


09.Tratamento de Exceções

Prof. Alexandre Krohn




Roteiro

- **Introdução**
- Disparo de Exceções
- Captura de Exceções
- Criação de Exceções
- Bloco finally
- Exercícios




Introdução à Exceções

- Mundo ideal: dados estão sempre na forma certa, arquivos desejados sempre existem, etc.
- Mundo real: dados ruins e bugs podem arruinar o programa.




Introdução à Exceções

- Necessidade de mecanismos para tratamento de erros.
- Antes da POO:
 - Variável global inteira com valores de 0 até n.
 - Na ocorrência de uma exceção:
 - Variável assumia um valor.
 - Remetia uma mensagem de erro.
 - Encerrava o programa.




Introdução à Exceções

- Depois da POO:
 - Classes de erros.
 - Possíveis tipos de erros e seus tratamentos são agrupados.
 - Não há necessidade de interromper o programa.
 - O mesmo erro é tratado quantas vezes for necessário.



Introdução à Exceções

- Idéia básica: “código ruim não será executado”.
- Nem todos os erros podem ser detalhados em tempo de compilação.
- Os que não podem devem ser lidados em tempo de execução.
- Estes últimos são o alvo da manipulação de exceções.



Introdução à Exceções

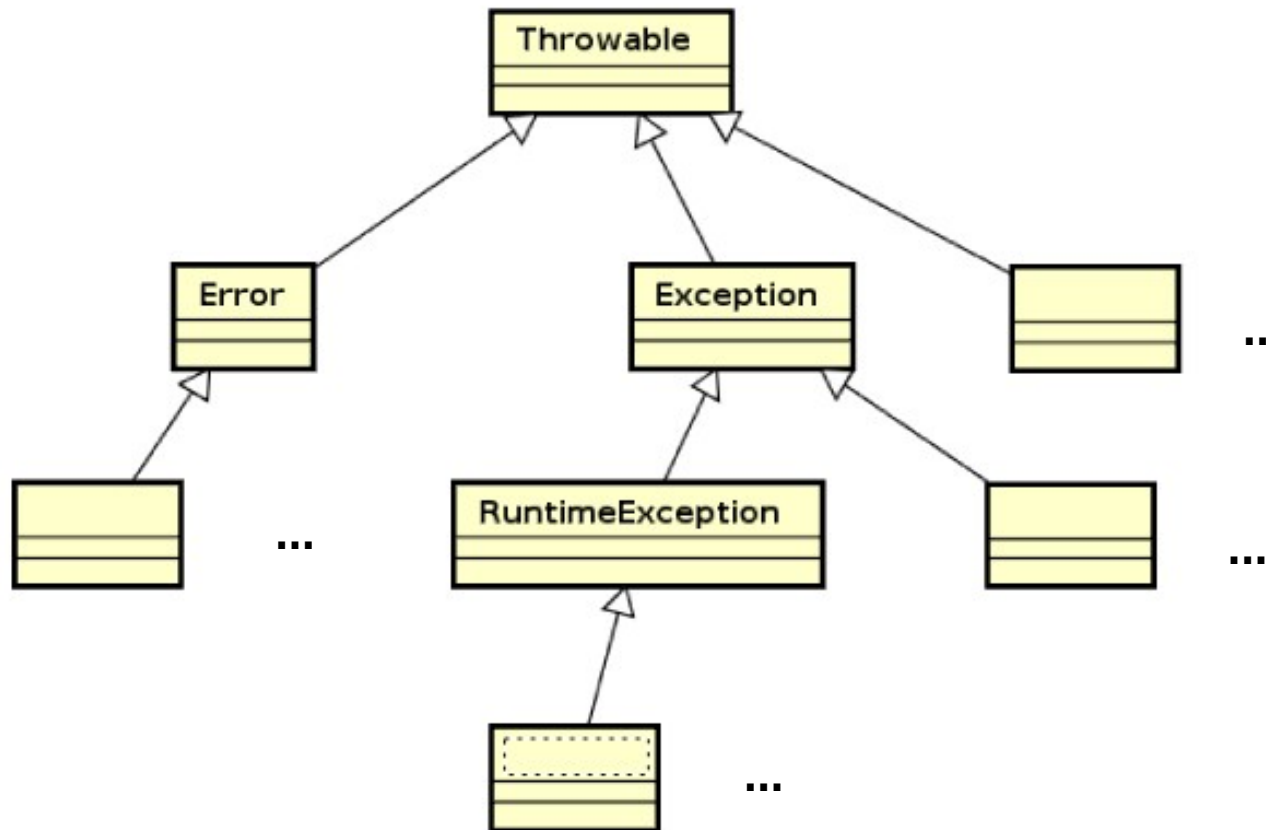
- **Premissa básica:** separar o processamento normal da manipulação de erros.
- Vantagens desse mecanismo:
 - Permite concentrar em lugares diferentes o “código normal” do tratamento do erro.
- Simplifica a criação de programas grandes usando menos código.
- Torna o código mais robusto, ao garantir que não há erros sem tratamento.



