**RESUMO DA AULA 09 - EXCEPTIONS**

ALUNO: Denilson José do Bom Jesus Silva de Lima

**- ROBUSTEZ**

Sistemas robustos fornecem formas eficientes para recuperação de falhas e mensagens apropriadas para erros. Fazem validação dos dados (dados condizentes com o mundo real) e garantem a consistência das operações (retornar a um estado seguro e estável e permitir ao usuário salvar o trabalho realizado).

**- EXCEÇÕES**

Mecanismos para lidar com erros ou situações indesejadas (Ex: divisão por zero); são objetos que encapsulam informações relevantes sobre o erro ocorrido (tipo de objeto que estende uma classe em Java chamado exceptions). Representam erro, mas evitam o tratamento deste erro e são indicados para os lugares onde podem ocorrer (só lança exceção se necessário, não como regra). (Ex:)

public class Saldo extends Exception{

public Saldo(int a, double b) {

super("Conta com saldo "+ a + "e saldo " + b);

}

public class Classe {

public void debitar(int valor) throws Saldo {

if(a <= valor) {

this.valor = this.valor + a;

} else {

Saldo s = new Saldo(this.a, this.b);

throw s;

}

}

}

A exceção é criada pelo usando "extends Exception", é declarada em um método pela palavra reservada "throws" (pode declarar mais de uma) e é lançada com "throw".

Não é possível ignorar uma exceção caso ela tenha sido implementada, se um método lança uma exceção é possível fazer duas coisas com ele: repassar a exceção ou tratar a exceção. Para repassar, é necessário declarar na assinatura do método (no exemplo, de um método que chama outro método que possui uma exceção implementada) que ele lança a mesma exceção. Exceções repassadas recebe automaticamente, porém repassa para outro. (Ex:)

public class Classe {

[...]

public void transferir(final Conta destino) throws Saldo {

this.debitar(valor);

}

[...]

}

Ou é possível tratar a exceção com \*try-catch [-finally]\*, que é quando se deseja tomar alguma providência caso uma exceção ocorra. O finally é opcional e sempre é

executado, independente se aconteceu a execução ou não. (Ex:)

public class Principal {

public static void main(String[] args){

Classe Classe1 = new Classe();

try {

Conta1.debitar(10);

} catch (Saldo s) {

System.err.println("Debito não realizado");

} catch (Exception s) {

}

}

}

Quando a exceção acontece, o estado do sistema até o momento antes do acontecimento do erro é guardado, a exceção é passada para o fluxo de controle que executa algo.

Uma dica: nunca deixar o bloco do catch vazio, mesmo que a execução nunca venha a ocorrer. O ideal é tratar diferentes execuções num só bloco (um único "try" com vários "catch"). Ideal também tratar primeiro as exceções mais específicas.

Em suma, exceções podem ser:

- Declaradas: indicadas onde podem ocorrer;

- Lançadas: Passadas para outro lugar (criadas explicitamente e repassadas) e;

- Tratadas: capturadas para se tomar providências;

**- HIERARQUIA DE EXCEÇÕES**

Throwable <- [Error]

↑

[IOException] -> [Exception] <- [RuntimeException]

-Erros de I/O ↑ - Erro de Cast

- URL [Execuções do Usuário] - Limites de Array e NullPointer

- RuntimeException: Não obriga a repassar ou tratar a exceção - geralmente ocorrem devido a má codificação (Ex:)

public class Saldo extends RuntimeException {

public static void main(String[] args){

Classe Classe1 = new Classe();

Conta1.debitar(10);

}

}

**- ALGUNS MÉTODOS**

- getMessage();

- printStackTrace();

**- EXCEÇÕES DO USUÁRIO**

Criadas para representar os erros que podem acontecer no sistema: erros de negócio e erros de persistência.

**- CHECKED E UNCHECKED EXCEPTIONS**

Checked -> Deve ser tratada ou lançada pelo programa.

Unchecked -> Não precisa ser capturada pelo programa e é pertencente a alguma subclasse de RunTimeException.