela deve ser utilizada e que obrigarão ao desenvolvedor o tratamento do código à medida que se utilizem dos métodos controlados.

Para criarmos exceções novas, deve-se criar uma classe que representa essa regra (sempre, de acordo com a convenção de programação Java, com a palavra Exception ao final), e fazemos com que ela herde a classe Exception. Essa nova classe terá métodos construtores que devem chamar a classe superior (com a palavra-chave super), inserindo a mensagem a ser mostrada na tela da console, caso o problema aconteça.

Vamos imaginar que para o exemplo anterior, em que estamos gerando um sistema de acesso à conta bancária, o cliente queira controlar a situação da conta corrente, e o programador não pode permitir que o cliente do banco saque um valor acima do valor de seu saldo. Para isso, podemos criar uma classe de exceção com esse objetivo, fazendo com que todo desenvolvedor de sistemas que venha a se utilizar do método de saque de valor fique a par da necessidade desse controle e preveja o seu tratamento.

Com isso, no exemplo pode-se ser criada a seguinte classe:

```
public class SemSaldoException extends Exception {
  public SemSaldoException () {
    super("Não há Saldo para esta Transação!!");
  }
  public SemSaldoException (String msg) {
    super(msg);
  }
}
```

No exemplo, criou-se a classe SemSaldoException, que herda Exception (e por definição é uma Exception). Essa classe, como normalmente é o padrão das exceções, possui dois métodos construtores: um sem parâmetro, com uma mensagem padrão; e outro que recebe a mensagem como parâmetro.

Além disso, o fato de herdar a classe Exception, e não a classe RuntimeException, torna-o um método que quando for chamado obrigará o seu tratamento no método que o chamar (utilizar).

Dessa forma, a classe Conta pode se utilizar dessa exceção para o método sacar(...), lançando a exceção e ficando da seguinte forma:

```
public class Conta {
   private double saldo;
   public void depositar(double valor) throws Exception {
     if (valor < 0)
            throw new IllegalArgumentException("Este valor não
pode ser negativo!!");
     else
            saldo += valor;
   }
   public void sacar(double valor) throws Exception {
     if (valor < 0)
            throw new IllegalArgumentException("Este valor não pode ser negativo!!");
     else if (valor > saldo)
            throw new SemSaldoException("Saque acima do Saldo...");
     else
            saldo -= valor;
   }
   public void mostrarSaldo() {
     System.out.println("Saldo atual: R$ " + String.format("%.2f", saldo));
   }
}
```

Assim, ao gerarmos a classe que vai se utilizar dos métodos depositar e sacar, se simplesmente chamarmos esses métodos sem tratá-los, observaremos o seguinte erro:

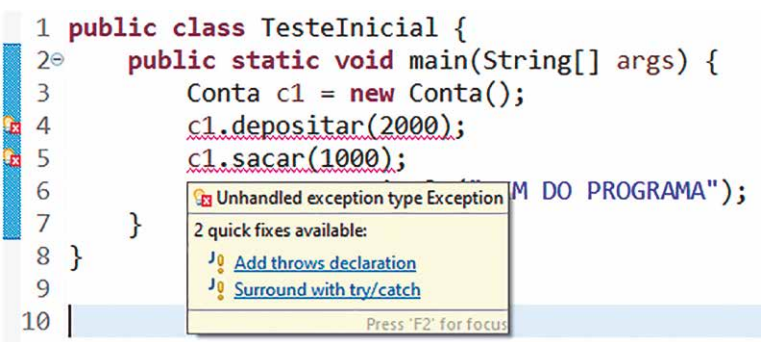

Figura 35 – Mensagem do erro de compilação do Eclipse sobre a chamada dos métodos depositar e sacar

Observa-se que duas linhas da classe TesteInicial (as linhas de código 4 e 5) estão acusando erro, pois os métodos depositar e sacar agora exigem o tratamento, o qual podemos ver no seguinte exemplo:

```
public class TesteInicial {
   public static void main(String[] args) {
     Conta c1 = new Conta();
     try {
              c1.depositar(2000);
              c1.mostrarSaldo();
              c1.sacar(1500);
              c1.mostrarSaldo();
              c1.sacar(2000);
     } catch (Exception e) {
              e.printStackTrace();
     }
     c1.mostrarSaldo();
     System.out.println("FIM DO PROGRAMA");
   }
}
```

Rodando o exemplo, teremos a seguinte saída na tela da console:

```
<terminated> TesteInicial [Java Application] C:\Program Files\Java\jre1.8.0_281\bin\javaw.exe
Saldo atual: R$ 2000,00
Saldo atual: R$ 500,00
SemSaldoException: Saque acima do Saldo...
        at Conta.sacar(Conta.java:14)
        at TesteInicial.main(TesteInicial.java:9)
Saldo atual: R$ 500,00
FIM DO PROGRAMA
```

Figura 36 – Impressão da tela da console ao rodarmos a classe TesteInicial

Neste exemplo, ao tentar sacar R$ 2.000,00 após sacar R$ 1.500,00, o sistema acusou uma exceção do tipo SemSaldoException, que foi a exceção criada para tratar essa regra de negócio da empresa.

