

Programação OO em Java

Profa Andréa Schwertner Charão

Aluno Alberto Francisco Kummer Neto

DLSC/CT/UFSM

Revisão das aulas 00 e 01

- Herança
- Polimorfismo
- Classes abstratas
- Interfaces
- Tipos genéricos/paramétricos
- Coleções

Cuidado!

- Nos próximos slides, parte dos códigos não estão visíveis.

Corpo suprimido: note a linha pontilhada abaixo do método.

No Netbeans, esse recurso se chama “code folding”.

```
public class Pessoa {  
    protected String nome;  
  
    public Pessoa() {  
        .....  
    }  
    public Pessoa(String nome) {  
        .....  
    }  
    public void mostra() {  
        .....  
    }  
  
    public class Estudante extends Pessoa {  
        private String curso;  
  
        public Estudante() {  
            .....  
        }  
        public Estudante(String nome, String curso) {  
            .....  
        }  
        @Override  
        public void mostra() {  
            .....  
        }  
    }  
}
```

Herança: Relação “IS-A” (I)

Superclasse de Estudante

Herda de Pessoa

Lembre-se dos
modificadores de
visibilidade!

```
public class Pessoa {  
    protected String nome;  
  
    public Pessoa() {  
  
    public Pessoa(String nome) {  
  
    public void mostra() {  
    }  
}  
  
public class Estudante extends Pessoa {  
    private String curso;  
  
    public Estudante() {  
  
    public Estudante(String nome, String curso) {  
  
    @Override  
    public void mostra() {  
    }  
}
```

Herança: Relação “IS-A” (II)

Chama mostra() de **Estudante**: o tipo da variável e é Estudante.

```
public static void main (String[] args) {  
    Estudante e = new Estudante("Jose", "Direito");  
    Pessoa refp = e;  
    e.mostra();  
    refp.mostra();  
}
```

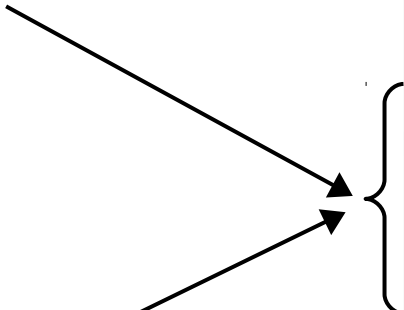
Chama mostra() de **Pessoa**: o tipo da variável refp é Pessoa.

**Exemplo de
polimorfismo
dinâmico!**

Polimorfismo (I)

Três formas diferentes de setar o horário.

Note que os métodos apresentam o mesmo nome.



```
class Relogio {  
    private int horas, minutos;  
    public Relogio() {  
        horas = minutos = 0;  
    }  
    public setHorario(int horas, int minutos) {  
  
    }  
    public setHorario(String relógio) {  
  
    }  
    public setHorario(long millis) {  
  
    }  
  
    public static void main(String[] args) {  
        Relogio r = new Relogio();  
        r.setHorario(12, 26);  
        r.setHorario("12:26");  
        r.setHorario(System.currentTimeMillis());  
    }  
}
```

Classes abstratas (I)

Classe marcada como
abstrata

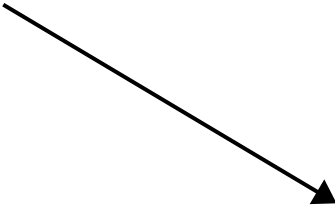
```
public abstract class AbstractVehicle {  
  
    public abstract float getTax();  
  
    public float getCost(int minutes) {  
        return getTax() * (minutes / 60);  
    }  
  
}
```

Herdeiros **devem**
implementar o método
getTax()

- Estratégia comum para a construção de bibliotecas
- O programador é responsável por completar o comportamento da classe
- A classe abstrata pode apresentar um comportamento “default”

Classes abstratas: Polimorfismo (II)

Implementa o comportamento faltante de AbstractVehicle



```
public abstract class AbstractVehicle {  
    public abstract float getTax();  
  
    public float getCost(int minutes) {  
        return getTax() * (minutes / 60);  
    }  
}  
  
public class Motorcycle {  
    public float getTax() {  
        return 3.0f; // Taxa paga por hora para motos.  
    }  
}
```


