FATEC – Votorantim
Profº Rodrigo Diver

Aula 8 – Tratamento de Exceções

- **Tratamento de Exceções**

Slides adaptados das notas de aula do Profº Edson Luiz – UNESP FEG

Tratamento de Exceções

- Exceção indica a ocorrência de erro durante a execução do programa.
  Exemplos de situações que provocam exceções:
  - índice de vetor fora dos limites;
  - valor numérico fora do intervalo representável;
  - divisão por zero;
  - falta de memória;
  - conversão de tipos de dados.

Exemplo:

```
String sIdade;
int idade;
sIdade = JOptionPane.showInputDialog("Idade?");
idade = Integer.valueOf(sIdade);
```

- O que acontece se o string fornecido pelo usuário não puder ser convertido para um valor numérico?
- A ocorrência de uma exceção provoca a terminação do programa.

Tratamento de Exceções

- O tratamento de exceções consiste em capturar as exceções: o programa continua sua execução e pode processar o erro de uma forma conveniente.
- Com o tratamento de exceções os programas se tornam mais tolerantes a falhas.
- Algumas exceções podem ocorrer de forma imprevisível, devido a problemas no ambiente JVM. Por exemplo: falta de memória. Este tipo de exceção é tratado pela classe Error.
- Este capítulo se concentra nas exceções previsíveis. Por exemplo: erro na conversão de dados. Este tipo de exceção é tratado pela classe Exception.
- A API Java define diversas subclasses da classe Exception, para o tratamento de exceções específicas. Por exemplo: ArithmeticException, IndexOutOfBoundsException, NullPointerException, ...

Tratamento de Exceções

- A classe Exception dispõe de dois métodos muito úteis para a depuração de programas:
  - printStackTrace(), que mostra a sequência de métodos chamados até que a exceção ocorreu, ou seja, o caminho de execução que levou à exceção.
  - getMessage(), que exibe a mensagem de erro associada à exceção.
- A classe Exception dispõe de dois construtores:
  - Exception(), que constroi um objeto padrão; e
  - Exception(String s), que constroi um objeto com uma mensagem de erro específica.

- Exemplo:

```
public class Exemplo
{
  public static void main (String [] args)
  {
    int vetor[] = new int[100];
    for (int i = 0; i <= 100; i++)
      vetor[i] = 1;
    System.out.println("Vetor unitário construído!");
  }
}
```

O que acontece quando o programa é executado?

```
Output - Aula06 (run)
  run:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 100
        at aula06.Exemplo.main(Exemplo.java:9)
  Java Result: 1
  BUILD SUCCESSFUL (total time: 0 seconds)
```

- O tratamento de exceções é feito com blocos try-catch.

```
public class Exemplo
{
  public static void main (String [] args)
  {
    int vetor[] = new int[100];
    try
    {
      for (int i = 0; i <= 100; i++)
        vetor[i] = 1;
      System.out.println("Vetor unitário construído!");
    }
    catch (Exception e)
    {
      System.out.println("Exceção: " + e.getMessage());
    }
  }
}
```

```
Output - Aula06 (run)
run:
Exceção: 100
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Observações importantes:
  - Um bloco try exige, pelo menos, um bloco catch.
  - Cada catch tem apenas um argumento (um objeto de uma classe de exceção), ou seja, cada catch captura exceções de uma classe específica de exceções.

```
try
{
   ...
}
catch (ArithmeticException e)
{
   ... um tratamento específico ...;
}
catch (Exception e)
{
   ... um tratamento geral ...;
}
```

  - Um bloco catch com argumento da classe Exception captura qualquer tipo de exceção. Este bloco deve ser o último da lista de blocos catch.

Certos pontos chave precisam ser lembrados:

- Em um método, pode haver mais de uma instrução que pode lançar uma exceção. Portanto, coloque todas essas instruções em seu próprio **bloco try** e forneça um manipulador de exceção separado em seu próprio **bloco catch para cada uma delas.**

- Se ocorrer uma exceção no **bloco try** , essa exceção será tratada pelo manipulador de exceções associado a ela. Para associar o manipulador de exceções, devemos colocar um **bloco catch depois dele**.

- Pode haver mais de um manipulador de exceções. Cada bloco catch é um manipulador de exceções que trata a exceção para o tipo indicado por seu argumento.

- O argumento **ExceptionType** declara o tipo de exceção que ele pode tratar e deve ser o nome da classe que herda da classe **Throwable** .
- Para cada **bloco try**, pode haver **um ou mais blocos catch**, **mas apenas um bloco finally.**

- **O bloco finally é opcional.** Ele sempre é executado **independentemente de uma exceção ter ocorrido no bloco try ou não.** Se ocorrer uma exceção, ela será executada após os **blocos try e catch.** E se não ocorrer uma exceção, **ela será executada após o bloco try** .

- O bloco **finally** em Java é usado para colocar códigos importantes que precisam ser executados, como códigos para fechar a conexão com o banco de dados.

Tratamento de Exceções