Saiba mais

Saiba mais sobre exceções no capítulo 11, itens 11.4 e 11.5 de:

HORSTMANN, C. *Conceitos de computação com Java*. 5. ed. Porto Alegre: Bookman, 2009.

Exemplo de aplicação

Com relação ao conceito das exceções, podemos afirmar que:

I – Quando um bloco try pode gerar vários tipos de exceções diferentes, podemos criar várias formas de tratamento, capturando individualmente cada uma delas, mas sempre lembrando de deixar a exceção mais genérica por último.

II – Quando uma exceção acontece, o sistema irá sempre acusar sua ocorrência para que se possa ter um controle dela a partir de mensagens na console e continuar a executar as linhas de códigos subsequentes à que gerou aquela exceção.

III – Para que uma exceção exija o seu tratamento, devemos lançar a exceção na declaração do método com a instrução throws, desde que não seja lançada utilizando-se a exceção RuntimeException (ou suas subclasses).

IV – Num mesmo bloco try, é possível que aconteçam duas exceções uma após a outra em uma mesma execução daquele bloco, e isso pode exigir um tratamento individual para cada uma delas.

De acordo com as afirmativas, podemos dizer que estão corretas:

A) Apenas as afirmativas II e III.

B) Apenas as afirmativas II e IV.

C) Apenas as afirmativas I e II.

D) Apenas as afirmativas I e III.

E) Apenas as afirmativas III e IV.

Resposta correta: alternativa D.

Resolução

I – Afirmativa correta.

Justificativa: um bloco try pode vir a gerar, em ocasiões diferentes, mais de um tipo de exceção, de modo que o tratamento pode ser individualizado em cada uma das ocasiões. O tratamento dá-se no bloco de captura (catch), o qual poderá existir um para cada tipo de exceção. Devemos sempre lembrar de incluir uma captura de exceção genérica (por exemplo a Exception ou a Throwable), pois erros inesperados podem ocorrer, e se não estiverem tratados nas capturas individuais, poderão travar a execução do servidor. No entanto, essa captura de erro genérico deve ser a última, pois o tratamento

se dá no bloco da primeira exceção da qual aquele erro se encaixa e pela característica de polimorfismo, uma exceção genérica, pode representar qualquer uma de suas especificidades (subclasses).

II – Afirmativa incorreta.

Justificativa: ao acontecer uma exceção, o sistema verifica se há tratamento para ela ou para uma generalização dela (superclasse); se não houver, o programa trava sua execução, encerrando-se. E mesmo que haja o tratamento, a execução não continua a partir do ponto da exceção, mas sim nos códigos subsequentes ao bloco try-catch de captura e tratamento.

III – Afirmativa correta.

Justificativa: um método em que na sua declaração é lançada uma exceção com a instrução throws (por exemplo, public void nomeMetodo() throws Excepton {...}) exigirá que o desenvolvedor envolva a chamada ao método em um bloco try-catch. É importante saber que a exceção RuntimeException, ou qualquer uma de suas subclasses, não exige o envolvimento com o bloco try-catch na invocação do método que porventura a lançar.

IV – Afirmativa incorreta.

Justificativa: quando ocorre uma exceção, o sistema interrompe a execução do bloco try e pula para a captura da exceção a partir dos eventuais blocos catch que existirem, e não mais volta a execução daquele bloco try, seguindo a execução dos códigos que existirem após a estrutura de tratamento (try-catch). Isso quer dizer que a partir da execução de um mesmo bloco try não é possível ocorrer mais uma exceção na mesma execução.

8 THREADS

A maioria dos programas que utilizamos em nosso micro trabalha no modo multitarefa (ou multitask), ou seja, funciona acionando vários recursos ao mesmo tempo. Nosso sistema operacional é um sistema multitarefa, já que conseguimos acionar vários programas ao mesmo tempo.

