SCC0504 - Programação Orientada a Objetos

Exceções

Prof. Jose Fernando Rodrigues **Junior**

http://www.icmc.usp.br/~junio

junio@icmc.usp.br

INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO - USP

Introdução

- **Sistemas computacionais trabalham sobre um conjunto grande de variáveis**
  - **Internas**: declaradas dentro do código, perfazendo o sistema computacional
  - **Controláveis**: variáveis externas como disco, rede, e dispositivo de entrada, os quais podem ser controlados
  - **Incontroláveis**: incêndios, terremotos, ...

- **Não é possível garantir que sempre irá funcionar**

➔ **Qualquer linguagem** de programação possui um sistema de tratamento de erros, chamado **sistema de tratamento de exceções**.

Tratamento de erros

Objetivo: **evitar e/ou corrigir erros** oriundos da execução de programas

O programador está sujeito a **uma série de erros**:

- Extrapolar os limites de um vetor
- Ignorar o domínio de algumas operações (ex. X / 0)
- Não realizar corretamente a conversão de tipos
- Tentar acessar um arquivo que não existe

Há ainda situações externas:

- Acessar página que não existe
- Gravar em disco cheio
- Sem acesso a determinado diretório
- A rede deixa de funcionar

Exceções

- Exceções se referem a **erros em tempo de execução**
- **Em Java: objetos** de classes especiais que são "**lançados**" quando ocorrem **condições excepcionais**
- Os métodos podem **capturar ou deixar passar** exceções
- Mecanismo **`try-catch`** é usado para capturar exceções

Objetivo: gerar programas **tolerantes a falhas**

Erros em tempo de execução

Tipos de erros:

- Erros de lógica de programação
  - Ex. limite de vetor ultrapassado, divisão por 0

  ➔ Devem ser corrigidos pelo programador

- Erros devido a condições do ambiente de execução
  - Ex. arquivo não encontrado, rede fora do ar

  ➔ Podem ser resolvidos

- Erros graves, quando não adianta tentar recuperação
  - Ex. falta de memória, erro da JVM

  ➔ Não podem ser resolvidos

Exceções em Java

- Em Java, uma exceção faz com que **uma instância de um objeto descendente das classes Exception, RuntimeException, ou Error seja criado**

- Esta **instância é passada para o runtime do java** – sistema que gerencia a execução das aplicações

- O runtime **procura um trecho de código (exception handler)** capaz de **gerenciar o erro – a ordem de procura segue a pilha (stack) de execução**

- A busca procura na pilha **até o main** – se não encontrar um exception handler, o **programa é terminado**

Vantagens

1) Separação entre código e tratamento de erro

2) Propagação considerando os erros

3) Agrupamento e diferenciação de tipos de erros

➔ Eficiente comunicação entre programadores e utilizadores de classes

Criando, usando, e tratando uma exceção

Criando uma exceção de valor indevido

```
public class InvalidValueException extends Exception {
    // Construtor 1
    public InvalidValueException () { }

    // Construtor 2
    public InvalidValueException (String msg) {
       super("Data inválida, mês fora do intervalo");
    }
}
```

Criando, usando, e tratando uma exceção

Usando a nova classe de exceção

```
public class Data {
  public void setData(int dia, int mes, int ano) throws
                              InvalidValueException  {


    if(mes < 1 || mes > 12)
        throw new InvalidValueException();
    else
      ...
   }
}
```

Criando, usando, e tratando uma exceção

Usando a exceção/Tratando a exceção

```
class MeuProjeto{
   public static void meuMetodo(){
       Data d = new Data();
       d.setData(8,4,2015); ➔ não compila
   }
}
class MeuOutroProjeto{
   public static void main(String args[]){
       MeuProjeto m = new MeuProjeto();   ➔ não compila
       d.meuMetodo();
   }
}
```

Criando, usando, e tratando uma exceção

Usando a exceção/Tratando a exceção

```
class MeuProjeto{
   public static void meuMetodo(){
       Data d = new Data();
       try{
           d.setData(8,4,2015);            /*Compilação OK*/
       }catch(InvalidValueException minhaExc){
           System.out.println(minhaExc.getMessage());
       }
   }
}
class MeuOutroProjeto{
   public static void main(String args[]){
       MeuProjeto m = new MeuProjeto();      /*Compilação OK*/
       d.meuMetodo();
   }
}
```

