Programação Orientada a Objetos (PO24CP-4CP)

Aula #12 - Pacotes e Visibilidade - Tratamento de Exceções

Profª Luciene de Oliveira Marin
lucienemarin@utfpr.edu.br

Pacotes e Visibilidade

Pacotes e Visibilidade

Para que servem os pacotes?

Pacotes são formas de organizar classes em grupos similares, para efeito de organização.

- São mapeamentos para diretórios (estão diretamente relacionados).
- Nome de pacote não conflita com nomes de classes.

E quando não declarávamos pacotes?

- Todas as classes estavam em um mesmo pacote anônimo, e
- Todas as classes em um mesmo pacote são acessíveis pelas outras.

Pacotes e Visibilidade

Modificadores de acesso *defaul* (vazio) e protected

Seus efeitos de uso surgem quando relacionados a classes de diferentes pacotes.

Modificador de acesso *default* - exemplo

Modificador de acesso *default* - exemplo

```
package DataHora;

public class Hora
{
  byte hora;
  byte minuto;
  byte segundo;

  public Hora(byte h,byte m,byte s)
  {
    hora = h; minuto = m; segundo = s;
  }
  public String toString()
  {
    return hora+":"+minuto+":"+segundo;
  }
} // fim da classe Hora
```

Modificador de acesso *default* - exemplo

```
package DataHora;

public class Data
{
  byte dia;
  byte mês;
  short ano;

  public Data(byte d,byte m,short a)
  {
    dia = d; mês = m; ano = a;
  }

  public String toString()
  {
    return dia+"/"+mês+"/"+ano;
  }

  } // fim da classe Data
```

Modificador de acesso *default* - exemplo

```
package DataHora;

public class DataHora // declaração da classe
{
  Data estaData;
  Hora estaHora;

  public DataHora(byte h,byte min,byte s,byte d,byte m,short a)
  {
    estaData = new Data(d,m,a);
    estaHora = new Hora(h,min,s);
  }

  public String toString()
  {
    String resultado = estaHora.hora+":"+estaHora.minuto+":"+estaHora.segundo;
    resultado += " de "+estaData.dia;
    resultado += " de ";
```

Modificador de acesso *default* - exemplo

```
switch(estaData.mês) // dependendo do valor do campo mês, concatena o nome
    do mês
   {
   case  1: resultado += "Janeiro"; break;
   case  2: resultado += "Fevereiro"; break;
   case  3: resultado += "Março"; break;
   case  4: resultado += "Abril"; break;
   case  5: resultado += "Maio"; break;
   case  6: resultado += "Junho"; break;
   case  7: resultado += "Julho"; break;
   case  8: resultado += "Agosto"; break;
   case  9: resultado += "Setembro"; break;
   case 10: resultado += "Outubro"; break;
   case 11: resultado += "Novembro"; break;
   case 12: resultado += "Dezembro"; break;
   }
  resultado += " de "+estaData.ano;
  return resultado;
  }
} // fim da classe DataHora
```

Modificador de acesso *default* - exemplo

```
/*Esta classe precisa das classes que estão no pacote DataHora*/
import DataHora.*;
/*A classe DemoDataHora demonstra usos de instâncias das classes que fazem parte
do pacote DataHora. Esta classe declara, inicializa e usa algumas instâncias das
classes Data, Hora e DataHora, mas tenta acessar campos destas classes que só
podem ser acessados por classes do mesmo pacote. Esta classe não pode ser compi-
lada por causa de erros intencionais.*/
class DemoDataHora
{  public static void main(String[] argumentos){
    Hora meiodia = new Hora((byte)12,(byte)00,(byte)00);
    Data hoje  = new Data((byte)11,(byte)5,(short)2001);
    DataHora agora = new DataHora((byte)22,(byte)35,(byte)00,
                                  (byte)11,(byte)5,(short)2001);
    System.out.println(meiodia);
    System.out.println(hoje);
    System.out.println(agora);
//Tentamos mudar os campos das classes, que não foram declarados como
//private, mesmo assim causando erros pois a classe DemoDataHora não
//pertence ao mesmo pacote que as classes Data, Hora e DataHora.
    meiodia.segundos = 17;
    hoje.mês = 2;
    DataHora.estaData.ano = 1969;
    } // fim do método main
} // fim da classe DemoDataHora
```

Pacotes e Visibilidade

Efeitos do modificador *default*:

Os campos e métodos declarados com o modificador *default* nas classes do pacote `DataHora` podem ser acessados sem problemas por outros métodos em outras classes no mesmo pacote.

Exemplo:

o método `toString` da classe `DataHora` usa diretamente campos que foram declarados sem modificadores nas classes `Data` e `Hora`.

Fora do pacote...

Outras classes em outros pacotes não tem acesso aos campos declarados como *default*:

- Exemplo: a classe *DemoDataHora*, que tenta acessar diretamente campos das classes `Data`, `Hora` e `DataHora` (erro).

