

Bacharelado em Ciências da Computação

BC1501 – Programação Orientada a Objetos

UFABC

Universidade Federal do ABC

Aula 05

**Material Adaptado do
Prof. André G. R. Balan**

Nesta Aula

- ▶ Tratamento de erros
- ▶ Exceções

Erros de programação

- ▶ Ao desenvolver um sistema computacional, um **programador** (que é um ser humano), **está sujeito a cometer uma série de erros**:
 - Extrapolar os limites de um vetor, acessando objetos que não existem
 - Ignorar o domínio de algumas operações:
 - Na divisão matemática o divisor não pode ser nulo
 - Uma string só pode ser convertida para um valor numérico se, de fato, ela representar um número
 - Não elaborar corretamente condições de parada, gerando loops infinitos

Erros de programação

- ▶ Além disso, mesmo um programa testado pode ainda falhar diante de situações externas:
 - O programa tenta **acessar uma página WEB que não existe**
 - O programa tenta **gravar dados em um disco cheio**
 - O programa tenta **acessar um dispositivo externo** que está terminantemente ocupado
 - O programa **não tem acesso** a um determinado diretório ou recurso externo
 - A rede deixa de funcionar
 - Etc..

Tópicos da aula

- ▶ Nesta aula vamos **estudar técnicas para evitar e/ou corrigir erros** oriundos da execução de programas
- ▶ **Tópicos**
 - **Programação defensiva**
 - Verificação de argumentos
 - Notificações
 - **Lançamento de exceções**
 - Hierarquia de classes
 - Efeito de uma exceção
 - Exceções verificadas e não verificadas
 - **Tratamento de exceções**
 - Múltiplas exceções
 - Propagação de exceções

Exemplo inicial

- ▶ Voltemos ao exemplo do exercício sobre **contatos profissionais**:

```
public abstract class Profissional {  
    public String nome;  
    public String email;  
    protected ArrayList<Profissional> contatos = new ArrayList<Profissional>();  
  
    public void adicionaContato(Profissional profissional) {  
        if (profissional == this) return;  
        if (contatos.contains(profissional)) return;  
        else {  
            contatos.add(profissional);  
            profissional.adicionaContato(this);  
        }  
    }  
    .  
    .  
    .  
}
```

Exemplo inicial

- ▶ Voltemos ao exemplo do exercício sobre contatos profissionais:

```
public abstract class Profissional {  
    public String nome;  
    public String email;  
    protected ArrayList<Profissional> contatos = new ArrayList<Profissional>();  
  
    public void adicionaContato(Profissional profissional) {  
        if (profissional == this) return;  
        if (contatos.contains(profissional)) return;  
        else {  
            contatos.add(profissional);  
            profissional.adicionaContato(this);  
        }  
    }  
    .  
    .  
    .  
}
```



*Neste ponto, o método pode gerar um **erro de execução** se o valor do parâmetro `profissional` for nulo!*

Tipos de programadores

- ▶ Na classe Profissional o **desenvolvedor não se preocupou em tratar todas as possíveis situações de erro**
 - *Se eu mesmo sou o usuário das classe que desenvolvi, eu sei que **nunca** vou chamar o método passando valor nulo!!*
- ▶ ***Mas, lembre-se:***
 - Se o programa ficar muito grande pode ser que **você não se lembre de tomar certos cuidados** ao utilizar alguns métodos que você mesmo criou
 - Você precisa considerar o fato de que **outras pessoas** (da sua empresa, por exemplo) **poderão fazer manutenção** no seu programa
 - Futuramente, **outras pessoas poderão estar interessadas em dar continuidade** ao seu trabalho.