Criando, usando, e tratando uma exceção

(Não) Tratando a exceção

```
class MeuProjeto throws InvalidValueException {
   public static void meuMetodo{
       Data d = new Data();
       d.setData(8,4,2015);            /*Compilação OK*/
   }
}
class MeuOutroProjeto{
   public static void main(String args[]){
       MeuProjeto m = new MeuProjeto();
       m.meuMetodo();                          ➔ não compila
   }
}
```

Criando, usando, e tratando uma exceção

(Não) Tratando a exceção

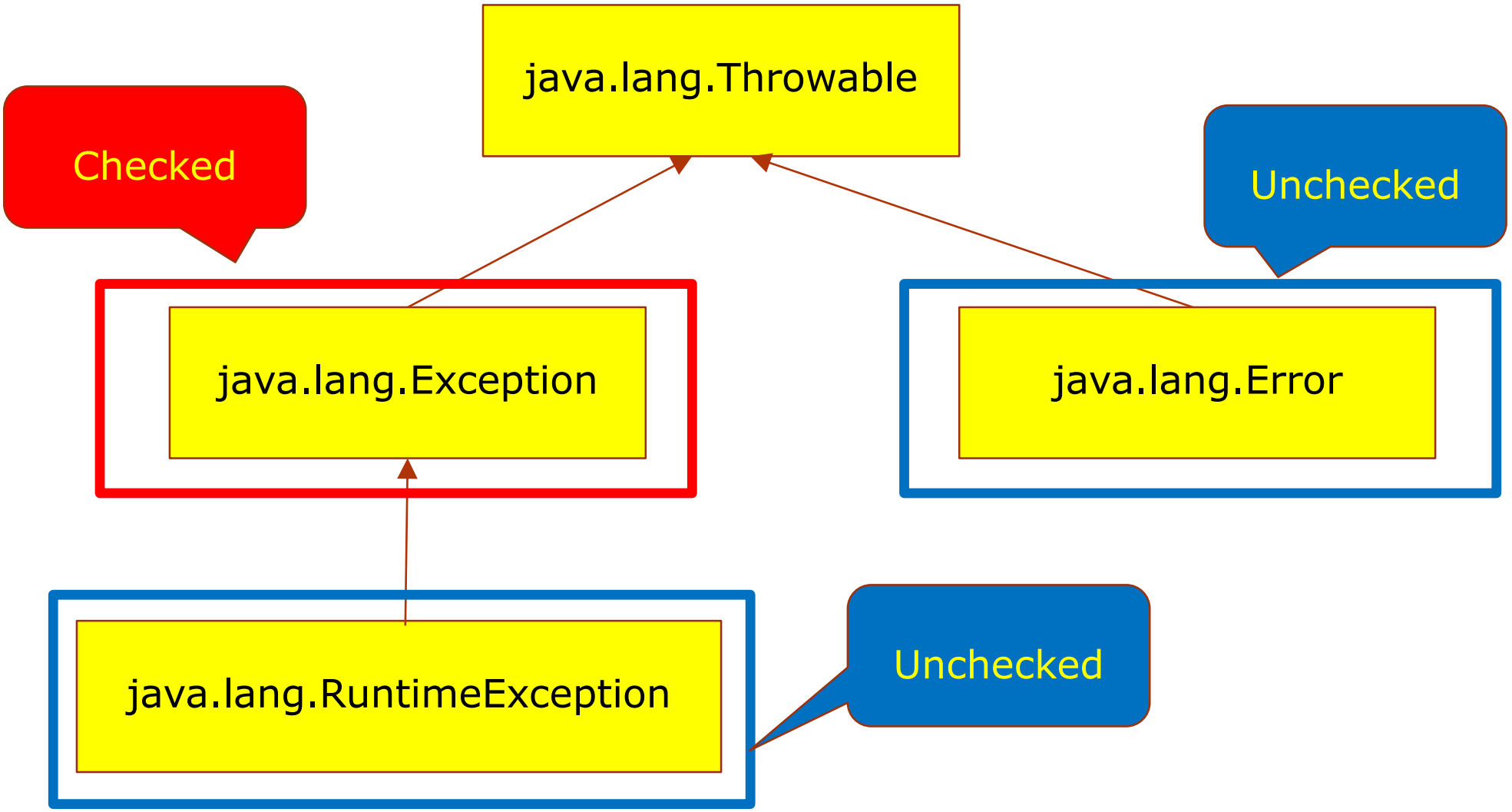
```
class MeuProjeto throws InvalidValueException {
   public static void meuMetodo{
       Data d = new Data();
       d.setData(8,4,2015);            /*Compilação OK*/
   }
}
class MeuOutroProjeto{
   public static void main(String args[]){
       MeuProjeto m = new MeuProjeto();
       try{
           m.meuMetodo();                          /*Compilação OK*/
       }catch(InvalidValueException minhaExc){
           System.out.println(minhaExc.getMessage());
       }
   }
```