Exceções : Básico

- Exceções são classes em Java
- Assim sendo, seguem o princípio de herança, com alguns tipos de exceções especializadas

Hierarquia de Exceções





Roteiro

- Introdução
- **Disparo de Exceções**
- Captura de Exceções
- Criação de Exceções
- Bloco finally
- Exercícios



Exceções

- **Exceção:** problema que impede a continuação do método ou escopo em execução.
- Importante: exceção \neq problema normal.
- Problema normal: há informação suficiente no contexto atual para lidar com ele.
- Exceção: não há informação suficiente.
- **Disparar uma exceção:** sair do contexto atual e delegar a solução a um contexto mais abrangente.



Exceções

- Ao disparar-se uma exceção, ocorre a seguinte sequência de eventos:
 - Um objeto exceção é criado.
 - A execução é interrompida.
 - O mecanismo de manipulação de exceções assume o controle e procura o manipulador de exceção adequado.
 - O manipulador da exceção trata o problema.

Exceções

- Exemplo: seja *t* uma referência para um objeto, que pode não ter sido inicializado.

```
if (t == null)
    throw new NullPointerException();
```

- A palavra chave ***throw*** dispara uma exceção e dá início à sequência de eventos citada anteriormente.

Exceções : Outro exemplo

```
if (t == null)
    throw new NullPointerException("t = null");
```

- Este construtor permite colocar informações pertinentes na exceção, que posteriormente podem ser extraídas usando outros métodos.

Disparando exceções:

- Em resumo, para se disparar uma exceção, segue-se os seguintes passos:
 - 1) Escolha uma classe de exceção apropriada.
 - 2) Instancie um objeto dessa classe.
 - 3) Dispare-o.

3) 2) 1)
`throw new NullPointerException();`



Roteiro

- Introdução
- Disparo de Exceções
- Captura de Exceções
- Criação de Exceções
- Bloco finally
- Exercícios



Capturando exceções

- Quando uma exceção é disparada, em algum lugar ela deve ser capturada.
- **Região protegida:** trecho de código que pode gerar exceções.
- **Manipuladores de exceções:** tratam as exceções que ocorreram dentro da região protegida. Vêm imediatamente após a mesma.

try/catch

```
try {
```

Região protegida : Bloco onde pode ocorrer uma exceção

```
} catch (ClasseDeExcecao c) {
```

Manipulador da Exceção : Bloco de código Com o tratamento para a exceção que ocorreu

```
}
```

Classe de Exceção :

Qual o tipo de exceção que será direcionada para o manipulador abaixo.

Variável da Exceção :

Referência para a exceção ocorrida, de onde se pode obter mais informações sobre a mesma.



Múltiplos manipuladores

- Pode-se usar vários manipuladores:

```
try { ...
```

```
} catch(ClasseDeExcecao1 c1) {
```

```
...
```

```
} catch(ClasseDeExcecao2 c2) {
```

```
...
```

```
} catch(ClasseDeExcecao3 c3) {
```

```
...
```

```
}
```

Neste caso, cada exceção específica é tratada pelo seu próprio manipulador



Capturando Exceções

- Processo: A exceção é disparada dentro de um bloco *try*.
- O mecanismo de manipulação de exceção procura o primeiro *catch* cujo tipo de exceção bata com a exceção disparada.
- O mecanismo entra no bloco do *catch* e o erro é tratado.