```
public class Exemplo2
{
  public static void main (String [] args) {
    try {
      int vetor[] = new int[100];
      float valor1 = 1, valor2 = 0;
      System.out.println(valor1/valor2);
      System.out.println(vetor[100]);
    }
    catch(ArrayIndexOutOfBoundsException  e) {
      System.out.println("Indice inválido. Exceção: " + e);
    }
    catch(ArithmeticException e) {
      System.out.println("Erro aritmético. Exceção: " + e);
    }
    catch(Exception e) {
      System.out.println("Ocorreu a exceção: " + e);
    }
    System.out.println("Após o tratamento de exceções...");
  }
}
```

Java permite a divisão por zero para as classes **Float** e **Double**, que resulta em **infinito**.

O que vai ocorrer neste caso?

Disparo de exceções

- A instrução throw provoca o disparo de uma exceção.

```
public class DisparaExcecao
{
  public static void main (String [] args)
  {
    int a,b,c;
    ...
    try
    {
      if (b == 0)
        throw new ArithmeticException("Erro de cálculo");
      ...
    }
    catch(ArithmeticException e)
    {
      System.out.println("Ocorreu a exceção: " + e);
    }
  }
}
```

- Um método pode declarar as classes das exceções que podem ocorrer durante sua execução. Neste caso, o programa deve fornecer, obrigatoriamente, um tratamento para as exceções dessas classes.

```
public class TrataExcecao
{
  public static void main (String [] args)
  {
    try {
      umMetodo();
    }
    catch (Exception e) {
      System.out.println("Indice fora dos limites");
    }
  }

  static void umMetodo() throws ArithmeticException
  {
    ...
  }
}
```

O tratamento pode ser **específico** (para uma classe de exceção específica) ou **geral** (para a classe **Exception**).

As classes de **exceções** que um método pode disparar devem ser declaradas após a cláusula **throws**. Esse método avisa que pode gerar (não necessariamente vai gerar) exceções da classe ArithmeticException.

Tratamento de Exceções

Exemplo:

```
public class DepuraPrograma {
  public static void main(String args[]) {
    try {
      metodo1();
    }
    catch (Exception e) {
      System.err.println(e + "\n");
      e.printStackTrace();
    }
  }
  public static void metodo1() throws Exception {
    metodo2();
  }
  public static void metodo2() throws Exception {
    metodo3();
  }
  public static void metodo3() throws Exception {
    throw new Exception("Exceção disparada no metodo3");
  }
}
```

```
java.lang.Exception: Exceção disparada no metodo3
        at aula06.Exemplo.metodo3(Exemplo.java:20)
        at aula06.Exemplo.metodo2(Exemplo.java:17)
        at aula06.Exemplo.metodo1(Exemplo.java:14)
        at aula06.Exemplo.main(Exemplo.java:6)
```

- Alguns métodos da Java API exigem o tratamento de exceções.
- Exemplo:

```
public void gravarArquivo(String texto, String arq)
{
  File f = new File(arq);
  PrintWriter pw = new PrintWriter(f);
  pw.println(texto);
  pw.flush();
  pw.close();
}
```

E se o arquivo não existir? O construtor da classe PrintWriter exige um tratamento para exceções da classe **FileNotFoundException**.

