PO-P03

1. O que é uma exceção em Java e qual é o propósito de usá-las em programas?

Em Java, uma exceção é um mecanismo que permite lidar com condições excepcionais ou erros durante a execução de um programa. Elas representam situações imprevistas ou excepcionais que podem ocorrer durante a execução do código.

O propósito principal das exceções é fornecer um mecanismo para lidar com erros e situações indesejadas de forma controlada e estruturada, permitindo que o código se recupere ou lide com essas situações de maneira adequada, em vez de simplesmente interromper abruptamente a execução do programa.

Algumas situações em que as exceções são úteis incluem:

Tratamento de erros:

Exceções permitem que você identifique e trate erros, como divisões por zero, acesso a recursos indisponíveis, tentativas de operações inválidas, entre outros.

Separação do código de tratamento de erros:

O código que lida com exceções pode ser separado do código funcional principal, facilitando a legibilidade e a manutenção do código.

Recuperação de falhas:

Por meio do tratamento de exceções, você pode implementar estratégias para lidar com falhas, como tentar novamente operações, liberar recursos adequadamente ou fornecer informações úteis para o usuário.

Hierarquia de exceções:

Em Java, as exceções são organizadas em uma hierarquia de classes, o que permite tratar diferentes tipos de exceções de maneira específica, capturando-as e tratando-as conforme necessário.

Notificação e registro de erros:

As exceções também podem ser usadas para registrar erros e notificar sistemas ou usuários sobre problemas que ocorreram durante a execução do programa.

No geral, o uso de exceções em Java proporciona um mecanismo robusto para lidar com situações inesperadas, ajudando a escrever código mais seguro, controlado e resiliente.

2. Pesquise sobre a diferença entre exceções verificadas e não verificadas em

Java. Dê exemplos de cada uma.

Em Java, as exceções são divididas em duas categorias principais: exceções verificadas (checked exceptions) e exceções não verificadas (unchecked exceptions). A diferença fundamental entre elas está na obrigatoriedade do tratamento em tempo de compilação.

Exceções Verificadas (Checked Exceptions):

Definição:

São exceções que são obrigatórias de serem tratadas ou declaradas pelo desenvolvedor em tempo de compilação.

Normalmente, são subclasses da classe Exception, exceto por subclasses de RuntimeException e suas subclasses.

Propósito:

Essas exceções são usadas para representar situações que o programa pode prever e recuperar-se. Exigem que o código cliente trate ou declare explicitamente a exceção.

Exemplo:

```
import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;


public class ExemploCheckedException {

   public static void main(String[] args) {

      try {

         BufferedReader reader = new BufferedReader(new FileReader("arquivo.txt"));

         String linha = reader.readLine();

         // ... (restante do código)

         reader.close();

      } catch (IOException e) {

         e.printStackTrace();

      }

   }

}
```

No exemplo acima, IOException é uma exceção verificada, forçando o tratamento ou a declaração da exceção na cláusula catch.

Exceções Não Verificadas (Unchecked Exceptions):

Definição:

São exceções que o compilador Java não exige que sejam tratadas ou declaradas em tempo de compilação.

São subclasses de RuntimeException ou suas subclasses.

Propósito:

Representam erros mais graves, como erros de lógica, divisão por zero, acesso a índices inválidos em arrays, entre outros.

Exemplo:

```
public class ExemploUncheckedException {

   public static void main(String[] args) {

      int[] array = {1, 2, 3};

      System.out.println(array[3]); // Isso causará ArrayIndexOutOfBoundsException

   }

}
```

ArrayIndexOutOfBoundsException é uma exceção não verificada que ocorre quando se tenta acessar um índice inválido em um array.

Em resumo, a diferença principal entre as exceções verificadas e não verificadas em Java é que as verificadas requerem tratamento ou declaração explícita em tempo de compilação, enquanto as não verificadas não exigem essa obrigatoriedade, geralmente representando problemas mais graves ou erros no código que podem não ser previstos durante a execução.