Enquanto estou escrevendo este livro-texto em meu computador, posso escutar uma playlist em meu fone de ouvido conectado ao micro acionado a partir de um streaming de músicas e, numa outra tela, tenho meu browser acessando meu e-mail, com o micro conectado à internet. Isso só é possível porque, assim como todo notebook existente, esse meu micro permite que vários programas rodem ao mesmo tempo, caracterizando um processo multitarefado – ou, como alguns livros descrevem, um processamento de sistemas em paralelo.

O recurso de programação que permite esse multiprocessamento de sistemas em paralelo é chamado de thread. A utilização de vários programas distintos normalmente já é gerenciado pelo próprio sistema operacional.

Como um exemplo mais simples, imagine que estejamos construindo um programa que gera um relatório em PDF. Como às vezes esse processo é um pouco demorado, para dar alguma satisfação ao usuário, é interessante que ele veja na tela uma barra de progresso, que pode dar uma ideia do tempo restante de processamento. Só isso já caracteriza um processamento paralelo.

Neste caso, quando temos um programa rodando e gerando vários processamentos, normalmente para isso utilizam-se threads, que permitem múltiplas atividades dentro de um único processo.

Existem várias razões para se utilizar threads:

- maior desempenho em ambientes multiprocessados;
- responsividade em interfaces gráficas;
- simplificação na modelagem de algumas aplicações.

8.1 Criando threads

Existem duas maneiras de se trabalhar com uma thread em Java:

- **utilizando herança**: criamos uma classe que herda (estende a ideia) a classe Thread;
- **usando implementação de interface**: criamos uma classe que implementa a interface Runnable.

Por boas práticas, geralmente implementamos a interface Runnable em vez de herdar a classe Thread. Isso ocorre porque a linguagem Java permite a herança direta de apenas uma classe, e como a herança é um recurso muito comum (é uma das características fundamentais da orientação a objetos), podemos fazer com que a classe herde uma outra classe e implemente a interface Runnable.

De qualquer forma, seja herdando a classe Thread, seja implementando a interface Runnable, o método que rodará em paralelo com qualquer outro processamento do sistema, inclusive com outras threads, será o método run(), a partir do acionamento do método start(), e é esse método run() que deve acionar todos os outros métodos necessários para o processo.

A seguir, veremos como é essa implementação com ambas as formas.

8.1.1 Pela herança da classe thread

Utilizando herança, a classe que terá um ou mais processamentos em paralelo deve herdar a classe Thread. Para que funcione, deve-se sobrescrever o método

```
public void run()
```

Exemplo:

```
public class Exemplo1 extends Thread {
   public void run() {
     // roda alguma lógica a ser
     // executada em paralelo com outros processos
   }
}
```

8.1.2 Usando a interface runnable

Implementando a interface Runnable, seremos obrigados a implementar o método

```
public void run() {…}
```

Exemplo:

```
public class Exemplo2 implements Runnable {
   public void run() {
     // roda alguma lógica a ser
     // executada em paralelo com outros processos
   }
}
```

8.1.3 Comparando

As duas formas são utilizadas praticamente da mesma forma. A única diferença é que uma herda a classe Thread e a outra implementa a interface Runnable. Como foi dito anteriormente, a vantagem de se implementar a interface Runnable é que a classe que se comportará como uma thread fica livre para herdar outra classe qualquer.

8.2 Rodando threads

Para acionarmos uma ou várias threads é necessário acionar o start em cada uma delas, ou seja, rodar o método start() que existe (por herança) em cada classe que implemente essa funcionalidade.

Quando acionamos o start, estamos iniciando um processamento em paralelo e liberando o programa para executar qualquer outra thread. O acionamento desse start aciona o método run() criado na classe que implementou ou herdou a thread.

Do exemplo anterior (a classe Exemplo), seja ela herdando a classe Thread ou implementando a interface Runnable, para que seu método run() rode em paralelo com outros processamentos, basta que acionemos o método start() que, automaticamente, seu método run() será acionado em paralelo.

A única diferença é que no segundo caso, ao se implementar a interface Runnable, é necessário que se crie uma instância da classe Thread em que esta recebe como parâmetro de seu método construtor a classe que implementou a inteface Runnable. Veja o exemplo a seguir:

```
Exemplo1 ex1 = new Exemplo1(); // que herda Thread

ex1.start();

Exemplo2 ex2 = new Exemplo2(); // que implementa Runnable

Thread t2 = new Thread(ex2);

// ou...: Thread t2 = new Thread(ex2, "Nome da Thread");

t2.start();
```

Neste caso, tanto o método run() da classe Exemplo1 quanto o da classe Exemplo2 rodarão em paralelo.

