LEANDRO GARCIA MAGALHÃES CERQUEIRA

PO3 - JAVA

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

R: Uma exceção como o próprio nome diz, é um evento que pode ou não ocorrer devido a um erro durante a execução do programa, interrompendo o seu fluxo normal. Essas exceções podem ser provenientes de erros de lógica ou tentativas de acesso de dispositivos/arquivos/espaços de memória externos. É importante salientar que em Java, que é uma linguagem orientada a objetos, exceções também são objetos de uma classe. O tratamento de exceções é importante porque permite que o programa lide com erros de forma mais elegante e robusta. Sem o tratamento adequado, o programa pode simplesmente parar de funcionar ou exibir mensagens de erro confusas para o usuário. O tratamento de exceções permite que o programa exiba mensagens de erro mais claras e precisas, além de permitir que o programa continue a executar mesmo após um erro ter ocorrido.

2. Pesquise sobre a diferença entre exceções verificadas e não verificadas em Java. Dê exemplos de cada uma.

R: Em Java, as exceções são divididas em dois tipos principais: exceções verificadas (checked exceptions) e exceções não verificadas (unchecked exceptions).

Exceções verificadas são aquelas que o compilador obriga o programador a tratá-las, seja através de um bloco try-catch ou declarando que o método pode lançar essa exceção usando a palavra-chave “throws”. Normalmente, essas exceções estendem, a classe “Exception” (ou suas subclasses) diretamente, exceto “RunTimeException” e suas subclasses.

Exemplos de classe tratada:

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ExcecaoVerificadaExemplo {
    public static void main(String[] args) {
        try {
            File arquivo = new File("arquivo.txt");
            Scanner leitor = new Scanner(arquivo);
            while (leitor.hasNextLine()) {
                System.out.println(leitor.nextLine());
            }
            leitor.close();
        } catch (FileNotFoundException e) {
            System.out.println("Arquivo não encontrado: " + e.getMessage());
        }
    }
}
```

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ExcecaoVerificadaExemplo {

    // Método que lê dados de um arquivo
    public static void lerArquivo(String nomeArquivo) throws FileNotFoundException {
        File arquivo = new File(nomeArquivo);
        Scanner leitor = new Scanner(arquivo);
        while (leitor.hasNextLine()) {
            System.out.println(leitor.nextLine());
        }
        leitor.close();
    }

    public static void main(String[] args) {
        String nomeDoArquivo = "arquivo.txt";

        try {
            lerArquivo(nomeDoArquivo);
        } catch (FileNotFoundException e) {
            System.out.println("Arquivo não encontrado: " + e.getMessage());
        }
    }
}
```

As exceções não verificadas são subclasses de “RuntimeException” e não precisam ser tratadas obrigatoriamente pelo programador. Elas podem ser capturadas usando um bloco try-catch, mas não é exigido pelo compilador. Exemplos de exceções não verificadas incluem “NullPointerException”, ArrayIndexOutOfBoundsException”, “ArithmeticException”, “IllegalArgumentException”.

Exemplo de classe não-verificada:

```
public class ExcecaoNaoVerificadaExemplo {
    public static void main(String[] args) {
        String str = null;

        // Tentativa de acessar o comprimento da string
        try {
            int length = str.length(); // Isso resultará em NullPointerException
            System.out.println("Comprimento da string: " + length);
        } catch (NullPointerException e) {
            System.out.println("Ocorreu uma exceção: " + e.getMessage());
        }

        // Alternativamente, sem try-catch, o programa terminará se a exceção ocorrer
        int length = str.length(); // Isso resultará em NullPointerException
        System.out.println("Comprimento da string: " + length); // O programa será er
    }
}
```

Portanto, a diferença fundamental entre exceções verificadas e não verificadas está na obrigatoriedade do tratamento pelo programador. Não é sobre a certeza de ocorrer ou não, mas sim sobre como o compilador trata essas exceções e como é exigido o tratamento delas no código-fonte.

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

try-catch-finally:

try: Define um bloco de código onde uma exceção pode ocorrer.
catch: Captura e trata a exceção lançada no bloco try.
finally: Bloco opcional que é executado sempre, independentemente de uma exceção ser lançada ou não. É usado para liberar recursos, como fechar arquivos abertos ou conexões com bancos de dados.

```
try {
    // Código que pode lançar uma exceção
} catch (ExcecaoEspecifica e) {
    // Tratamento da exceção específica
} catch (OutraExcecao e) {
    // Tratamento de outra exceção
} finally {
    // Código que sempre será executado
}
```

throws:

A palavra-chave throws é usada na assinatura do método para declarar que um método pode lançar uma exceção verificada e transferir a responsabilidade de tratamento para o código que chama esse método.

```
public void meuMetodo() throws MinhaExcecao {
    // Código que pode lançar MinhaExcecao
}
```

Throw:

A palavra-chave throw é usada para explicitamente lançar uma exceção em um determinado ponto do código.

```
if (condicao) {
    throw new MinhaExcecao("Mensagem de erro");
}
```

Múltiplos catch e exceções encadeadas:

É possível ter vários blocos catch para lidar com diferentes tipos de exceções.

Também é possível encadear exceções, onde uma exceção capturada pode ser relançada após tratamento.

```
try {
    // Código que pode lançar diferentes exceções
} catch (ExcecaoTipoA e) {
    // Tratamento para ExcecaoTipoA
} catch (ExcecaoTipoB e) {
    // Tratamento para ExcecaoTipoB
    throw new MinhaExcecao("Mensagem", e); // Relança uma nova exceção encadeada
}
```

Boas práticas:

- Capturar exceções de forma específica e tratar de acordo com o contexto.
- Fechar recursos abertos no bloco `finally` para garantir que sejam liberados, mesmo se uma exceção for lançada.
- Evitar capturar exceções muito genéricas, como `Exception`, sempre que possível.
- Usar exceções para condições excepcionais e não para controle de fluxo normal do programa.

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.

R: É apropriado criar uma exceção própria e personalizada quando você está lidando com situações específicas dentro do seu domínio de aplicação que não são adequadamente representadas pelas exceções existentes na biblioteca Java. Isso permite que você forneça informações mais contextuais sobre o erro que ocorreu, facilitando o entendimento e o tratamento adequado em sua aplicação. Um exemplo de quando eu precisaria criar uma exceção personalizada seria se eu

estivesse desenvolvendo um sistema de gerenciamento de biblioteca e precisasse lidar com a situação em que um livro está indisponível no momento em que um usuário tenta emprestá-lo. Para resolver isso eu criaria uma exceção personalizada para representar essa situação.

Criação da exceção:

```
// Definição da exceção personalizada
public class LivroIndisponivelException extends Exception {
    public LivroIndisponivelException() {
        super("O livro está indisponível no momento.");
    }

    public LivroIndisponivelException(String mensagem) {
        super(mensagem);
    }
}
```

Utilizando a exceção que criei:

```
public class Biblioteca {
    private boolean livroDisponivel = false;

    public void emprestarLivro() throws LivroIndisponivelException {
        if (!livroDisponivel) {
            throw new LivroIndisponivelException();
        }
        // Lógica para emprestar o livro...
    }

    // Outros métodos da classe Biblioteca...
}
```