Exemplo:

```
public class Teste {  
  
    public static void main(String[] args) {  
  
        char[] vetor = {'a', 'b', 'c', 'd', 'e'};  
  
        for(int i = 0; i < 10; i++) {  
  
            try {  
                System.out.println(i + " : " + vetor[i]);  
            } catch(ArrayIndexOutOfBoundsException e) {  
                System.out.println("Estourado o limite do vetor");  
                System.out.println("na posição " + i);  
                break;  
            }  
        }  
    }  
}
```

Exemplo:

```
public class Teste {  
    public static void main(String[] args) {  
        char[] vetor = {'a', 'b', 'c', 'd', 'e'};  
        for(int i = 0; i < 10; i++) {  
            try {  
                System.out.println(i + " : " + vetor[i]);  
            } catch(ArrayIndexOutOfBoundsException e) {  
                System.out.println("Estourado o limite do vetor");  
                System.out.println("na posição " + i);  
                break;  
            }  
        }  
    }  
}
```

Saída:

0 : a
1 : b
2 : c
3 : d
4 : e

Estourado o limite do vetor
na posição 5



Roteiro

- Introdução
- Disparo de Exceções
- Captura de Exceções
- Criação de Exceções
- Bloco finally
- Exercícios



Criando Exceções

- Cria-se uma exceção extendendo a classe ***Exception***.
- A classe ***Exception*** possui uma série de **constructores** que devem ser sobrescritos.



Exemplo

- Vamos definir uma exceção para uma operação de saque em uma conta bancária.
- Essa exceção deverá ocorrer sempre que se tentar um saque de saldo maior que o disponível na conta;



Exemplo : ***SaldoInsuficienteException***

```
package conta;
```

```
public class SaldoInsuficienteException extends Exception {
```

```
    public static final String MESSAGE = "O saldo é insuficiente para a operação de saque!";
```

```
    public SaldoInsuficienteException() {  
        super(MESSAGE);  
    }
```

```
    public SaldoInsuficienteException(String message) {  
        super(MESSAGE + " " + message);  
    }
```

```
    public SaldoInsuficienteException(Throwable cause) {  
        super(MESSAGE, cause);  
    }
```

```
    public SaldoInsuficienteException(String message, Throwable cause) {  
        super(MESSAGE + " " + message, cause);  
    }
```

```
    public SaldoInsuficienteException(String message, Throwable cause, boolean enableSuppression,  
        boolean writableStackTrace) {  
        super(MESSAGE + " " + message, cause, enableSuppression, writableStackTrace);  
    }
```

```
}
```

Pode-se utilizar sempre esta estrutura, mudando apenas o nome da classe e o texto da mensagem (MESSAGE)

Exemplo : *ContaBancaria*

```
package conta;
```

```
public class ContaBancaria {
```

```
    private String numero;  
    private double saldo;
```

```
    public ContaBancaria() {..}
```

```
    public ContaBancaria(String numero) {..}
```

```
    public String getNumero() {..}
```

```
    public void setNumero(String numero) {..}
```

```
    public double getSaldo() {..}
```

```
    public void depositar(double valor) {  
        saldo += valor;  
    }
```

```
    public void sacar(double valor) throws SaldoInsuficienteException {  
        if(valor > saldo) {  
            throw new SaldoInsuficienteException();  
        }  
        saldo -= valor;  
    }  
}
```

O método que pode gerar uma exceção tem que declarar a exceção através da cláusula ***throws***

Exemplo : *main*

```
public static void main(String[] args) {  
  
    ContaBancaria c1 = new ContaBancaria("123-4");  
    c1.depositar(1500);  
  
    try {  
        c1.sacar(1000);  
        System.out.println("Saque efetuado com sucesso!");  
    } catch (SaldoInsuficienteException e) {  
        double saldo = c1.getSaldo();  
        System.out.println("O saque não foi possível");  
        System.out.println("Você só tem R$ " + saldo);  
    }  
}
```

Programação Ori

<terminated> Principal (2) [Java Applica
Saque efetuado com sucesso!