Observação

O programador não tem nenhum controle sobre o escalonador do processador. Isso significa que sendo os processos executados paralelamente, não há como saber qual a ordem de execução das linhas de código de cada um desses métodos acionados pela thread.

Um bom exemplo de acionamento de threads é o conhecido e clássico jogo do Pac-Man. Se esse jogo fosse desenvolvido na linguagem Java, poderia ser construído de várias maneiras, e para cada personagem teríamos uma thread controlando seus movimentos. Uma thread controlaria os movimentos do personagem principal a partir do acionamento das setas do teclado (acionamento realizado pelo usuário-jogador), enquanto outra controlaria cada um dos movimentos dos fantasmas que perseguirão o personagem principal do jogo, com estratégias controladas por algum programa de inteligência artificial que seria acionado pelo método run() de cada uma das classes que os representam.

Construindo um programa que utiliza Thread

Neste exemplo, construiremos uma classe chamada Contagem com um método run() que, ao rodar, imprimirá os números de 1 a 10 na tela da console, sendo cada número numa linha, e precedidos de uma identificação (uma string) recebida a partir do método construtor daquela classe. Essa classe implementará a interface Runnable, e com isso será possível executá-la por meio de uma thread.

```
public class Contagem implements Runnable {
   private String id = "";
```

```
    public void run() {
      System.out.println("Iniciando para " + id + ":");
      for (int x = 1; x <= 10; x++) {
              System.out.println(id + ": " + x);
              // Dando um tempo de espera de 0,1 segundos
              try {
                      Thread.sleep(100);
              } catch (InterruptedException e) {
              }
      }
    }
    public Contagem(String s) {
      id = s;
    }
}
```

A partir dessa classe, construiremos uma classe de teste que irá instanciar a classe Contagem duas vezes, cada uma com uma identificação diferente, de modo que iremos rodar individualmente o método run() de cada uma delas e, em seguida, criaremos duas threads individuais, em que acionaremos esses mesmos métodos a partir do método start(). Com isso, veremos a diferença do acionamento individual do método run() e do seu acionamento via thread, por meio do método start().

```
public class TesteCont {
    public static void main(String[] args) {
      Contagem c1 = new Contagem("C1");
      Contagem c2 = new Contagem("C2");
      // Primeiro: rodaremos individualmente os
      // métodos de ambos os Objetos (c1 e c2)
      c1.run();
      c2.run();
      // Segundo: acionaremos os mesmos métodos
      // a partir de Threads
      Thread t1 = new Thread(c1);
      Thread t2 = new Thread(c2);
      t1.start();
      t2.start();
    }
}
```

Ao rodarmos o programa, veremos que o sistema primeiro rodará **todo** o método run() para c1, mostrando os números de 1 a 10 precedidos do termo C1, em seguida rodará **todo** o método run() para c2, mostrando também os números de 1 a 10, no caso, precedidos do termo C2. Isso acontece porque estamos acionando individualmente, e diretamente, o método run() de cada um dos objetos. Em seguida a isso, veremos que para as duas classes, os métodos run() serão acionados quase ao mesmo tempo (um

logo após o outro, pois o start é dado individualmente para cada um deles) e que rodarão praticamente juntos, tanto para c1 quanto para c2 (às vezes, inclusive, alternando a impressão entre um e outro).

Saiba mais

Saiba mais sobre threads no capítulo 11, itens 11.4 e 11.5, de:

WINDER, R. *Desenvolvendo software em Java*. Rio de Janeiro: LTC, 2009.

Exemplo de aplicação

Vimos no item anterior que a linguagem Java é uma linguagem multithread. Isso quer dizer que:

A) Uma classe pode herdar múltiplas classes.

B) Podemos executar diversos eventos simultaneamente a partir de um mesmo programa Java.

C) Um mesmo método pode ser criado múltiplas vezes em uma mesma classe.

D) Um mesmo erro pode ser capturado de múltiplas formas.

E) Um programa Java pode ser desenvolvido por uma equipe com múltiplas pessoas.

Resposta correta: alternativa B.

Resolução

Nesta unidade, estudamos o que é thread e qual a sua utilidade. Vimos que ele pode ser acionado a partir do acionamento do método start() da classe Thread, executando uma série de eventos definidos e programados no método run() de uma classe que herda a classe Thread ou implementa a interface Runnable.

A) Alternativa incorreta.

Justificativa: a herança de classes independe de thread. Na linguagem Java, não existe herança múltipla, mas sim por encadeamento.

B) Alternativa correta.

Justificativa: o objetivo do Thread é justamente possibilitar a execução simultânea de diversos eventos (métodos) sempre que necessário, como acontece em games, ou ainda quando, por exemplo, queremos mostrar para o usuário uma barra de progresso mostrando o andamento da execução de uma transação, quando ela pode ser demorada.