Tipos de programadores

- ▶ Pergunta:
 - *De quem é a responsabilidade por tratar de situações como esta? Do **desenvolvedor** da classe, ou **usuário** da classe?*
- ▶ Resposta: Ambos devem se preocupar em lidar com tais situações. Porém, **a iniciativa deve partir sempre do desenvolvedor da classe**, pois o seu trabalho sempre precede o trabalho do usuário.
- ▶ Tais atitudes constituem uma **Programação Preventiva**

Programação Defensiva

► Verificação de argumentos

- Uma das **principais atitudes** do desenvolvedor de classes com relação à programação defensiva é a **verificação de argumentos** passados aos métodos da classe. Exemplo:

```
public abstract class Profissional {  
    public void adicionaContato(Profissional profissional) {  
                if (profissional == null) return;  
        if (profissional == this) return;  
        if (contatos.contains(profissional)) return;  
        else {  
            contatos.add(profissional);  
            profissional.adicionaContato(this);  
        }  
    }  
    . . .  
}
```

Programação Defensiva

- ▶ Mas, no caso anterior, o desenvolvedor da classe Profissional **não transmite nenhuma informação de retorno** sobre o que aconteceu na chamada do método.
- ▶ Se, por algum motivo, o método não adicionou o contato, o **usuário poderá não ter conhecimento**

Programação Defensiva

- Isto pode ser feito por meio de uma mensagem na tela. Ex:

```
public abstract class Profissional {  
    . . .  
    public void adicionaContato(Profissional profissional) {  
        if (profissional == null) {  
            System.out.println("Parâmetro nulo!");  
            return;  
        }  
        if (profissional == this) {  
            System.out.println("O objeto não pode adicionar-se!");  
            return;  
        }  
        . . .  
    }  
}
```

Programação Defensiva

- ▶ Isto pode não ser uma boa alternativa em várias situações. Ex:
 - Neste caso estamos supondo que a aplicação está sendo utilizada por um ser humano que verá a mensagem. Entretanto, **há varias aplicações que são executadas sem o monitoramento de um usuário humano.**
 - Em outros casos, mesmo quando há um humano interagindo no sistema, **pode ser inviável que este usuário venha a tomar uma decisão segundo alguma mensagem recebida.** Ex: suponha um usuário de um caixa eletrônico de banco recebendo uma mensagem “Acesso a uma posição inválida do vetor”

Programação Defensiva

- ▶ Alternativamente, **uma comunicação** com o método chamador for feita por meio do valor de retorno do método chamado.

```
public boolean adicionaContato(Profissional profissional) {  
    if (profissional == null) return false;  
    if (profissional == this) return false;  
    if (contatos.contains(profissional)) return false;  
    else {  
        contatos.add(profissional);  
        profissional.adicionaContato(this);  
        return true  
    }  
}
```



Programação Defensiva

- ▶ Entretanto, **alguns problemas surgem** quando a classe utilizada tenta se comunicar com o usuário/utilizador por meio do valor de retorno dos métodos:
 - **Fica difícil e confuso transmitir informações mais detalhadas** para o utilizador da classe sobre uma operação malsucedida (motivos, quais dados incorretos, etc...)
 - **Não há como disciplinar (obrigar) o utilizador a realmente verificar o valor de retorno**
 - Ex: se o usuário não verificar o valor de retorno ele pode, simplesmente, acabar gerando um erro ao acessar um objeto que não existe (NULL)

Exceções

- ▶ **Por estes motivos**, principalmente, as linguagens de programação orientadas a objeto incorporam um **mecanismo eficiente de “comunicação” entre programadores** desenvolvedores e utilizadores de classes, para lidarem com situações especiais (exceções).
- ▶ **Lançamento e tratamento de exceções**


Lançamento de Exceções

- ▶ Neste contexto, uma exceção é um objeto que representa os detalhes de uma falha de execução do programa, podendo esta falha ser ou não crítica
- ▶ Lançar uma exceção é uma ação realizada pela classe utilizada (Ex: **Profissional**), que é realizado pela palavra reservada **throw**.
- ▶ Ao lançar uma exceção, a classe utilizada comunica de maneira eficiente que não foi capaz de realizar com sucesso a operação solicitada.

