nemo

ontology & conceptual modeling research group

Desenvolvimento OO com Java

7 – RTTI e Interfaces

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 7

- Apresentar os conceitos relacionados à identificação de tipos em tempo de execução (Run-Time Type Identification – RTTI);
- Mostrar como a palavra reservada `interface` permite a criação de classes abstratas puras (interfaces);
- Explicar como Java pode simular herança múltipla usando interfaces.

Polimorfismo e extensão

- Com polimorfismo, podemos esquecer a classe de um objeto e trabalhar com a superclasse:
  - A interface de ambas é a mesma;
  - A amarração dinâmica garante que o método da classe correta será executado.
- O que acontece se a subclasse estende a superclasse (adiciona mais funcionalidade)?
- Se a superclasse não possui aquela funcionalidade, não podemos chamá-la!

Polimorfismo e extensão

```
abstract class Animal {
  public abstract void comer();
}
class Cachorro extends Animal {
  public void comer() {
     System.out.println("Comendo um osso...");
  }
  public void latir() {
     System.out.println("Au Au!");
  }
}
class Gato extends Animal {
  public void comer() {
     System.out.println("Comendo um peixe...");
  }
  public void miar() {
     System.out.println("Miau!");
  }
}
```

Polimorfismo e extensão

```
public class Teste {
  public static void main(String[] args) {
     Animal[] vet = new Animal[] {
          new Cachorro(), new Gato(),
          new Gato(), new Cachorro()
     };
     for (int i = 0; i < vet.length; i++) {
        vet[i].comer();
        // Erro: vet[i].latir();
     }
  }
}
```

Estreitamento (*downcast*)

- Precisamos relembrar a classe específica do objeto para chamarmos métodos que não estão na interface da superclasse;
- Para isso faremos estreitamento:

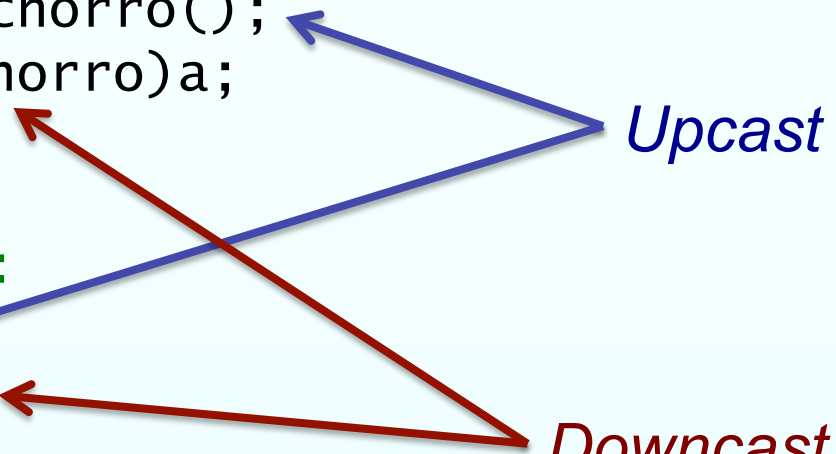
Ampliação (*upcast*)	Estreitamento (*downcast*)
`int` para `long`	`long` para `int`
`float` para `double`	`double` para `float`
`Cachorro` para `Animal`	`Animal` para `Cachorro`
`Gato` para `Animal`	`Animal` para `Gato`

Upcast vs. *downcast*

- Ampliação é automática e livre de erros:
  - A classe base não pode possuir uma interface maior do que a classe derivada;
  - Não é necessário explicitar o *upcast*.
- Estreitamento é manual e pode causar erros:
  - A classe base pode ter várias subclasses e você está convertendo para a classe errada;
  - É necessário explicitar o *downcast*;
  - Pode lançar um erro (`ClassCastException`);
  - Pode haver perda de informação (tipos primitivos).