C) Alternativa incorreta.

Justificativa: o thread não está relacionado ao desenvolvimento de métodos com características semelhantes. No entanto, sabemos que não pode haver mais de um método com mesma assinatura em uma mesma classe.

D) Alternativa incorreta.

Justificativa: a captura de erros ou exceções pode permitir que situações diferentes possam ser tratadas de formas diferentes. Mas essa característica não está relacionada a threads, mas sim às múltiplas utilidades do tratamento de exceções.

E) Alternativa incorreta.

Justificativa: o fato de trabalharmos individualmente ou em equipe não está relacionado à utilização de threads em um sistema. Normalmente trabalha-se em equipe na construção de sistemas em Java, e essa característica apenas exige um bom gerenciamento e acompanhamento de projetos.

Para finalizar esta unidade, observe os exemplos de aplicação a seguir. As resoluções são apresentadas na sequência.

Exemplos de aplicação

1 – Criar uma classe abstrata para guardar funções diversas (pode ser a classe FuncoesDiversas). Criar nessa classe uma função estática de nome ehNumero(...) que deve receber um valor do tipo string e retornar true ou false dependendo se o valor recebido for numérico (real, ou seja, double) ou não. Procure fazer uma verificação em que o sistema tente transformar o valor string recebido em um valor numérico do tipo double, utilizando-se da estrutura try-catch.

2 – Criar uma classe de teste com o método main, de modo que uma variável deve receber o valor a partir de uma input box (utilizar JOptionPane), e em seguida deve-se testar o valor utilizando-se do método criado no exercício 1 e somente transformar o valor em número se ele for efetivamente numérico.

3 – Criar uma classe abstrata chamada Calculadora com um método estático de nome dividir(...) que deve receber dois parâmetros do tipo double, de modo que o método deverá retornar o resultado da divisão do valor do primeiro parâmetro pelo valor do segundo. Faça com que esse método lance uma exceção caso o segundo parâmetro tenha valor zero (com a mensagem "Valor inapropriado – Zero!"), de modo que sempre que esse método for acionado (chamado) por qualquer ação de uma classe, ele obrigue que essa chamada seja feita sobre uma estrutura de tratamento de exceção (try-catch).

Criar então numa classe de teste que tenha em seu método main uma chamada para este método com quaisquer valores numéricos (zero ou não).

Observação importante: no Java, uma divisão de um valor numérico do tipo double por zero não gera uma exceção no sistema, retornando como resultado o valor Infinity (infinito).

4 – Criar uma thread equivalente à do exemplo descrito antes nesta unidade (exemplo da classe Contagem), mas que deverá imprimir os números de 1 a 100, de forma que cada número impresso tenha um tempo aleatório de espera entre uma impressão e outra.

Observação: o exemplo a seguir é o de um código que gera um número aleatório entre 1 e 100:

Sabendo que o número entre parênteses do método sleep(...) é o tempo de espera em milissegundos (milésimos de segundos), faça com que o número aleatório seja um valor inteiro entre 0 e 10 décimos de segundos.

Lembre-se que:

100 milésimos de segundo = 1 décimo de segundo

200 milésimos de segundo = 2 décimos de segundo

300 milésimos de segundo = 3 décimos de segundo

...

Exemplo de código que retorna um número inteiro aleatório entre 1 e 100:

```
Random rndm = new Random();

// Neste caso deve-se importar a Classe java.util.Random
...
int num = rndm.nextInt(100) + 1;
// Desta forma a variável "num" obterá um número entre 1 e 100
// Obs,: A função nextInt(valor) da Classe Random retorna
// um número entre 0(zero) e valor-1.
```

5 – Pesquise e descreva quando devemos utilizar a exceção do tipo IllegalArgumentException, que é uma subclasse da classe RuntimeException.

Resolução

Existem várias formas de se resolver cada um dos exercícios propostos. Nossa intenção é apresentar aqui uma dessas possíveis soluções, tentando dar uma lógica simples e prática.

1 –

```
public abstract class FuncoesDiversas {
   public static boolean ehNumero(String str) {
     boolean res = false;
     try {
              double num = Double.parseDouble(str);
              res = true;
     } catch (Exception e){
     }
     return res;
   }
}
```

A ideia dessa solução é tentar (try) transformar em double o valor recebido como string, de forma que a variável res, que inicia com o valor false, somente se tornará true após a transformação do valor em número caso isso seja possível (lembrando que quando ocorre uma exceção no bloco try, que no caso acontecerá se o valor não for numérico, o sistema para de executá-lo, passando a executar o bloco de captura catch), de forma que depois, ao sair do bloco try-catch, continua a executar a função, retornando o valor de res tendo este sido alterado ou não.