Tipos de exceção Java

- **Checked (Exception)**: devem, OBRIGATORIAMENTE, ser tratadas por código exception handler, caso contrário não é possível compilar o programa
  - Requer **try-catch** ou **throws**

- **Unchecked:** sem tratamento obrigatório, dois subtipos
  - **Runtime (RuntimeException)**: eventos internos da lógica e do fluxo do programa; podem ser detectados pelo programador, mas não pelo compilador, exemplo: *null pointer exception*
  - **Erro (Error):** eventos externos não previsíveis, por exemplo, erro mecânico de disco → *java.io.IOError*

Tipos de exceção Java



Exemplo - unchecked runtime

Agenda eletrônica: Onde pode ocorrer erro?

```
public class Agenda extends ArrayList <Contato>{
     public Contato getContato(String nome) {
        for(int i=0;i<this.size();i++){
            Contato t = this.get(i);
            if(t.getNome().compareTo(nome) == 0)
                return t;
        }
        return null;
     }

     public void atualizaTelefone(String nome, String nTel){
         Contato c = this.getContato(nome);
         c.setTelefone(nTel);
     }
}
```

Exemplo

- `AgendaEletrônica` é um exemplo onde o desenvolvedor não se preocupou com situações adversas

- Chamar o método `atualizaTelefone` com um nome de contato não cadastrado irá gerar um erro de execução, pois a variável `c` será nula

Exemplo

Como ser despedido do seu emprego rapidamente:

```
…
    public void atualizaTelefone(String nome, String nTel){
         try{
            Contato c = this.getContato(nome);
            c.setTelefone(nTel);
         }catch(NullPointerException e){
            /*Nada aconteceu, eu sou um ótimo programador*/
         }
    }
...
```

Lançando uma exceção

- A **API Java** lança exceções em diversos contextos
- Qualquer programador **pode criar suas exceções**

```
if(condicao_de_erro == true)
   throw new Exception("Condicao de erro detectada");
```

- Objeto precisa ser criado com `new` e lançado com `throw`
- Ao lançar uma exceção, a classe comunica que não foi capaz de realizar a operação

➔ **trata-se, na verdade, de um sistema que detecta o erro ANTES que ele aconteça**

Lançando uma exceção

Exemplo:

```
public void atualizaTelefone(String nome, String novoTelefone){
     Contato c = contatos.get(nome);
     if(c.startsWith("999"))
         throw new Exception("Numero invalido");
     c.setTelefone(novoTelefone);
}
```

Sintaxe: **throw new** Tipo_de_Exceção("*string de diagnóstico (opcional)*")

Lançando uma exceção

Sintaxe: **throw new** Tipo_de_Exceção(“*string de diagnóstico (opcional)*”)

- A string de diagnóstico é opcional, mas é importante para transmitir informações mais detalhadas
- O utilizador do objeto que recebeu a exceção poderá verificar a string por meio de `getMessage` ou `toString`

Efeitos de exceção

Uma exceção lançada interrompe o fluxo do programa

- Que passa a seguir a exceção
  - o **método termina** sem precisar executar a instrução de retorno return

- Se o método onde ela ocorrer não a capturar, **ela será propagada** para quem chamou, e assim por diante
- Se ninguém capturar a exceção, ela **causará o término da aplicação**
- **Se ela for capturada, o controle pode ser recuperado**

Ou o programador trata a exceção gerada,

ou a execução do programa será abortada

Capturando e tratando exceção

- Capturar uma exceção significa **providenciar um trecho de código que detecta o lançamento** de uma exceção e dispara ações correspondentes

- Isso é feito pelo bloco try … catch

- *Sintaxe:*

```
try {
    // chamadas de métodos que podem lançar exceções
}
catch (Exception e) {
    //    ações    correspondentes    à    detecção    de    uma
     determinada exceção
}
```

Capturando e tratando exceção

- Se uma exceção for detectada, a execução é abortada e o fluxo do programa segue para a **primeira instrução do bloco `catch`**, correspondente àquela exceção

- **Se não ocorrer** o lançamento de uma exceção, **o bloco `catch` é ignorado**

Propagando uma exceção

Ex.: Se temos a seguinte instrução:

```
FileReader stream = new FileReader("\teste.txt");
```