Lançamento de Exceções

► Exemplo:

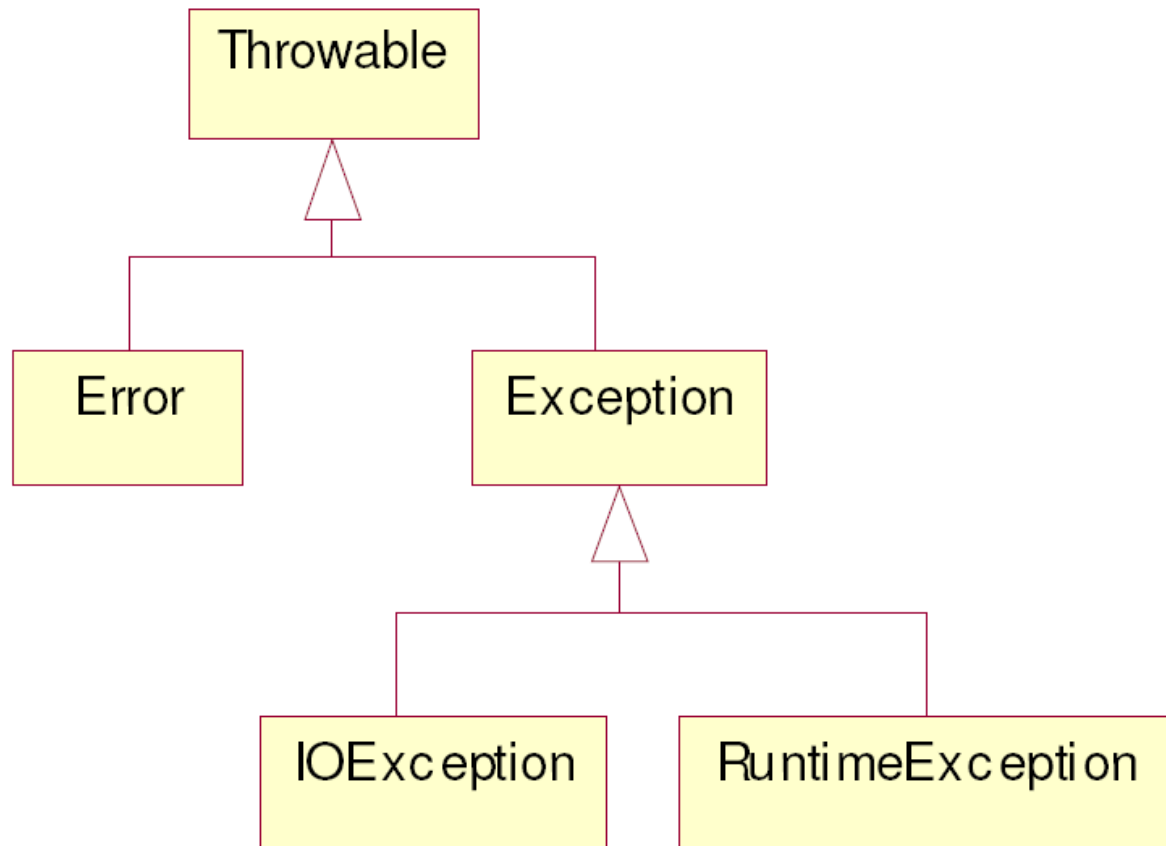
```
public boolean adicionaContato(Profissional profissional) {  
    if (profissional == null)  
        throw new NullPointerException("Parâmetro Nulo");  
    if (profissional == this)  
        throw new RuntimeException("Tentativa de adicionar o próprio objeto");  
    if (contatos.contains(profissional))  
        throw new RuntimeException("Objeto já adicionado");  
    else {  
        contatos.add(profissional);  
        profissional.adicionaContato(this);  
    }  
}
```



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

Classes de Exceção

- ▶ O pacote java.lang define uma hierarquia de classes para as exceções:



Classes de Exceção

- ▶ A classe **Error** define um tipo de erro causado pelo sistema, ex: **StackOverflow**, **OutOfMemory**, e não pode ser lançado diretamente pelo programador
- ▶ A hierarquia **Exception**, por sua vez, se divide em dois ramos **RuntimeException** (*captura não obrigatória*) e **IOException** (*captura obrigatória*).

Efeitos de uma exceção

► No método que lançou a exceção

- A execução do método termina imediatamente
- Não é exigido que um método com valor de retorno não *void* execute a instrução de retorno *return*
- Isto é interessante, pois o método realmente não foi capaz de executar a ação, por isso, não lhe é exigido nenhum retorno.