2 –

```
import javax.swing.JOptionPane;
public class TesteFD {
   public static void main(String[] args) {
     String str = JOptionPane.showInputDialog(null,
                    "Digite o valor!", "Entrada de Dados",
                    JOptionPane.INFORMATION_MESSAGE);
     double val = -1;
     if (FuncoesDiversas.ehNumero(str)) {
              val = Double.parseDouble(str);
     }
     System.out.println("Valor = " + val);
   }
}
```

3 –

```
public abstract class Calculadora {
   public static double dividir(double a, double b)
              throws Exception {
     double res = -1;
     if (b == 0)
```

```
                throw new Exception("Valor inapropriado – Zero!");
        else res = a/b;
        return res;
    }
}
public class TesteCalc {
    public static void main(String[] args) {
        double a = (double)5/(double)0;
        System.out.println(a);
        try {
                double x = Calculadora.dividir(6, 0);
                System.out.println(x);
        } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
        System.out.println("FIM DO PROGRAMA.");
    }
}
```

Esse teste terá como resultado (na tela da console):

```
Infinity
java.lang.Exception: Valor inapropriado – Zero!
FIM DO PROGRAMA.
    at Calculadora.dividir(Calculadora.java:6)
    at TesteCalc.main(TesteCalc.java:6)
```

Neste caso, inicialmente foi forçada uma conta de divisão por zero com valores do tipo double (para isso, fizemos um casting para double do valor zero – o que poderia ter sido feito simplesmente com o termo double a = 5.0 / 0;). Podemos observar que a execução não gerou uma exceção, e o resultado final foi o termo Infinity. Vê-se nesse programa que a chamada ao método dividir(...) está "obrigatoriamente" envolta numa estrutura try-catch; se não estivesse, o sistema não iria compilar o programa e consequentemente não iria rodá-lo.

4 –

```
import java.util.Random;
public class Contagem implements Runnable {
    private String id = "";
    public void run() {
        Random rndm = new Random();
        System.out.println("Iniciando para " + id + ":");
        for (int x = 1; x <= 100; x++) {
```

```
                System.out.println(id + ": " + x);
                // Dando um tempo de espera de valores diferentes
                // aleatoriamente variando entre 1 e 10
                // décimos de segundo
                try {
                        int num = rndm.nextInt(10) + 1;
                        Thread.sleep(num * 100);
                } catch (InterruptedException e) {
                }
        }
    }
    public Contagem(String s) {
      id = s;
    }
}
```

Veja o comando a seguir:

```
int num = rndm.nextInt(10) + 1;
Thread.sleep(num * 100);
```

Nele, inicialmente gera-se um número aleatório entre 1 e 10; depois, no método sleep, multiplica-se o valor por 100 para que o tempo de espera seja um número inteiro em décimos de segundos.

A exceção IllegalArgumentException é uma exceção bastante utilizada sempre que precisamos validar algum valor, a fim de que esteja dentro de determinados parâmetros aceitáveis, seja para o programa, seja para alguma regra de negócios da empresa em relação àquele sistema.

A única atenção a essa exceção é que, como ela é uma subclasse da RuntimeException, se lançarmos um método com essa exceção, o sistema não obrigará o tratamento com a estrutura try-catch quando aquele método for acionado (chamado). Nesse caso, o ideal é lançar a exceção IllegalArgumentException com o processo throw, mas lançar para o método a exceção Exception juntamente com o comando throws.

Resumo

Nesta unidade, vimos que algumas ações realizadas por métodos podem resultar em erros (ou exceções) que podem barrar o funcionamento do compilador, podendo inclusive interromper a execução de todas as aplicações de um servidor de aplicação Java. Para evitar que essa interrupção aconteça, nos utilizamos de uma estrutura de tratamento de exceções que, apesar de não evitar que a exceção aconteça, consegue evitar que haja a interrupção do sistema que está sendo rodado e consequentemente evita a interrupção do funcionamento do servidor.