3. Como você pode lidar com exceções em Java? Quais são as palavras-chave e as práticas comuns para tratamento de exceções?

Em Java, o tratamento de exceções é feito utilizando blocos try-catch-finally, juntamente com algumas palavras-chave e práticas comuns para garantir um tratamento adequado das exceções. Aqui estão as principais palavras-chave e práticas para lidar com exceções em Java:

Palavras-chave:

try:

Usada para envolver o código que pode gerar uma exceção.

Bloco onde a exceção pode ocorrer.

catch:

Usada para capturar e tratar exceções específicas que ocorrem no bloco try.

Um ou mais blocos catch podem ser associados a um bloco try para tratar diferentes tipos de exceções.

finally:

Opcionalmente usado após um bloco try-catch para definir código que será executado independentemente se houver uma exceção ou não.

É frequentemente usado para liberar recursos (como fechar arquivos, conexões com banco de dados, etc.).

Práticas Comuns:

Identificar e Tratar Exceções Específicas:

Utilize blocos catch para capturar exceções específicas (herdadas de Exception) que você espera ou pode lidar de maneira adequada.

Trate exceções de forma apropriada para o contexto do seu programa.

Encadeamento de Exceções:

Use a palavra-chave throws nos métodos para propagar exceções não tratadas para quem os chama, se apropriado.

Use o operador throw para lançar exceções personalizadas ou propagar exceções em blocos catch.

Bloco finally:

Use o bloco finally para garantir a execução de código crítico, como liberação de recursos, independentemente de exceções terem ocorrido ou não.

Tratamento de Exceções Genéricas:

Você pode capturar exceções genéricas usando catch(Exception e) para lidar com qualquer exceção que não seja tratada de forma mais específica.

Exceções Personalizadas:

Crie exceções personalizadas estendendo as classes Exception ou RuntimeException para casos específicos do seu programa.

Exemplo:

```
import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;


public class ExemploTratamentoExcecoes {

   public static void main(String[] args) {

      BufferedReader reader = null;

      try {

         reader = new BufferedReader(new FileReader("arquivo.txt"));

         String linha = reader.readLine();

         // ... (restante do código)

      } catch (IOException e) {

         System.out.println("Erro de leitura: " + e.getMessage());

      } finally {

         try {

            if (reader != null) {

               reader.close(); // Liberar recursos no bloco finally

            }

         } catch (IOException e) {

            System.out.println("Erro ao fechar o arquivo: " + e.getMessage());

         }

      }

   }

}
```

Neste exemplo, o bloco try-catch é usado para ler um arquivo. O bloco finally é usado para garantir que o recurso (nesse caso, o arquivo) seja fechado independentemente de uma exceção ter ocorrido ou não durante a leitura. Isso é útil para garantir que recursos sejam liberados corretamente mesmo em caso de exceções.

4. O que é o bloco "try-catch" em Java? Como ele funciona e por que é importante usá-lo ao lidar com exceções?

O bloco try-catch em Java é uma estrutura utilizada para lidar com exceções durante a execução de um código. Ele permite que você envolva um bloco de código que pode gerar exceções em um contexto controlado. Aqui está uma explicação detalhada:

Funcionamento do bloco try-catch:

try:

O bloco try é usado para envolver o código onde uma exceção pode ocorrer.

Qualquer exceção lançada dentro deste bloco é monitorada.

catch:

O bloco catch é usado para capturar e tratar exceções específicas que foram lançadas no bloco try. Permite que você especifique o tipo de exceção que deseja capturar e forneça código para lidar com essa exceção. Pode ter vários blocos catch para diferentes tipos de exceções. Por que é importante usá-lo ao lidar com exceções?

Tratamento Controlado de Exceções:

O uso do try-catch permite que você trate exceções de maneira controlada, evitando que elas interrompam abruptamente a execução do programa.

