Aula 10

Professores:

Carlos Bazílio
Isabel Rosseti

Tratamento de Exceções

Conteúdo:

- revisão da aula anterior
- motivação
- exceção
- exercício

Revisão da aula anterior

- coleções:
  - listas
  - iteradores
  - conjuntos
  - mapeamentos
  - vetores de tamanho ilimitado
  - vetores
- classes genéricas

Motivação

- um método pode detectar uma falha mas não estar apto a resolver sua causa, devendo repassar essa função a quem saiba

- esta falha, a princípio, não compromete a execução do programa

Exceção

- diz-se que uma exceção é lançada para sinalizar alguma falha
- o lançamento de uma exceção causa uma interrupção abrupta do trecho de código que a gerou
- o controle da execução vai para o primeiro trecho de código (na pilha de chamadas) apto a tratar a exceção lançada

Suporte a exceções

- as linguagens OO tipicamente dão suporte ao uso de exceções
- para usarmos exceções precisamos de:
  - uma representação para a exceção
  - uma forma de lançar a exceção
  - uma forma de tratar a exceção

Exceções em Java

- Java dá suporte ao uso de exceções:
  - são representadas por classes
  - são lançadas pelo comando **throw**
  - são tratadas pela estrutura **try-catch-finally**

Exceções em Java

- de modo geral, um método que lance uma exceção deve declarar isso explicitamente

- para uma classe representar uma exceção, ela deve pertencer a uma certa hierarquia

Exemplo de uso

considere a classe:

```
public class Calc {
  public int div(int a, int b) {
    return a/b;
  }
}
```

o método **div**, se for chamado com **b** igual à zero, dará um erro

esse erro poderia ser sinalizado através de uma exceção

OBS: Na verdade, **b** é um parâmetro de entrada.

Modelando uma exceção

- vamos modelar uma exceção que indica uma tentativa de divisão por zero:

```
public class DivByZero extends Exception {
  public String toString() {
    return "Division by zero.";
  }
}
```

Lançando uma exceção

- agora vamos fazer com que o método **div** lance a exceção que criamos:

```
public class Calc {
  public int div(int a, int b) throws DivByZero {
    if (b == 0) throw new DivByZero();
      return a/b;
  }
}
```

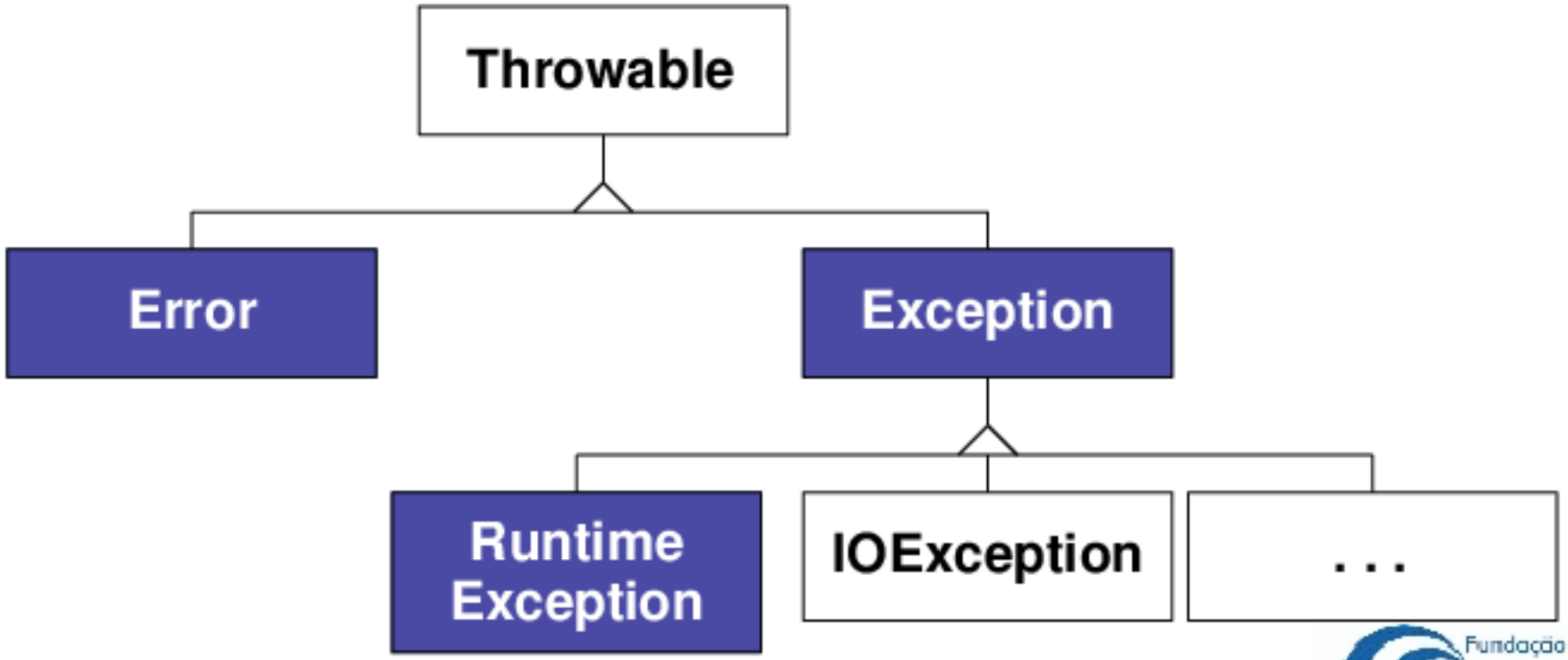
Tratando uma exceção

- podemos, agora, utilizar o método **div** e tratar a exceção, caso esta ocorra:

```
...
Calc calc = new Calc();
try {
   int div = calc.div(x, y);
   System.out.println(div);
} catch (DivByZero e) {
   System.out.println(e);
}
...
```