Para isso, utilizamos uma estrutura de tentativa de execução e de captura de exceção, que é a estrutura try-catch. A sintaxe dessa estrutura é:

```
try {
  // códigos que podem gerar uma exceção (tenta executar)
} catch (Excecao01 e) {
  // bloco de tratamento realizado a partir da
  // identificação de uma exceção do tipo Excecao01
} catch (Excecao02 e) {
  // bloco de tratamento realizado a partir da
  // identificação de uma exceção do tipo Excecao02
...
} catch (ExcecaoN e) {
  // bloco de tratamento para exceção mais genérica
  // (geralmente Exception ou Throwable)
} finally {
  // bloco opcional mas que sempre será executado
  // independente da ocorrência ou não de uma exceção
}
```

Aqui, pode-se capturar e tratar diversas exceções, desde que estejam dispostas em ordem hierárquica (de herança), sendo a mais genérica por último, e nas quais o termo finally contém o bloco que será executado independentemente se a exceção acontecer ou não. O tratamento da exceção é a lógica de ação realizada (no bloco de código) caso a exceção aconteça.

Podemos forçar, para um método, o uso da estrutura de tratamento de exceções, utilizando os termos throw e throws (em conjunto), obrigando assim a criação de uma estrutura específica (obrigar uma atenção especial)

sempre que o desenvolvedor for se utilizar daquele método. De uma forma geral, a sintaxe do método que lança uma exceção é:

```
modificador tipoRetorno nomeMetodo(parametros) throws Excecao {
    ...
    throw new ExcecaoEspecifica (mensagem);
    ...
}
```

Podemos também criar exceções novas que especifiquem eventuais regras de negócios da empresa para a qual estamos gerando um sistema em Java. Para isso, basta criar uma classe, que por convenção tenha seu nome terminado com a palavra Exception e que herde a classe Exception. Esta classe criada deve ter um método construtor que acione o método construtor de sua classe mãe por meio do método super(), enviando como parâmetro uma string com a mensagem de exceção.

A thread é um recurso (uma classe Java) que permite que nosso sistema tenha multiprocessamento. Quando um programa exige a existência de multitarefa, como a maioria dos games, por exemplo, fazemos com que cada uma dessas tarefas seja rodada a partir de uma thread.

Uma thread (uma classe) pode ser criada por herança da classe Thread, ou implementando a interface Runnable (sendo esta última a forma mais utilizada), de modo que a lógica com as linhas de comando, que devem rodar em paralelo com outras, deve ser codificada no método run() dessa classe (método público e sem retorno – void).

A diferença entre uma forma ou outra está na hora de acionar o processo de rodagem em paralelo (acionar o método start()) das classes Thread criadas. Esse acionamento faz rodar o método run() das classes Thread. Desse modo, dependendo de como se criou a thread (se herdando a classe Thread ou implementando a interface Runnable), temos a seguinte sintraxe de acionamento:

```
Exemplo ex = new Exemplo();

// quando Exemplo herda Thread

ex.start();
```

Ou:

```
Exemplo ex = new Exemplo();

// quando Exemplo implementa Runnable

Thread t = new Thread(ex);

// ou...: Thread t = new Thread(ex, "Nome da Thread");

t.start();
```

Exercícios

Questão 1. (FGV/2018. Adaptada) Avalie o código Java mostrado a seguir:

```
public class X
{
    public static void main(String [] args)
    {
        try
        {
            falha();
            System.out.print("A1");
        }
        catch (RuntimeException ex)
        {
            System.out.print("A2");
        }
        catch (Exception ex1)
        {
            System.out.print("A3");
        }
        finally
        {
            System.out.print("A4");
        }
        System.out.print("A5");
    }
    public static void falha()
    {
        throw new RuntimeException();
    }
}
```

Ao ser executado, o programa produz:

A) A1A2A3

B) A1A2A3A4

C) A2A4

D) A2A4A5

E) A2A3A4A5

Resposta correta: alternativa D.

Análise da questão

Para analisar o código, deve-se lembrar como funciona uma estrutura try-catch-finally, responsável pela captura e pelo tratamento de exceções em Java. A estrutura é iniciada com um bloco try, que tenta executar comandos que podem gerar exceções. Depois, há um ou mais blocos catch, responsáveis por tratar exceções específicas. Ao fim da estrutura, pode haver um bloco finally, que sempre é executado (caso exista). Vamos, agora, à análise do código da questão.

Ao executar o bloco try, é acionado o método falha(), que é responsável pelo lançamento da exceção RuntimeException. Depois, a execução avança para o bloco catch responsável por tratar essa exceção, imprimindo "A2". O bloco finally, que sempre será executado, é responsável por imprimir "A4". Após esse bloco, já fora da estrutura try-catch-finally, a execução do código continua e é feita a impressão "A5". Portanto, o programa produz "A2A4A5".

Questão 2. (Copese – UFPI/2017. Adaptada). As unidades concorrentes em Java são métodos denominados run, cujo código pode estar em execução simultânea com outros métodos (de outros objetos) e com o método main. Uma das maneiras de definir uma classe com um método run é definir uma subclasse da classe Thread predefinida e substituir o método run.