O compilador irá informar que há uma exceção verificada (checked) que deve ser capturada ou declarada

- *"unreported exception java.io.FileNotFoundException must be caught or declared to be thrown"*

Propagando uma exceção

Para capturar:

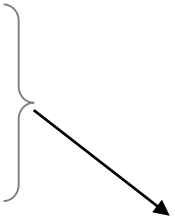
```
try {
    FileReader stream = new FileReader(“c:\\teste.txt");
 }
catch (IOException ex) {
            System.out.print(ex.toString());
 }
```

Capturando e tratando exceção

Exemplo:

```
try  {
    FileWriter stream = new
    FileWriter("c:\\teste.txt");
    PrintWriter out = new PrintWriter(stream);

    out.println("oi");
    out.close();
}
catch ( IOException erro )  {
    System.out.println (erro.getMessage());
}
```



Podem lançar exceções
Ex: java.io.FileNotFoundException

Capturando e tratando exceção

O bloco catch nomeia o tipo de exceção que ele é projetado para tratar, podendo haver vários blocos catch, um para cada tipo de exceção

Assim, o tratamento da exceção será mais detalhado

```
try { . . . }
catch (FileNotFoundException e) {
    // ações correspondentes à detecção de uma
    exceção referente a um arquivo não encontrado
}
catch (NumberFormatException e) {
    // ações correspondentes à detecção de uma
    exceção referente a uma conversão inadequada de
    string para número
}
```

Propagando uma exceção

Para propagar: instrução throws

```
public void teste() throws FileNotFoundException
    FileReader stream = new FileReader("c:\\teste.txt");
 }
```

Método (teste) não se responsabilizará por tratar a exceção: ele irá passar essa responsabilidade pra frente!!, ou seja, para qualquer método que o tenha chamado

Propagando uma exceção

A propagação pode ser feita ao longo de vários métodos

Ex:

```
public void teste3() throws FileNotFoundException {
   FileReader stream = new FileReader("c:\\teste.txt");
 }
 public void teste2() throws FileNotFoundException {
   teste3();
 }
public void teste1() {
   try {
      teste2();
   }
   catch(FileNotFoundException e) {
   }
}
```

Propagando uma exceção

Importante:

Todo método que lança uma exceção verificada, deverá adicionar a clausula **`throws`** no seu cabeçalho

Ex:

```
public void setData(int dia, int mês, int ano) throws IOException {

     if (dia<0 || dia > 31) throw new IOException("Dia inválido");
     if (mês<0 || mês > 12) throw new IOException("Mês inválido");
     this.dia = mês;
     this.mês = mês;
     this.ano = ano;
 }
```

A cláusula finally

O bloco `try...catch` pode incluir um terceiro componente opcional: `finally`

```
try {
 // chamadas de métodos que podem lançar exceções
}
catch (Exception e) {
 // ações correspondentes à detecção de uma determinada exceção
}
finally {
 //executado sempre
}
```

A cláusula finally

As instruções no bloco finally **serão executadas** impreterivelmente

```
try {
 // chamadas de métodos que podem lançar exceções
}
catch (Exception e) {
 // ações correspondentes à detecção de uma determinada exceção
}
finally {
 //executado sempre
}
```

A cláusula finally

```
public void openFile(){
        FileReader reader = null;
        try {
            reader = new FileReader("someFile");
            int i=0;
            while(i != -1){
                i = reader.read();
                System.out.println((char) i );
            }
        } catch (IOException e) {
            System.out.println(e.getMessge());
        } finally {
            if(reader != null){

                    reader.close();   //Sempre



            }
            System.out.println("--- File End ---");
        }
    }
```

A cláusula finally

```
public void openFile(){
        FileReader reader = null;
        try {
            reader = new FileReader("someFile");
            int i=0;
            while(i != -1){
                i = reader.read();
                System.out.println((char) i );
            }
        } catch (IOException e) {
            System.out.println(e.getMessge());
        } finally {
            if(reader != null){
                try {
                    reader.close();   //Sempre
                } catch (IOException e) {
                   System.out.println(e.getMessge());
                }
            }
            System.out.println("--- File End ---");
        }
    }
```

Novas classes de exceções

- Caso nenhuma classe de exceção existente tiver um significado ligado à exceção que se queira lançar, **uma classe de exceção que a represente pode ser criada**;

- Para isso é só fazer essa classe derivar de **Exception, RuntimeException, ou Error** ou qualquer outra abaixo na hierarquia

- Boa prática: **prefira usar as classes de exceções presentes na API Java** antes de criar suas próprias exceções