nemo

ontology & conceptual modeling research group

Desenvolvimento OO com Java

8 – Classes Internas

Vítor E. Silva Souza

(vitorsouza@inf.ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

Centro Tecnológico

Universidade Federal do Espírito Santo

Objetivos da parte 8

- Apresentar o conceito de classes internas, explicando seu propósito;
- Mostrar os diferentes tipos de classes internas e como implementar cada tipo em Java.

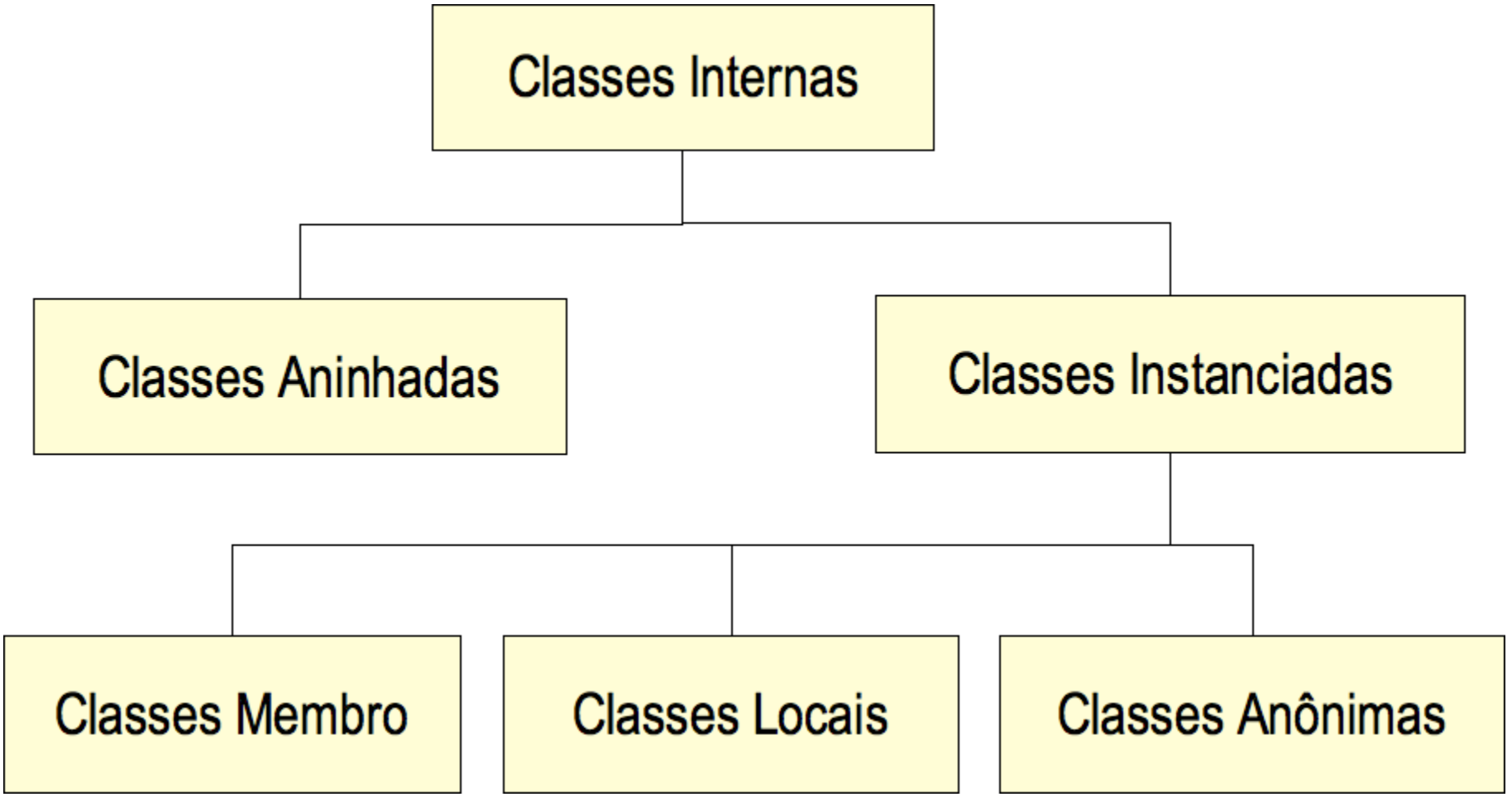
Classes externas e internas

- Até agora definimos classes no nível superior, dentro de pacotes;
- Podemos definir classes dentro de classes, como se fossem atributos ou métodos;
- Vantagens:
  - Legibilidade: agrupamento por similaridade;
  - Ocultamento: podem ser privadas ou protegidas;
  - Redigibilidade: classes internas possuem acesso aos membros privados da classe que a definiu e vice-versa. De fato, classes internas surgiram na versão 1.1 de Java com este propósito.