Modificador de acesso protected (protegido)

Modificador de acesso `protected`

Efeitos de uso:

Atributos e **métodos** declarados com o modificador `protected`

Podem ser usados diretamente por **todas** as classes pertencentes ao mesmo pacote.

- Por esta razão não existe sentido em declararmos campos e métodos protegidos em classes de mesmo pacote.

Podem ser usados/acessados diretamente por classes **herdeiras** de **pacotes diferentes**.

- Desta forma, `protected` é menos restrito do que *default*

Pacotes e Visibilidade

- Pacote `geometria`

```
package geometria;
public class Circulo implements ObjetoGeometrico
{
 protected Ponto2D centro;
 protected double raio;
 public Circulo(Ponto2D centro, double raio){
  this.centro = centro;
  this.raio = raio;
 }
 public Ponto2D centro() { return centro; }

 public double calculaArea() { return Math.PI*raio*raio; }

 public double calculaPerimetro() { return 2.0*Math.PI*raio; }

 public String toString(){
  return "Circulo com centro em "+centro+" e raio "+raio;
 }
}
```

Pacotes e Visibilidade

- Pacote `minhaApp`

```
package minhaApp;

import geometria.Circulo;
import geometria.Ponto2D;

public class MeuCirculo extends Circulo
{
  private int id;
  public MeuCirculo(Ponto2D centro, double raio,int id)
  {
     super(centro, raio); // Construtor da superclasse deve ser chamado
     this.id = id;
  }
  public String toString()
  {
    return "Circulo com ID "+id+", centro em "+centro+" e raio "+raio;
  }
}
```

Pacotes e Visibilidade

- Algumas regras são mais complicadas, ex. envolvendo sobreposição (herdeiras não podem ser mais restritas).

```
package minhaApp;
import geometria.Circulo;
import geometria.Ponto2D;
public class CirculoAproximado extends Circulo
{
    // O construtor parametrico deve existir.
    public CirculoAproximado(Ponto2D centro, double raio)
    {
        super(centro, raio);
    }
    public double calculaArea()
    {
        return 3.14*raio*raio;
    }
    protected double calculaPerimetro() // Erro de compilacao!
    {
        return 3.14*Math.PI*raio;
    }
}
```

Regras de sobreposição de métodos:

- Um método declarado como `private` em uma classe ancestral pode ser sobreposto por métodos declarados como `private`, *default*, `protected` ou `public` em uma classe descendente.
- Um método declarado como *default* em uma classe ancestral pode ser sobreposto por métodos declarados como *default*, `protected` ou `public` em uma classe descendente, mas **não** pode ser sobreposto por um método declarado como `private`.

Regras de sobreposição de métodos: (cont.)

- Um método declarado como `protected` em uma classe ancestral pode ser sobreposto por métodos declarados como `protected` ou `public` em uma classe descendente, mas **não** pode ser sobreposto por um método declarado como `private` ou *default*.
- Um método declarado como `public` em uma classe ancestral somente pode ser sobreposto por métodos declarados como `public` em uma classe descendente, **não** podendo ser sobreposto por um método declarado como `private`, *default* ou `protected`.

Pacotes e Visibilidade

- Para compilar:
  - A partir do diretório raiz. Por exemplo:

```
C:\ExemplosPackage>javac geometria/*.java

C:\ExemplosPackage>javac minhaApp/*.java

C:\ExemplosPackage>
```

Pacotes e Visibilidade

Efeito de pacotes e modificadores de acesso a membros de classe:

	`private`	*default*	`protected`	`public`
Visível dentro da mesma classe	sim	sim	sim	sim
Visível dentro do mesmo pacote pela subclasse	não	sim	sim	sim
Visível dentro do mesmo pacote por não subclasses	não	sim	sim	sim
Visível dentro de pacote diferente pela subclasse	não	não	sim	sim
Visível dentro de pacote diferente por não subclasses	não	não	não	sim

- Regra mais simples: usar somente `public` e `private`.
- Considerar necessidade de `protected` - métodos `get` e `set` podem ser usados.

Tratamento de Exceções

Tratamento de Exceções

Exceção - o que é?

Evento que indica a ocorrência de algum problema durante a execução do programa

Tratamento de exceções

- Mecanismo de Java para tratar eventos que impossibilitam a execução normal do programa.
- Assumimos que é possível continuar a execução em alguns casos
  - Assim, permite aos programas **capturar** e **tratar erros** em vez de deixá-los simplesmente ocorrer.
  - Deve ser utilizado em situações em que o sistema pode recuperar-se do mau funcionamento que causou a exceção,
  - Ou ao menos informar precisamente ao usuário o que houve!
- Substituem séries de `if/else`'s

Tratamento de Exceções

Exceção - o que é?