Evita a Interrupção do Fluxo do Programa:

Ao capturar exceções, você pode fornecer um comportamento alternativo ou mensagem de erro ao usuário em vez de deixar o programa encerrar inesperadamente.

Recuperação de Erros:

Permite a recuperação de erros conhecidos, como tentativas de acessar recursos indisponíveis, divisões por zero, entre outros.

Liberar Recursos:

É comumente utilizado para liberar recursos alocados, como fechar arquivos, conexões com banco de dados, etc., garantindo que esses recursos sejam liberados independentemente de exceções.

Exemplo:

```
import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;


public class ExemploTryCatch {

   public static void main(String[] args) {

      BufferedReader reader = null;

      try {

         reader = new BufferedReader(new FileReader("arquivo.txt"));
```

```
            String linha = reader.readLine();
            // ... (restante do código)
        } catch (IOException e) {
            System.out.println("Erro de leitura: " + e.getMessage());
        } finally {
            try {
                if (reader != null) {
                    reader.close(); // Liberar recursos no bloco finally
                }
            } catch (IOException e) {
                System.out.println("Erro ao fechar o arquivo: " + e.getMessage());
            }
        }
    }
}
```

Neste exemplo, o bloco **try** envolve a leitura de um arquivo. Se ocorrer uma exceção durante a leitura, o bloco **catch** captura a exceção e trata dela, imprimindo uma mensagem de erro. O bloco **finally** é usado para garantir que o recurso (o arquivo) seja fechado, independentemente de exceções terem ocorrido ou não durante a leitura, assegurando a liberação correta dos recursos.

5. Quando é apropriado criar suas próprias exceções personalizadas em Java e como você pode fazer isso? Dê um exemplo de quando e por que você criaria uma exceção personalizada.

É apropriado criar suas próprias exceções personalizadas em Java quando você está lidando com situações específicas do seu domínio de aplicação que não são representadas adequadamente pelas exceções padrão fornecidas pelo Java. Criar exceções personalizadas pode tornar o código mais legível, manutenível e fornecer informações mais significativas sobre erros específicos em seu programa.

Como criar uma exceção personalizada:

1 - Criar uma classe que estende Exception ou RuntimeException:

Você pode criar sua própria classe de exceção estendendo a classe Exception (para exceções verificadas) ou RuntimeException (para exceções não verificadas).

2 - Adicionar construtores e métodos, se necessário:

Adicione construtores para definir mensagens de erro personalizadas ou outros dados relevantes à exceção.

Você também pode adicionar métodos adicionais conforme necessário para fornecer funcionalidades específicas à sua exceção.

Exemplo de criação de exceção personalizada:

Suponha que você esteja desenvolvendo um sistema de gerenciamento de pedidos e deseje criar uma exceção para lidar com a situação em que um pedido não pode ser processado devido a um problema de estoque insuficiente.

```
public class EstoqueInsuficienteException extends Exception {
    public EstoqueInsuficienteException() {
        super("Não há estoque suficiente para processar este pedido.");
    }

    public EstoqueInsuficienteException(String message) {
        super(message);
    }
}
```

Quando criar uma exceção personalizada:

Cenários de Domínio Específico:

Quando você está lidando com situações específicas do seu domínio de aplicação que não são bem representadas pelas exceções existentes.

Informações Adicionais Necessárias:

Quando você precisa fornecer informações adicionais ou contextuais sobre um erro específico que não podem ser transmitidas por meio das exceções padrão.

Facilidade de Identificação de Erros:

Para facilitar a identificação de problemas e erros específicos que ocorrem dentro do seu programa.

No exemplo acima, a exceção **EstoqueInsuficienteException** é criada para representar o cenário em que não há estoque suficiente para processar um pedido. Isso permite que o código que manipula pedidos lance essa exceção quando necessário, fornecendo informações específicas sobre o problema de estoque insuficiente. Essa personalização ajuda na identificação e tratamento preciso desse tipo específico de situação de erro.