Upcast vs. *downcast*

```
public class Teste {
  public static void main(String[] args) {
     Animal a = new Cachorro();
     Cachorro c = (Cachorro)a;
     c.latir();

     // Forma resumida:
     a = new Gato();
     ((Gato)a).miar();
  }
}
```



RTTI: Run-Time Type Identification

- O mecanismo que verifica o tipo de um objeto em tempo de execução chama-se RTTI;
- RTTI = Run-Time Type Identification ou Identificação de Tipos em Tempo de Execução;
- Este mecanismo garante que as conversões são sempre seguras;
- Não permite que um objeto seja convertido para uma classe inválida:
  - Fora da hierarquia: erro de compilação;
  - Dentro da hierarquia: erro de execução.

RTTI: Run-Time Type Identification

```
public class Teste {
  public static void main(String[] args) {
     Animal a = new Cachorro();

     // Sem erro nenhum:
     Cachorro c = (Cachorro)a;

     // Erro de execução (ClassCastException):
     Gato g = (Gato)a;

     // Erro de compilação:
     String s = (String)a;
  }
}
```

O operador `instanceof`

- O mecanismo de RTTI permite que você consulte se um objeto é de uma determinada classe;
- Operador `instanceof`:
  - Sintaxe: `<objeto> instanceof <Classe>`
  - Retorna `true` se o objeto for instância (direta ou indireta) da classe especificada;
  - Retorna `false` caso contrário.

O operador instanceof

```
public class Teste {
  public static void main(String[] args) {
     Animal[] vet = new Animal[] {
          new Cachorro(), new Gato(),
          new Gato(), new Cachorro()
     };
     for (int i = 0; i < vet.length; i++) {
        if (vet[i] instanceof Cachorro)
           ((Cachorro)vet[i]).latir();
        else if (vet[i] instanceof Gato)
           ((Gato)vet[i]).miar();
     }
  }
}
```

O uso de `instanceof` deve ser raro

- Não é uma boa prática usar `instanceof`:
  - Use polimorfismo;
  - Use classes genéricas (veremos adiante).
- Use `instanceof` apenas quando não há outra solução.

Trocando instanceof por polimorfismo

```
abstract class Animal {
  public abstract void comer();
  public abstract void falar();
}

class Cachorro extends Animal {
  public void comer() { /* ... */ }
  public void falar() { /* ... */ }
}

class Gato extends Animal {
  public void comer() { /* ... */ }
  public void falar() { /* ... */ }
}
```

Trocando instanceof por genéricos

```
public class Teste {
  public static void main(String[] args) {
    Cachorro c;

    List lista = new ArrayList();
    lista.add(new Cachorro());
    Object o = lista.get(0);
    if (o instanceof Cachorro) c = (Cachorro)o;

    // Com genéricos.
    List<Cachorro> listaGen;
    listaGen = new ArrayList<Cachorro>();
    listaGen.add(new Cachorro());
    c = listaGen.get(0);
  }
}
```

Interfaces

- Uma classe abstrata é pura quando:
  - Possui métodos abstratos;
  - Não possui métodos concretos;
  - Não possui atributos.
- Java oferece a palavra reservada `interface`:
  - Cria uma classe abstrata pura;
  - Chamaremos pelo nome de interface;
  - Ao conversar com outros programadores, cuidado para não confundir com “interface com o usuário”.

Características

- Estabelece a interface (o contrato) de um conjunto de classes;
- Permite a construção de código genérico:
  - Trabalha com qualquer objeto que implemente a interface;
  - Obriga programadores a implementar determinados métodos em suas classes para usar seu código.
- Podem ser públicas ou amigas (mesma regra das classes);
- Classes utilizam `implements` ao invés de `extends` para implementar uma interface.

Interfaces

```
interface Forma {
  void desenhar();
  void aumentar(int t);
}

abstract class Poligono implements Forma {
  private int lados;
  public Poligono(int lados) {
     this.lados = lados;
  }
  public int getLados() { return lados; }
  public abstract void pintar(int cor);
}
```

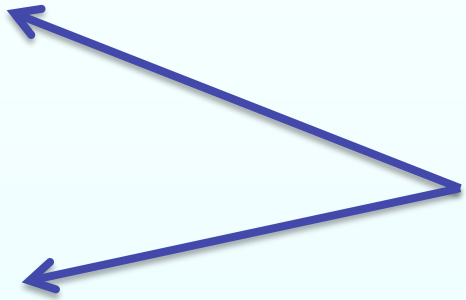
Interfaces

```
class Linha implements Forma {
  private int x1, y1, x2, y2;

  public void desenhar() {
     /* ... */
  }

  public void aumentar(int t) {
     /* ... */
  }
}
```