```
public void gravarArquivo(String texto, String arq) {
  File f = new File(arq);
  try {
    PrintWriter pw = new PrintWriter(f);
    pw.println(texto);
    pw.flush();
    pw.close();
  }
  catch (Exception e) {
    System.out.println("Arquivo não existe: " + e.getMessage());
  }
}
```

O tratamento de exceção é obrigatório, mas pode ser um tratamento **geral** (para a classe **Exception**).

FATEC – Votorantim
Profº Rodrigo Diver

Aula 9 – Serialização e Persistência de Objetos

- **Gravação em arquivos.**

Slides adaptados das notas de aula do Profª Alba Lopes – IFRN

Serialização e Persistência de Objetos

- Para salvar o estado atual de um objeto é necessário serializa-lo antes de registrar em um arquivo.
- A serialização transforma os dados e a estrutura de uma objeto em uma cadeia de bytes.
- Podendo assim ser salva em um arquivo ou banco de dados.
- Para Serializar um objeto em Java é necessário que a classe do objeto implemente a *interface Serializable* do pacote *java.io.Serializable.*

Serialização e Persistência de Objetos

- A *interface Serializable* não possui métodos para se implementar e nem atributos.
- A herança entre classes é naturalmente afetada pela serialização.
- Toda classe que estender de uma classe que implementa a interface **Serializable**, também estará implicitamente marcada como **serializavel.**


```java
public class Motocicleta implements Serializable{
    public String marca;
    public String modelo;
    public int velocidade;

    public void acelerar(int valor){
        velocidade+= valor;
    }
    public void frear(int valor){
        velocidade-= valor;
    }
    public void parar(){
        velocidade = 0;
    }

    /* demais métodos */

}
```

Serialização e Persistência de Objetos

- Se uma classe implementar a interface *Serializable* é possível utilizar métodos para gravar essa classe em um arquivo de texto, por exemplo, bem como ler os dados do arquivo e atribuí-los a objetos.

- O procedimento é simples e pode-se considerar genérico para qualquer classe serializada.

- A classe apresentada a seguir pode ser usada como padrão sempre que desejado gravar um objeto em arquivo e ler os dados para um objeto.

Serialização e Persistência de Objetos

```
public class Serializador {

    public void gravar(String caminho, Object objeto)
            throws FileNotFoundException, IOException{
        FileOutputStream outFile = new FileOutputStream(caminho);
        ObjectOutputStream s = new ObjectOutputStream(outFile);
        s.writeObject(objeto);
        s.close();
    }

    public Object ler(String caminho)
            throws FileNotFoundException, IOException, ClassNotFoundException{
        FileInputStream inFile = new FileInputStream(caminho);
        ObjectInputStream s = new ObjectInputStream(inFile);
        Object objeto = s.readObject();
        s.close();
        return objeto;
    }
}
```

Serialização e Persistência de Objetos

- Algumas classes são fundamentais para esse processo:
  - **ObjectOutputStream:** classe que possui o método (***write***) deserialização. Permite gravar o objeto em arquivo.
  - **ObjectInputStream:** classe que possui o método (***read***) de serialização. Permite ler os dados do arquivo e transformá-lo em **Object**.
  - **FileInputStream** e **FileOutputStream** trabalham diretamente com a gravação e leitura de arquivo em disco.

Serialização e Persistência de Objetos

```
public class Serializador {

    public static void gravar(String caminho, Object objeto)
            throws FileNotFoundException, IOException{
        FileOutputStream outFile = new FileOutputStream(caminho);
        ObjectOutputStream s = new ObjectOutputStream(outFile);
        s.writeObject(objeto);
        s.close();
    }

    public static Object ler(String camin
            throws FileNotFoundException, IOException, ClassNotFoundException{
        FileInputStream inFile = new FileInputStream(caminho);
        ObjectInputStream s = new ObjectInputStream(inFile);
        Object objeto = s.readObject();
        s.close();
        return objeto;
    }
}
```

FileOutputStream cria um arquivo no caminho passado por parâmetro

ObjectOutputStream recebe como parâmetro o arquivo que irá armazenar objetos serializáveis

writeObject grava os bytes referentes ao objeto serializável no arquivo.

Serialização e Persistência de Objetos