Exemplo : *main*

```
public static void main(String[] args) {  
  
    ContaBancaria c1 = new ContaBancaria("123-4");  
    c1.depositar(500);  
  
    try {  
        c1.sacar(1000);  
        System.out.println("Saque efetuado com sucesso!");  
    } catch (SaldoInsuficienteException e) {  
        double saldo = c1.getSaldo();  
        System.out.println("O saque não foi possível");  
        System.out.println("Você só tem R$ " + saldo);  
    }  
}
```



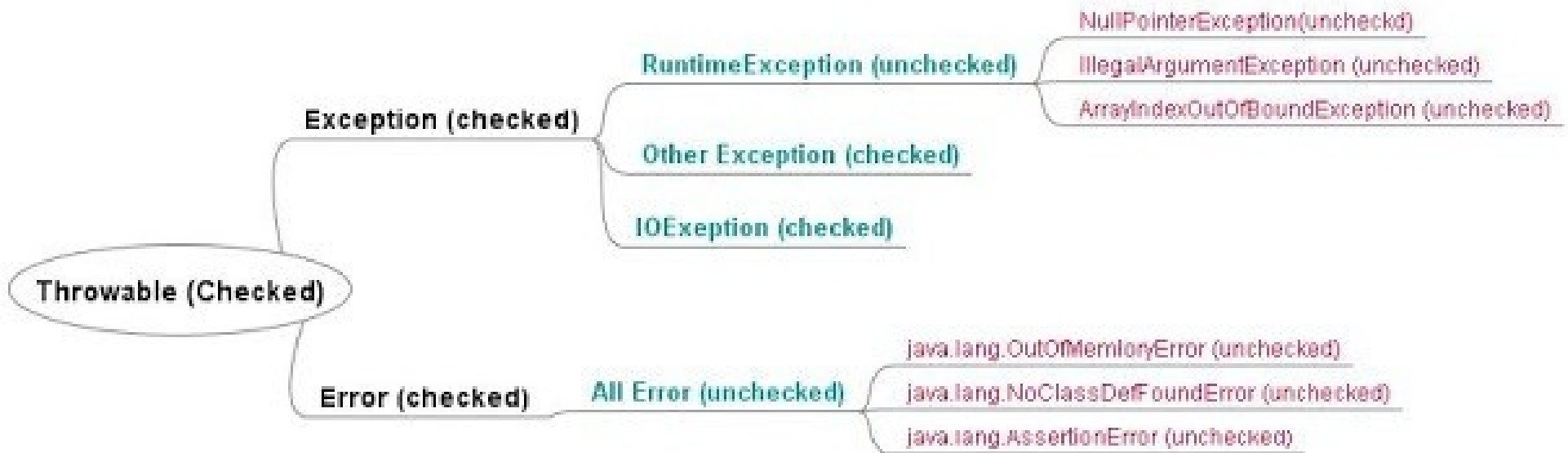
<terminated> Principal (2) [Java A

Programação Orientada a Objetos

O saque não foi possível

Você só tem R\$ 500.0

Exceções checadas e não checadas



Exceções checadas e não checadas

- Dependendo de qual “ramo” da hierarquia de throwable for herdado, o comportamento da aplicação será diferente.
- Há três tipos principais de exceções:
 - Exception
 - RuntimeException
 - Error



Exception

- Exceções que herdam de **Exception** são exceções **checadas**.
- Isso significa que sempre que as mesmas podem ocorrer, o código que tem potencial para gerá-las deve ficar dentro de um bloco ***try/catch***



Exception

- Nossa exceção ***SaldoInsuficienteException*** é uma exceção checada.
- Outros exemplos são todas as exceções de “regra de negócio” que podemos implementar



RuntimeException

- Exceções que herdam de **RuntimeException** são exceções **não checadas**.
- Nesse caso, **não** somos obrigados a usar o bloco **try/catch**, mas se ocorrer uma exceção ela pode “quebrar” o programa



RuntimeException

- Como exemplos de exceções não checadas, temos:
 - `ArrayIndexOutOfBoundsException`
 - `NullPointerException`
 - `NumberFormatException`

Lembre-se sempre : A ocorrência de RuntimeException indica a existência de
Erro de programação!