Membros da interface

- Métodos definidos na interface são automaticamente públicos;
- Atributos definidos na interface são automaticamente públicos e estáticos.

```
interface Forma {
  int x;
  void desenhar();
}

interface Forma {
  public static int x;
  public void desenhar();
}
```

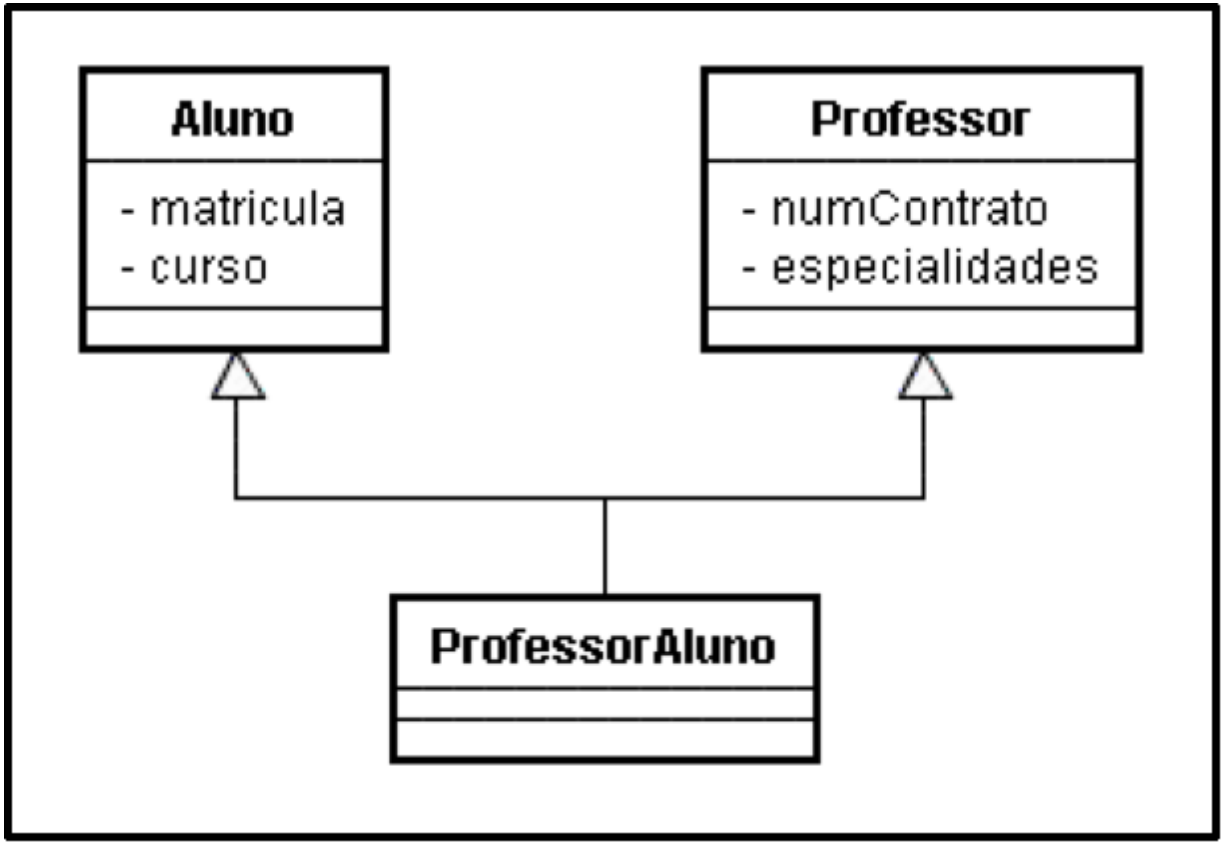


Membros da interface

```
interface Forma {
  void desenhar();
  void aumentar(int t);
}

class Linha implements Forma {
  // Erro: reduziu de público para amigo!
  void desenhar() {
     /* ... */
  }

  // Erro: reduziu de público para privado!
  private void aumentar(int t) {
     /* ... */
  }
}
```