Evento que indica a ocorrência de algum problema durante a execução do programa

Tratamento de exceções

- Mecanismo de Java para tratar eventos que impossibilitam a execução normal do programa.
- Assumimos que é possível continuar a execução em alguns casos
  - Assim, permite aos programas **capturar** e **tratar erros** em vez de deixá-los simplesmente ocorrer.
  - Deve ser utilizado em situações em que o sistema pode recuperar-se do mau funcionamento que causou a exceção,
  - Ou ao menos informar precisamente ao usuário o que houve!
- Substituem séries de `if/else`'s

Em Java

- Em Java, o **tratamento de exceções** foi projetado para situações em que um método detecta um erro e é incapaz de lidar com este
  - Não é possível garantir que existirá algum trecho para tratar a exceção disparada
- Quando um erro ocorre é criado um **objeto de exceção**
  - Contém informações sobre o erro, incluindo seu tipo e o estado do programa quando o erro ocorreu.

Desenvolvendo códigos com tratamento de exceção

- O primeiro passo para tratar exceções é colocar todo o código que possa vir a disparar uma exceção dentro de um bloco **try. . . catch**

```
try{
  instrucoes que possam vir a disparar uma excecao;
}catch(Tipo da excecao){
  instrucoes para lidar com a excecao gerada
}
System.out.println("continuando o programa");
```

- As linhas dentro do bloco **try** são executadas sequencialmente
  - **Se ocorrer** uma exceção, o fluxo de execução passa automaticamente para um bloco **catch**
  - **Se não ocorrer** exceção, então o fluxo de execução passa para a próxima linha após os blocos **catch**

Exemplo 1: Divisão por zero

```
public static void main(String[] args){
 Scanner ler = new Scanner(System.in);
 int a, b, res;

 try{
  a = ler.nextInt();
  b = ler.nextInt();

  res = a / b;

  System.out.println(a + " dividido por " + b + " = " +
    res);

 }catch(Exception e){
    System.err.println("Ocorreu o erro: " + e.toString()
    );
 }
 System.out.println("Fim do programa");
}
```

Determinando o tipo da exceção

- Para cada bloco **try** é possível ter um ou mais blocos **catch**
  - Cada bloco **catch** é responsável por tratar um tipo específico de exceção
- No exemplo anterior, o bloco **catch** capturava a exceção mais genérica possível em Java
  - Capturava objetos da classe `Exception`
- Em Java existem diversas outras classes para exceções, todas herdam da classe `Exception`
  - `ClassNotFoundException`, `ArithmeticException`, `FileNotFoundException`, `...`

Sequência de blocos catch

Deve-se colocar a captura de exceções específicas antes das exceções mais genéricas

Determinando o tipo da exceção

- Para cada bloco **try** é possível ter um ou mais blocos **catch**
  - Cada bloco **catch** é responsável por tratar um tipo específico de exceção
- No exemplo anterior, o bloco **catch** capturava a exceção mais genérica possível em Java
  - Capturava objetos da classe `Exception`
- Em Java existem diversas outras classes para exceções, todas herdam da classe `Exception`
  - `ClassNotFoundException`, `ArithmeticException`, `FileNotFoundException`, ...

Sequência de blocos catch

Deve-se colocar a captura de exceções específicas antes das exceções mais genéricas

Capturando exceções específicas

```
public static void main(String[] args){
  Scanner ler = new Scanner(System.in);
  int a, b, res;
  try{
    a = ler.nextInt();
    b = ler.nextInt();

    res = a / b;

    System.out.println(a + " dividido por " + b + " = "
    + res);
  }catch(java.util.InputMismatchException e){
  System.out.println("Erro: Valores nao inteiros. ");
  }catch(java.lang.ArithmeticException e){
  System.out.println("Erro: Divisao por zero ");
  }catch(Exception e){
  System.out.println("Ocorreu o erro: " + e.toString());
  }
  System.out.println("Fim do programa");
}
```

Bloco finally

- As linhas dentro do bloco **finally** sempre serão executadas, independente de ocorrer exceção ou não
  - O bloco **finally** é o local ideal para colocar o código que liberará os recursos que foram adquiridos em um bloco **try**

Exemplo

Um arquivo é aberto dentro do bloco **try** e o local para fechar este arquivo é dentro do bloco **finally**, pois independente de ocorrer ou não uma exceção após a abertura do arquivo dentro do bloco **try**, este arquivo sempre será fechado.

Bloco finally

```
 System.out.println("Ola mundo");
try{
    System.out.println("Primeira instrucao");
    int a = 10 / 0 ;
    System.out.println("Terceira instrucao");

}catch(Exception e){
   System.out.println("Executada somente se ocorrer
   excecao");

}finally{
 System.out.println("Executa sempre");
}
System.out.println("Executa sempre - fora do bloco");
```

Leitura recomendada

HOSRTMANN, C. S., CORNELL, G. Core Java - 8ª Edição, 2010. Capítulo 11.

Bloco finally

```
 System.out.println("Ola mundo");
try{
    System.out.println("Primeira instrucao");
    int a = 10 / 0 ;
    System.out.println("Terceira instrucao");

}catch(Exception e){
   System.out.println("Executada somente se ocorrer
   excecao");

}finally{
 System.out.println("Executa sempre");
}
System.out.println("Executa sempre - fora do bloco");
```

Leitura recomendada

HOSRTMANN, C. S., CORNELL, G. Core Java - 8ª Edição, 2010. Capítulo 11.