Sobre a classe Thread da linguagem Java, avalie as afirmativas a seguir.

I – A classe provê diversos métodos para controle de execução de threads, sendo o método start responsável por iniciar a execução do thread.

II – O método sleep é utilizado para forçar um método a atrasar sua execução até que o método run de outro thread tenha completado sua execução.

III – O método yield, que não possui parâmetros, é um pedido do thread em execução para entregar o processador voluntariamente.

É correto o que se afirma em:

A) I, apenas.

B) II, apenas.

C) I e III, apenas.

D) II e III, apenas.

E) I, II e III.

Resposta correta: alternativa C.

Análise das afirmativas

I – Afirmativa correta.

Justificativa: uma das maneiras de se trabalhar com threads em Java é criar uma classe que herda a classe Thread. Para acionarmos uma ou várias threads, é necessário rodar o método start() que existe, por herança, em cada classe que implemente essa funcionalidade. Quando acionamos o start, estamos iniciando um processamento em paralelo.

II – Afirmativa incorreta.

Justificativa: o método sleep() da classe Thread faz com que o thread em questão fique suspenso, ou "adormecido", por um intervalo de tempo pré-determinado. Portanto, sua função não é atrasar a execução de um método até que outro thread tenha completado sua execução.

III – Afirmativa correta.

Justificativa: o método yield() da classe Thread é capaz de parar a execução do thread atual, dando uma "oportunidade" de execução para outros threads, de mesma prioridade, que estão aguardando. Mesmo com essa capacidade, o método yield() nem sempre interrompe uma execução. Caso não haja outros threads aguardando, ou eles sejam de prioridade mais baixa, o thread atual continuará a ser executado.

REFERÊNCIAS

Textuais

BARNES, D. J.; KÖLLING, M. *Programação orientada a objetos com Java*: uma introdução prática usando o BlueJ. São Paulo: Pearson Prentice Hall, 2004.

CAELUM. *Java e orientação a objetos* – curso FJ-11. [s.d]. Disponível em: https://cutt.ly/CTCBse7. Acesso em: 10 nov. 2021.

CLARO, D. B.; SOBRAL, J. B. M. *Programação em Java*. Florianópolis: Copyleft Pearson Education, 2008. Disponível em: https://cutt.ly/ITCBvIK. Acesso em: 7 ago. 2021.

DEITEL, P.; DEITEL, H. *Java*: como programar. São Paulo: Pearson Education do Brasil, 2017.

FURGERI, S. *Java 7*: ensino didático. 2. ed. São Paulo: Érica, 2012.

FURGERI, S. *Java 8*: ensino didático – desenvolvimento e implementação de aplicações. São Paulo: Érica, 2015.

HORSTMANN, C. *Conceitos de computação com Java*. 5. ed. Porto Alegre: Bookman, 2009.

HORSTMANN, C. S.; CORNELL, G. *Core Java, volume I*: fundamentos. São Paulo: Pearson Prentice Hall, 2010.

OLIVEIRA, A. P.; MACIEL, V. V. *Java na prática*: volume I. Viçosa: Departamento de Informática da Universidade Federal de Viçosa, 2002. Disponível em: https://cutt.ly/oTCBJoK. Acesso em: 2 ago. 2021.

ORACLE. *The Java™ Tutorials*. [s.d.]a. Disponível em: https://cutt.ly/YTDYEfo. Acesso em: 15 jul. 2021.

ORACLE. *The Java™ Tutorials*: formatting numeric print output. [s.d.]b. Disponível em: https://cutt.ly/2TDTFDr. Acesso em: 22 nov. 2021.

ORACLE. *The Java™ Tutorials*: using JAR files – the basics. [s.d.]c. Disponível em: https://cutt.ly/YTfIwV2. Acesso em: 21 out. 2021.

ORACLE. *Timeline of key Java milestones*. 2020. Disponível em: https://cutt.ly/LTDT8zY. Acesso em: 29 ago. 2021.

SCHILDT, H. *Java para iniciantes*: crie, compile e execute programas Java rapidamente. 6. ed. Porto Alegre: Bookman, 2015.

TIOBE. *Índice Tiobe*. [s.d.]. Disponível em: https://cutt.ly/wTa0X8E. Acesso em: 11 nov. 2021.

WINDER, R. *Desenvolvendo software em Java*. Rio de Janeiro: LTC, 2009.

Interativa

Informações:
www.sepi.unip.br ou 0800 010 9000