Herança múltipla

- Algumas vezes queremos herdar comportamento de duas classes:



Herança múltipla

- Herança múltipla pode trazer problemas:
  - Colisão de nomes;
  - Herança repetida.
- Java prima pela simplicidade e não permite herança múltipla:

```
// Isto não existe em Java!
class ProfessorAluno extends Professor, Aluno
{

}
```

Herança múltipla com interfaces

- Interfaces permitem que simulemos a herança múltipla:

```
interface VeiculoComPortas {
  void abrirPortas();
}

interface VeiculoComPortaMalas {
  void abrirPortaMalas();
}

interface VeiculoComTetoSolar {
  void abrirTetoSolar();
}
```

Herança múltipla com interfaces

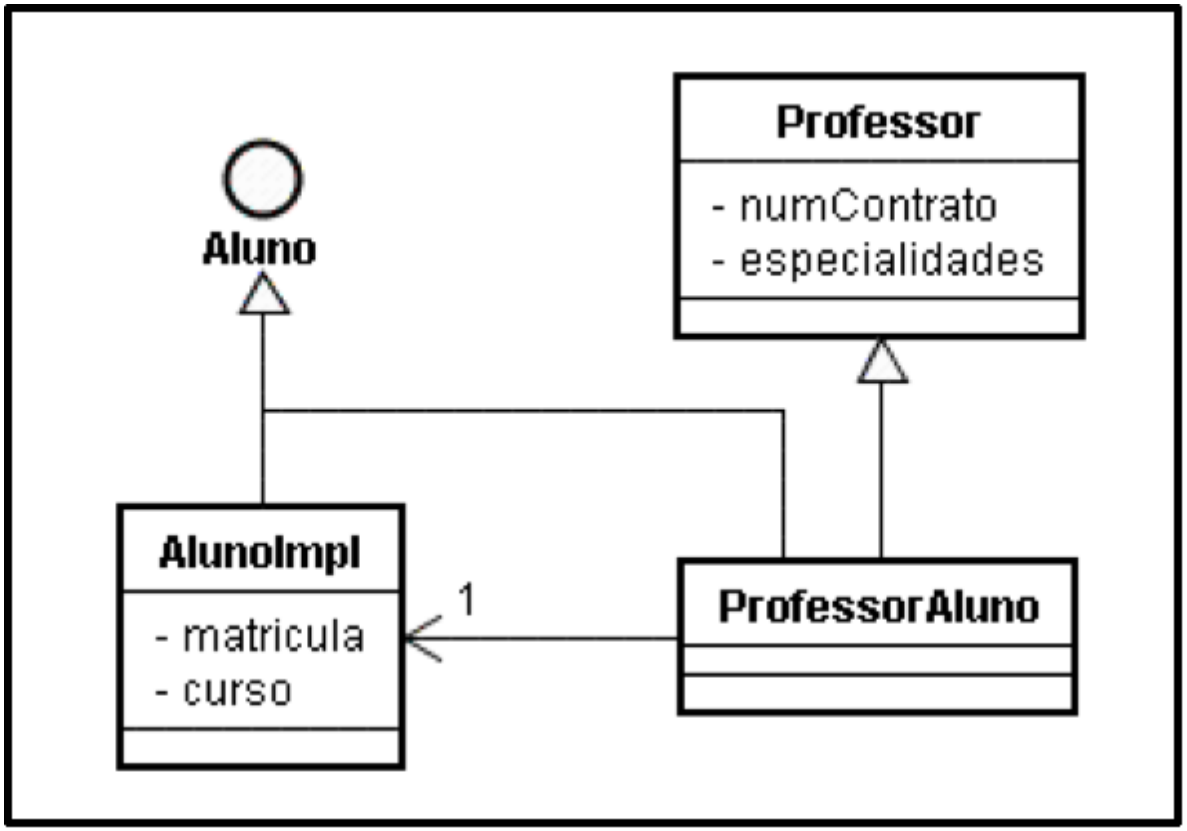
```
class Carrao implements VeiculoComPortas,
            VeiculoComPortaMalas, VeiculoComTetoSolar {
  public void abrirPortas() { }
  public void abrirPortaMalas() { }
  public void abrirTetoSolar() { }
  public void andar() { }
}

public class Teste {
  static void ap(VeiculoComPortas v) {
    v.abrirPortas();
  }

  static void apm(VeiculoComPortaMalas v) {
    v.abrirPortaMalas();
  }
```