Tipos de exceções em Java

- Java possui dois tipos de exceções:
  - **checked exceptions**
  - **unchecked exceptions**



Checked exceptions

- são exceções que devem ser usadas para modelar falhas contornáveis
- devem sempre ser declaradas pelos métodos que as lançam
- precisam ser tratadas

Checked exceptions

- para criarmos uma classe que modela uma checked exception, devemos estender a classe **Exception**

- essa exceção será sempre verificada pelo compilador para garantir que seja tratada quando recebida e declarada pelos métodos que a lançam

Unchecked exceptions

- são exceções que devem ser usadas para modelar falhas incontornáveis
- não precisam ser declaradas e nem tratadas

Unchecked exceptions

- para criarmos uma classe que modela uma unchecked exception, devemos estender a classe **Error** ou **RuntimeException**
- esse tipo de exceção não será verificado pelo compilador
- não criamos exceções desse tipo porque elas são usadas pela própria linguagem

Estrutura **try-catch-finally**

- usamos **try-catch** para tratar uma exceção
- a terceira parte dessa estrutura, **finally**, especifica um trecho de código que será *sempre* executado, não importando o que acontecer dentro do bloco **try-catch**
- não é possível deixar um bloco **try-catch-finally** sem executar sua parte **finally**

Exemplo de **try-catch-finally**

```
void readFile(String name) throws IOException {
  FileReader file = null;
  try {
    file = new FileReader(name);
    ... // lê o arquivo
  } catch (Exception e) {
    System.out.println(e);
  } finally {
    if (file != null) file.close();
  }
}
```

Tratando múltiplas exceções

```
try {
   ...
} catch (Exception1 e1) {
   ...
} catch (Exception2 e2) {
   ...
} finally {
   ...
}
```

Exercício: enunciado

Defina um TAD Dicionário através de uma interface Java. Esta interface deve sinalizar, através de exceções, os usos indevidos que forem efetuados pelo cliente.

Um dicionário, também chamado de tabela associativa, é um TAD que permite a armazenagem de valores associados a chaves (você pode encarar um dicionário como um array que é indexado por chaves ao invés de números).

Projete seu dicionário para usar objetos quaisquer como chaves e valores. Escreva a classe DicionarioImpl que implementa sua interface Dicionario, e uma classe de teste que efetue as operações possíveis para todas as classes existentes.

Exercício: solução (interface Dicionario)

```
public interface Dicionario {
 void insere(Object c, Object v) throws DicionarioExc;

 Object consulta(Object c);

 void remove(Object c) throws DicionarioExc;
}
```

Exercício: solução (classe DicionarioExc)

```
public class DicionarioExc extends Exception {
  public DicionarioExc (String msg) {
    super(msg);
  }
}
```

Exercício: solução (classe DicionarioImpl)

```
public class DicionarioImpl
    implements Dicionario {
  class Par {
    Object chave;
    Object valor;
    Par(Object c, Object v) {
      chave = c;
      valor = v;
    }
  };

  Par[] pares;
  int ultimo;
```

Exercício: solução (classe DicionarioImpl)

```
public class DicionarioImpl
    implements Dicionario {
  class Par {
   Object chave;
   Object valor;
   Par(Object c, Object v) {
    chave = c;
    valor = v;
   }
  };

  Par[] pares;
  int ultimo;
```

```
DicionarioImpl(int tamanho) {
 pares = new Par[tamanho];
 ultimo = -1;
}
```

Exercício: solução (classe DicionarioImpl)

```
public class DicionarioImpl
    implements Dicionario {
  class Par {
    Object chave;
    Object valor;
    Par(Object c, Object v) {
      chave = c;
      valor = v;
    }
  };

  Par[] pares;
  int ultimo;
```

```
DicionarioImpl(int tamanho) {
 pares = new Par[tamanho];
 ultimo = -1;
}

public void insere(Object c,
 Object v) throws DicionarioExc{
 if(ultimo == pares.length-1){
  throw new DicionarioExc("Nao
 existe espaco para inserir"+c);
 }
 ++ultimo;
 pares[ultimo] = new Par(c,v);
}
```

Exercício: solução (classe DicionarioImpl)

```
public Object consulta(Object c) {
  for(int i=0; i<=ultimo; i++)
    if(pares[i]!=null && pares[i].chave.equals(c))
      return pares[i].valor;
  return null;
}
```

Exercício: solução (classe DicionarioImpl)

```
  public Object consulta(Object c) {
    for(int i=0; i<=ultimo; i++)
      if(pares[i]!=null && pares[i].chave.equals(c))
        return pares[i].valor;
    return null;
  }

  public void remove(Object c) throws DicionarioExc{
    for(int i=0; i<=ultimo; i++)
      if(pares[i].chave.equals(c)){
        pares[i] = pares[ultimo--];
        return;
      }
    throw new DicionarioExc("Nao foi encontrada a chave "+c);
  }
}
```

Exercício: solução (classe Teste)

```
class Teste {
  public static void main(String[] args) {
    Dicionario d = new DicionarioImpl(2); //espaço para 2 pares
    try{
      d.insere("Jose", "2512-1212");
      d.insere("Joao", "2512-1313");
      d.insere("Maria", "3512-2299");
    } catch (Exception e) {
      System.out.println(e);
    }
    String tel = (String)d.consulta("Joao");
    if(tel != null) System.out.println("tel de Joao e: "+tel);
    else            System.out.println("tel nao foi encontrado");
    tel = (String)d.consulta("Luis");
    if(tel != null) System.out.println("tel de Luis e: "+tel);
    else            System.out.println("tel nao foi encontrado");
  }
}
```