Código fonte e *bytecode*

- Como são definidas dentro de outras classes, ficam no mesmo arquivo `.java`;
- Ao compilar, gera-se vários arquivos `.class`, compondo o nome da classe externa e interna;
- Ex.:
  - `Externa.java` contém a classe Externa, que define uma classe interna chamada Interna;
  - Ao compilar, gera-se `Externa.class` e `Externa$Interna.class`.
- A classe interna não pode ter o mesmo nome da classe externa que a define.

Tipos de classes internas



Classes aninhadas

- Tipo mais simples de classe interna;
- Classe definida dentro de outra, mas funciona como classe de nível superior;
- Permite definir acesso privado ou protegido e agrupa classes logicamente relacionadas;
- Definida como se fosse um membro `static`;
- Referência via `Externa.Interna`, como se fosse um atributo estático.

Classes aninhadas

```
class Par {
  private Chave chave;
  private Valor valor;

  public Par(Chave chave, Valor valor) {
     this.chave = chave;
     this.valor = valor;
  }

  static class Chave {
     private String nome;
     public Chave(String nome) {
        this.nome = nome;
     }
  }
```

Classes aninhadas

```
  protected static class Valor {
     private int valor;
     public Valor(int valor) {
        this.valor = valor;
     }
  }
}

public class Teste {
  public static void main(String[] args) {
     Par.Chave chave = new Par.Chave("Nota");
     Par.Valor valor = new Par.Valor(10);
     Par par = new Par(chave, valor);
  }
}
```

Classes instanciadas

- Ao contrário das classes aninhadas, classes instanciadas não são `static`;
- Por isso, é preciso ter um objeto da classe externa para referenciarmos à classe interna;
- Podem ser de três tipos:
  - Classes membro;
  - Classes locais;
  - Classes anônimas.

Classes membro

- Definida como um membro (não-estático) da classe;
- Portanto, pode ser pública, privada, protegida ou amiga;
- Tem acesso total a todos os membros (inclusive privados) da classe que a define;
- Para ser criada, é preciso ter uma instância da classe que a define:
  - Referência: `Externa.Interna`;
  - Criação: `ext.new Interna();`

Classes membro

```
class TimeFutebol {
  private Tecnico tecnico;
  private Jogador[] time = new Jogador[11];
  class Pessoa {
     String nome;
     Pessoa(String nome) {
        this.nome = nome;
     }
  }
  private class Tecnico extends Pessoa {
     Tecnico(String nome) {
        super(nome);
     }
  }
```

Classes membro

```
  public class Jogador extends Pessoa {
     public Jogador(String nome) {
        super(nome);
     }
  }
  public void setTecnico(String nome) {
     this.tecnico = new Tecnico(nome);
  }
  public void addJogador(int c, Jogador j) {
     time[c] = j;
  }
}
```

Classes membro

```
public class Teste {
  public static void main(String[] args) {
    TimeFutebol selecao = new TimeFutebol();
    selecao.setTecnico("Felipão");

    TimeFutebol.Jogador jogador;
    jogador = selecao.new Jogador("Júlio César");
    selecao.addJogador(12, jogador);

    jogador = selecao.new Jogador("Fred");
    selecao.addJogador(9, jogador);

    // ...
  }
}
```

Classes membro

- Bastante úteis quando precisamos criar classes que são utilizadas apenas por uma classe;
- Exemplos:
  - Uma lista encadeada e cada nó da lista;
  - Uma janela (GUI) e seus *listeners*;
  - Etc.
- Criam uma nova forma de utilizar `this`:
- `ClasseExterna.this.membro`.

Uso de this

```
class A {
  public String nome = "a";
  public String sobrenome = "s";
  public class B {
     public String nome = "b";
     public class C {
        public String nome = "c";
        public void imprime() {
           System.out.println(nome);
           System.out.println(this.nome);
           System.out.println(C.this.nome);
           System.out.println(B.this.nome);
           System.out.println(A.this.nome);
           System.out.println(sobrenome);
        }
     }
  }
}
```

Uso de this

```
public class Instanciadas {
  public static void main(String[] args) {
     A a = new A();
     A.B b = a.new B();
     A.B.C c = b.new C();
     c.imprime();
  }
}

// Resultado: c, c, c, b, a, s
```

Classes membro e interfaces

- Estratégia para esconder a classe interna e ainda assim utilizá-la externamente;
- Define-se a classe membro como `private` ou `protected`;
- Define-se sua interface como interface de nível superior (`public` ou *package-private*);
- Faz a classe interna implementar a interface;
- Adiciona um método na classe externa para retornar uma instância da classe interna, com *upcasting* para a interface.