Herança múltipla com interfaces

```
  static void ats(VeiculoComTetoSolar v) {
     v.abrirTetoSolar();
  }

  static void a(Carrao v) {
     v.andar();
  }

  public static void main(String[] args) {
     Carrao c = new Carrao();
     ap(c);    // Upcast: VeiculoComPortas
     apm(c);   // Upcast: VeiculoComPortaMalas
     ats(c);   // Upcast: VeiculoComTetoSolar
     a(c);     // Sem upcast.
  }
}
```

Solução para herança múltipla

- Escolha uma das classes concretas para herdar;
- Para as demais, crie interfaces e implemente-as;
- Use composição para fazer reuso.



Interface ou classe abstrata?

- Sempre que possível, use interfaces;
- Lembre-se que Java não suporta herança múltipla:
  - Quando um objeto estende uma classe, não poderá estender nenhuma outra;
  - Não "queime" a herança sem necessidade!

Colisão de nomes em interfaces

- Assim como na herança múltipla, pode haver colisão de nomes em interfaces:
  - Não há colisão de implementação (simplifica);
  - Especificadores de acesso podem colidir;
  - Tipos de retorno podem colidir.

```
interface Forma {
  void desenhar();
  void inverter();
}

class UmaForma implements Forma {
  protected void desenhar() { }    // Erro!
  public int inverter() { }        // Erro!
}
```

Hierarquias de interface

- Uma interface pode estender outra:

```
interface Forma {
  void desenhar();
  void aumentar(int t);
}

interface FormaInversivel extends Forma {
  void inverter();
}

class UmaForma implements Forma { /* ... */ }

class OutraForma implements FormaInversivel
{ /* ... */ }
```

A interface Comparable

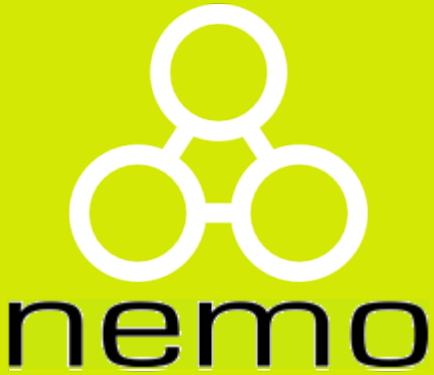
- Um exemplo de interface na API Java é a interface `Comparable`;
- Define o método `compareTo(Object obj)`:
  - Compara o objeto atual (`this`) com o objeto informado (`obj`);
  - Retorna 0 se `this` = `obj`;
  - Retorna um número negativo se `this` < `obj`;
  - Retorna um número positivo se `this` > `obj`.
- Métodos genéricos a utilizam para ordenar coleções de elementos.

A interface Comparable

```
import java.util.Arrays;

class Valor implements Comparable {
  int valor;
  public Valor(int v) { valor = v; }

  public int compareTo(Object obj) {
     return valor - ((Valor)obj).valor;
  }

  public String toString() {
     return "" + valor;
  }
}
```

A interface Comparable

```
public class Teste {
  static void imprimir(Object[] vetor) {
    for (int i = 0; i < vetor.length; i++)
      System.out.print(vetor[i] + "; ");
    System.out.println();
  }
  public static void main(String[] args) {
    Valor[] vetor = new Valor[] {
      new Valor(10), new Valor(3),
      new Valor(15), new Valor(7)
    };
    imprimir(vetor);   // 10; 3; 15; 7;
    Arrays.sort(vetor);
    imprimir(vetor);   // 3; 7; 10; 15;
  }
}
```



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