Classes abstratas II

- Métodos abstratos só podem ser declarados em classes abstratas
- Em geral, classes abstratas também possuem métodos concretos
- Se uma classe só tem métodos abstratos, é melhor declará-la como **interface**

Erro: classe não abstrata com método abstrato

```
class Bicho
{
    protected String nome;
    public Bicho(String nome)
    {
        this.nome = nome;
    }
    abstract public String som();
}
```

mensagem do compilador:

```
Bicho.java: Bicho is not abstract and does
not override abstract method som() in
Bicho
class Bicho
^
1 error
```

Interfaces

- São um tipo de encapsulamento contendo principalmente métodos
- Definem um conjunto de métodos (comportamento) que devem ser implementados em classes que herdam a interface

```
interface Matricial
{
    public void transpoe();
    public void inverte();
}
```

Implementando interfaces (I)

- Usar a palavra-chave **implements**
- Todos os métodos da interface devem ser implementados!

```
class MatrizEsparsa
    implements Matricial
{
    public void transpoe()
    { ... }
    public void inverte()
    { ... }
}
```

Implementando interfaces (II)

ArrayList e **Vector** tem o **comportamento** em comum: implementam a interface **List**.

```
{ import java.util.List;
  import java.util.ArrayList;
  import java.util.Vector;

  public class Interfaces {
    public static void main {
      List <String> listArray = new ArrayList <> ();
      listArray.add("Jose");
      System.out.println(listArray.get(0));

      List <String> vectorArray = new ArrayList<> ();
      vectorArray .add("Jose");
      System.out.println(vectorArray .get(0));
    }
  }
```

Semântica de herança se aplica a interfaces.

Mais sobre interfaces (I)

- Java suporta "herança múltipla" de interfaces, mas não de classes

```
class A {...}  
interface B {...}  
interface C {...}  
  
class X extends A  
implements B,C  
{...}
```

Mais sobre interfaces (II)

- Atributos declarados em interfaces são implicitamente `public static final` (constantes)
- Métodos declarados em interfaces são implicitamente `public abstract`

Veja mais sobre interfaces em:

<http://download.oracle.com/javase/tutorial/java/concepts/interface.html>

Tipos genéricos (I)

- Classes genéricas definidas em função de algum parâmetro (tipos parametrizáveis)
- Usa a notação “diamante” para representar o tipo parametrizável
- Largamente conhecido como **Polimorfismo paramétrico**

Parâmetros de métodos também possuem tipos!

```
public setHorario(int horas, int minutos) {
```

Tipos genéricos (II)

Marcação *diamond* indica que a classe usa tipo genérico.

Atributos x e y da classe são do tipo genérico T.

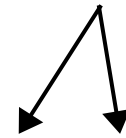
```
public class Ponto2D <T> {  
    private T x;  
    private T y;  
  
    void set(T x, T y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    T getX() {  
        return x;  
    }  
  
    T getY() {  
        return y;  
    }  
}
```

T pode ser o tipo de argumento de métodos...

... assim como tipo de retorno!

Tipos genéricos (III)

É possível utilizar mais de um tipo paramétrico.



```
public class Par <K, V> {  
    protected K chave;  
    protected V valor;  
  
    Par (K chave, V valor) {  
  
        K getChave() {  
  
        V getValor() {  
  
        void setValor(V valor) {  
    }  
}
```

Usando tipos genéricos

- Para usar o tipo, define-se o parâmetro específico

O tipo paramétrico deve ser um tipo de objeto!

Tipos primitivos não podem ser utilizados!

~~int, float, char~~

```
public static void main(String[] args) {  
    Ponto2D <Float> pFloat = new Ponto2D <> ();  
    pFloat.set(0.4f, 0.1f);  
    System.out.printf("x=%.2f, y=%.2f\n",  
        pFloat.getX(),  
        pFloat.getY());  
  
    Ponto2D <Integer> pInteger = new Ponto2D <>  
        ();  
    pInteger.set(10, 5);  
    System.out.printf("x=%d, y=%d\n",  
        pInteger.getX(),  
        pInteger.getY());  
}
```

Veja mais em:

<http://download.oracle.com/javase/tutorial/java/generics/index.html>