Error

- Exceções do tipo Error são aquelas fatais, das quais não há uma maneira de fazer o sistema se recuperar:
- Exemplos:
 - OutOfMemoryException
 - NoClassDefFoundException
- Estão relacionadas com a **indisponibilidade de recursos** no computador



Roteiro

- Introdução
- Disparo de Exceções
- Captura de Exceções
- Criação de Exceções
- **Bloco finally**
- Exercícios



Bloco finally

- Frequentemente existe algum trecho de código que deve ser executado independente de uma exceção ter ou não ter sido disparada.
- Problema: se o método alocou um recurso e uma exceção foi disparada, o recurso pode não ter sido liberado.



Bloco finally

- Apesar de Java possuir Garbage Collector, pode ser necessário retornar algum recurso não relacionado a memória para seu estado original.
- Exemplos: fechar um arquivo, fechar uma conexão de rede, redesenhar algum componente na tela, etc.
- Para isso existe o bloco *finally*



Bloco finally

- O bloco *finally* é colocado após o bloco *try/catch*, e é sempre executado, independente de haver ocorrido exceção

Bloco finally

Nesse exemplo, a leitura do teclado foi feita para um String para posterior conversão. Isso pode gerar o erro do usuário informar valores que não podem ser convertidos em números

```
Scanner in = new Scanner(System.in);

try {

    System.out.println("Informe o primeiro numero");
    String resposta = in.nextLine();
    double nr1 = Double.parseDouble(resposta);

    System.out.println("Informe o segundo numero");
    resposta = in.nextLine();
    double nr2 = Double.parseDouble(resposta);

    double resultado = nr1 + nr2;

    System.out.println("A soma é " + resultado);

} catch (NumberFormatException nfe) {
    System.out.println("Não é possível somar coisas que não são números");
} finally {
    in.close();
}
```

Ocorrendo ou não uma Exceção, o **scanner** deve ser fechado, por isso o **finally**



Por fim

- Prefira
- Do que

```
String a;
```

```
int tamanho = 0;
```

```
if (a != null) {  
    tamanho = a.length();  
}
```

```
String a;
```

```
int tamanho;
```

```
try {  
    tamanho = a.length();  
} catch (NullPointerException npe) {  
    tamanho = 0;  
}
```

A performance do tratamento de exceções não é tão boa,
e portanto ele só deve ser feito quando necessário

Por fim

- Ao invés de

```
String a;
```

```
for (int i = 0; i < 10; i++) {  
    try {  
        System.out.println(a.length());  
    } catch (NullPointerException npe) {  
        System.out.println("A String a é nula!");  
    }  
}
```



Por fim

- Faça

```
String a;
```

```
try {  
    for (int i = 0; i < 10; i++) {  
        System.out.println(a.length());  
    }  
} catch (NullPointerException npe) {  
    System.out.println("A String a é nula!");  
}
```

O tratamento de Exceções
aplicado ao bloco
todo é mais eficiente,
limpo, e possui
performance melhor!



Finalizando

- Use exceções para:
 - 1) Consertar o problema e chamar o método que causou a exceção de novo
 - 2) Contornar o erro e continuar sem tentar o método novamente
 - 3) Calcular algum resultado alternativo em vez daquele que o método deveria produzir
 - 4) Fazer o que for possível no contexto atual e lançar a mesma exceção para o contexto superior
 - 5) Fazer o que for possível no contexto atual e lançar uma exceção diferente para o contexto superior
 - 6) Terminar o programa
 - 7) Simplificar. Se seu esquema de exceções complica as coisas, então ele será ruim para ser usado
 - 8) Tornar sua biblioteca e seu programa mais seguros



Dúvidas?





Atividades

- Execute as atividades presentes no documento

09.Lista.de.Exercícios.POO.pdf

Próximos passos

- Coleções