Classes membro e interfaces

```
interface Bicho {
  String getNome();
}

class PetShop {
  private class Gato implements Bicho {
     String nome;
     public Gato(String nome) {
        this.nome = nome;
     }
     public String getNome() {
        return nome;
     }
  }
```

Classes membro e interfaces

```
  public Bicho comprar(String nome) {
     return new Gato(nome);
  }
}

public class Teste {
  public static void main(String[] args) {
     PetShop loja = new PetShop();
     // Erro: PetShop.Gato bicho;
     Bicho bicho = loja.comprar("Garfield");
     System.out.println(bicho.getNome());
  }
}
```

Classes locais

- Classes declaradas como variáveis locais: dentro de blocos;
- Diferenças com relação à classes membro:
  - Assim como variáveis locais, não podem ter especificador de acesso;
  - Escopo de visibilidade: só podem ser usadas dentro do bloco nas quais foram definidas;
  - Acesso: possuem acesso às variáveis e parâmetros do bloco, desde que sejam declarados como final.
- Pode ser usada a mesma estratégia com interface explicada para classes membro.

Classes locais

```
public class Teste {
  static void f(final double d, long l) {
    final String s = "Pode";

    class Impressora {
      void imprimir() {
        System.out.println(d);
        System.out.println(s);
        // Erro: System.out.println(l);
        // Erro: System.out.println(r);
      }
    }
    final String r = "Não pode";
    Impressora imp = new Impressora();
    imp.imprimir();
  }
```

Classes locais

```
  public static void main(String[] args) {
     f(1.5, 1000);

     // Erro: Impressora imp2;
     // Erro: imp2 = new Impressora();
  }
}
```

Classes anônimas

- Semelhante às classes locais:
  - Mesmas regras (de escopo, de acesso, ...);
  - Não possuem nome;
  - Não podem definir construtores;
  - Só permitem a criação de uma instância.

Classes anônimas

```
interface Bicho {
  String getNome();
}

class PetShop {
  public Bicho comprar(final String nome) {
     return new Bicho() {
        public String getNome() {
           return nome;
        }
     };      // Note o ";"
  }
}
```

Classes anônimas

```
public class Teste {
  public static void main(String[] args) {
     PetShop loja = new PetShop();
     Bicho bicho = loja.comprar("Odie");
     System.out.println(bicho.getNome());
  }
}
```

Sintaxe complexa

- Classes anônimas são criadas a partir de outras classes ou interfaces;
- Sintaxe estranha à primeira vista, porém promove grande redigibilidade;
  - A adição em complexidade é compensada pela facilidade na escrita de códigos muito comuns, principalmente na construção de interfaces gráficas.
- É como se criássemos uma classe local que estendesse uma outra ou implementasse uma interface.

Código equivalente com classe local

```
class PetShop {
  public Bicho comprar(final String nome) {
     class BichoAnonimo implements Bicho {
        public String getNome() {
           return nome;
        }
     }
     return new BichoAnonimo();
  }
}
```

Nomes dos arquivos de bytecode

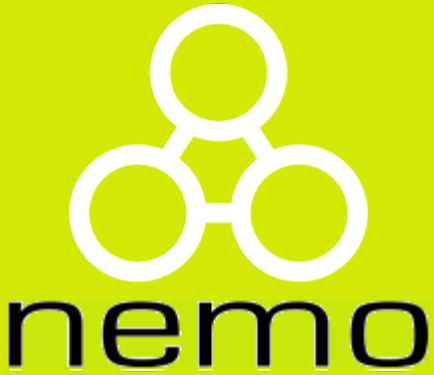
- Como classes internas não possuem nome, Java usa números inteiros para os nomes dos arquivos `.class`:
  - `PetShop.class` contém a classe `PetShop`;
  - `PetShop$1.class` contém a classe anônima.

Níveis de aninhamento

- Classes aninhadas podem, por sua vez, definir classes internas aninhadas;
- `Par.Chave.Subchave`, etc.;
- Classes instanciadas (internas e não-aninhadas) não podem definir classes aninhadas;
- Classes instanciadas podem definir classes internas também instanciadas

Quando usar?

- Classes Aninhadas:
  - Agrupar classes logicamente relacionadas.
- Classes Membro:
  - Esconder uma classe que é utilizada somente dentro de uma outra classe.
- Classes Locais:
  - Quando o escopo é um bloco e são criadas várias instâncias.
- Classes Anônimas:
  - Quando o escopo é um bloco e é criada apenas uma instância.



http://nemo.inf.ufes.br/