Efeitos de uma exceção

► Na chamada do método

- Irá depender de acordo com a existência ou não de um código para tratar a exceção lançada.
- Em ambos os casos a execução do método chamado será abortada
- Ex: `p1.adcionaContato(null)`
// A seguinte instrução não será executada
`p1.mostraContatos();`
- Isso ilustra perfeitamente o poder das exceções em evitar que o código do usuário utilizador prossiga sem se importar com o problema detectado:
 - Ou o programador trata a exceção gerada
 - Ou a execução do programa será abortada

Efeitos de uma exceção

- ▶ Existe também um efeito bem interessante para os métodos que lançam uma exceção:
 - Se a exceção for lançada dentro de um método construtor, a criação do objeto será abortada.
 - Isto é interessante para garantir que na criação de um objeto, todos os parâmetros sejam passados corretamente. Ex: não é possível criar um objeto contado sem as informações principais: nome e telefone.
- ▶ Para o método que chamou o construtor o efeito é o mesmo:
 - A chamada do construtor foi abortada (**não será retornado null!!**)
 - O chamador deverá tratar a exceção ou o programa será abortado.

Capturando e tratando exceções

- ▶ Capturar uma exceção significa providenciar um trecho de código, por parte do utilizador da classe, que detecte o lançamento de uma exceção e dispare ações correspondentes.
- ▶ Isto é feito por meio do 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 exceções

► 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 na escrita dos dados" );  
}
```

Capturando e tratando exceções

- ▶ As duas primeiras chamadas de métodos dentro do bloco try do exemplo anterior podem lançar exceções. Ex:
`java.io.FileNotFoundException`
- ▶ 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

Capturando e tratando exceções

- ▶ O bloco **catch** nomeia o tipo de exceção que ele é projetado para tratar, podendo haver a especificação de **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 um conversão inadequada de  
    // string para número  
}
```

Exceções não verificadas

- ▶ Uma exceção é dita **não-verificada** se ela for uma subclasse da classe RuntimeException.
- ▶ Ao chamar um método que pode lançar uma exceção não verificada, o programador **pode ou não** incluir instruções de captura e tratamento (try...catch)
- ▶ **Ele não é obrigado a fazer isso**

Exceções verificadas

- ▶ Uma exceção é dita **verificada** se ela for uma subclasse da classe **IOException**.
- ▶ Ao chamar um método que pode lançar uma exceção verificada, o programador **deve obrigatoriamente** incluir instruções de captura e tratamento (try...catch)
- ▶ Só há **uma opção**:
 - *Propagar a exceção*

Propagando uma exceção

- ▶ Exemplo: Se tivermos a seguinte instrução no nosso programa, o compilador irá reclamar, dizendo que uma exceção verificada (“*unreported*”) deve ser capturada ou propagada:

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

- ▶ Mais especificamente:
 - “*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());  
}  
}
```

- ▶ A **propagação** de uma exceção é feita pelo método que chama um método lançador de exceção

Propagando uma exceção

▶ Exemplo

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

- ▶ A propagação é indicada com a cláusula **throws** no cabeçalho do método
- ▶ Significa que este método (teste) não se responsabilizará por tratar a exceção: **ele irá passar essa responsabilidade para frente!!**, ou seja, para qualquer método que o chamar

Propagando uma exceção

▶ Exemplo 2

```
public void setNascimento(String nascimento) throws  
    ParseException {  
  
    SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");  
  
    this.nascimento = df.parse(nascimento);  
  
}
```

Propagando uma exceção

- ▶ A propagação pode ser feita através de vários métodos. Ex:

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

Definindo 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 ou qualquer outra abaixo da hierarquia de Exception;

Definindo novas classes de exceções

▶ Exemplo:

```
public class InvalidValueException extends IOException {  
    // Construtor 1  
    public InvalidValueException () { }  
  
    // Construtor 2  
    public InvalidValueException (String msg) {  
        super(msg);  
    }  
}
```

Finally

```
try {  
    FileReader stream = new FileReader("c:\\teste.txt");  
}  
catch (IOException ex) {  
    System.out.print(ex.toString());  
}  
  
finally { // ← Opcional!  
    // Este bloco sempre será executado!  
    // Com ou sem exceção...  
}
```