```
public class Serializador {

    public static void gravar(String caminho, Object objeto)
            throws FileNotFoundException, IOException{
        FileOutputStream outFile = new FileOutputStream(caminho);
        ObjectOutputStream s = new ObjectOutputStream(outFile);
        s.writeObject(objeto);
        s.close();
    }

    public static Object ler(String caminho)
            throws FileNotFoundException, IOException, ClassNotFoundException{
        FileInputStream inFile = new FileInputStream(caminho);
        ObjectInputStream s = new ObjectInputStream(inFile);
        Object objeto = s.readObject();
        s.close();
        return objeto;
    }
}
```

FileInputStream abre o arquivo que está no caminho passado por parâmetro

ObjectInputStream recebe como parâmetro o arquivo que irá armazenar objetos serializáveis

readObject lê os bytes referentes ao objeto serializável e retorna o objeto.

Gravando o objeto no arquivo “teste.dat”

```
public class TestarGravacaoObj {
    public static void main(String[] args) throws IOException {
        Motocicleta m = new Motocicleta();
        m.setMarca("Honda");
        m.setModelo("Titan");
        m.setVelocidade(10);
        Serializador.gravar("teste.dat", m);
    }
}
```

Como o método foi definido como estático, pode-se chamar o método sem precisar instanciar a classe.

Serialização e Persistência de Objetos

- O arquivo “teste.dat” é salvo na pasta de origem do projeto Java.
- Também é possível passar como parâmetro um caminho completo com o nome do arquivo.

```
// Código para abrir uma janela de seleção de pasta.
String filename = File.separator;
JFileChooser fc = new JFileChooser(new File(filename));
fc.showOpenDialog(jPanel1);
File selFile= fc.getSelectedFile();

//Código para abrir uma janela para salvar um arquivo na pasta
String filename = File.separator;
JFileChooser fc = new JFileChooser(new File(filename));
fc.showSaveDialog(jPanel1);
File selFile= fc.getSelectedFile();
```

Serialização e Persistência de Objetos

Exercícios

Arquivo Cadastro

Equipamentos

Bicicleta - BIC

- BIC-Bicicleta
  - BIC.1-Roda
    - BIC.1.1-Aro
  - BIC.2-Quadro
    - BIC.2.1-Cubo

Componente

Adicionar
Editar
Deletar

Title 1	Title 2	Title 3	Title 4

Serviços

Adicionar
Editar
Deletar

Title 1	Title 2	Title 3	Title 4

Exercícios

1. Implemente os métodos necessários para realizar a edição e remoção de componentes do cadastro.

2. Implemente os métodos necessários para realizar a adição, edição e remoção dos serviços no cadastro de componentes.

3. Implemente os métodos para realizar o cálculo de custo dos componentes e Equipamentos.

4. Adicione as funcionalidades para listar todos os Equipamentos com seus custos organizados em uma tabela.

Bibliografia

- **Java Básico e Orientação a Objeto (livro);**
  - Clayton Escouper das Chagas; Cássia Blondet Baruque; Lúcia Blondet Baruque.
  - Fundação CECIERJ, 2010: https://canal.cecierj.edu.br/012016/d7d8367338445d5a49b4d5a49f6ad2b9.pdf
- **Java e Orientação a Objetos (apostila);**
  - Caelum, 2023: https://www.caelum.com.br/apostila/apostila-java-orientacao-objetos.pdf
- **Modificadores de acesso em Java(artigo);**
  - DevMedia, 2023: https://www.devmedia.com.br/modificadores-de-acesso-em-java/18822
- **Get e Set - Métodos Acessores em Java(artigo);**
  - DevMedia, 2023: https://www.devmedia.com.br/get-e-set-metodos-acessores-em-java/29241
- **Sobrecarga e sobreposição de métodos em orientação a objetos(artigo);**
  - DevMedia, 2023: https://www.devmedia.com.br/sobrecarga-e-sobreposicao-de-metodos-em-orientacao-a-objetos/33066
- **Análise e modelagem de sistemas com a UML: com dicas e exercícios resolvidos (livro);**
  - Luiz Antônio Pereira, 2011: https://luizantoniopereira.com.br/downloads/publicacoes/AnaliseEModelagemComUML.pdf

Bibliografia

- **Herança em Orientação a Objeto(artigo);**
  - DevMedia, 2023: https://www.devmedia.com.br/conceitos-e-exemplos-heranca-programacao-orientada-a-objetos-parte-1/18579