

Computação II – Orientação a Objetos

Fabio Mascarenhas - 2016.1

<http://www.dcc.ufrj.br/~fabiom/java>

Interfaces

- Uma *interface* é uma forma abstrata de descrever um objeto
- A classe fixa a forma de um objeto e as assinaturas e as implementações das suas operações
- Uma interface fixa apenas as assinaturas das operações
- Sintaticamente, uma interface tem apenas declarações de métodos, sem corpo

```
public interface Tijolo {  
    boolean testaColisao(Bola bola);  
    void desenhar(Tela t);  
    int getPontos();  
}
```

Interfaces e abstrações

- Interfaces são uma ferramenta poderosa de *abstração*: representar um conceito pelas suas características essenciais
- Com elas, podemos decompor nossos problemas em pequenas partes genéricas
- Vamos ver um exemplo prático de como mesmo uma interface simples pode ser combinada de maneiras poderosas:

```
public interface Funcao {  
    double getValor(double x); // f(x)  
    String getFormula(); // x^2, 2*x, 5*x + 2*x^2  
}
```

Funções simples

- A função *constante* sempre retorna o mesmo valor

```
public class Const implements Funcao {  
    double c;  
    public Const(double _c) { c = _c; }  
    public double getValor(double x) { return c; }  
    public String getFormula() { return ""+c; }  
}
```

- A função *potência* retorna o x elevado a algum n

```
public class Pot implements Funcao {  
    double n;  
    public Pot(double _n) { n = _n; }  
    public double getValor(double x) { return Math.pow(x, n); }  
    public String getFormula() { return "x^"+n; }  
}
```

- A função *exponencial* retorna algum n elevado a x

```
public class Exp implements Funcao {  
    double b;  
    public Exp(double _b) { b = _b; }  
    public double getValor(double x) { return Math.pow(b, x); }  
    public String getFormula() { return ""+b+"^x"; }  
}
```

Polimorfismo

- Nem todas as linguagens orientadas a objeto possuem interfaces como as de Java, mas todas elas permitem o *polimorfismo* que obtemos com interfaces
- Polimorfismo é poder operar com objetos diferentes de maneira uniforme, mesmo que cada objeto implemente a operação de uma maneira particular; basta que a assinatura da operação seja a mesma para todos os objetos
- Em programas OO reais, é muito comum que todas as operações sejam chamadas em referências para as quais só vamos saber qual classe concreta o objeto vai ter em tempo de execução
- Vamos ver muitas aplicações diferentes de polimorfismo ao longo do curso

Polinômios

- Podemos aplicar um *fator de escala* a uma função para obter uma nova função:

```
public class Escala implements Funcao {
    double c; Funcao f;
    public Escala(double _c, Funcao _f) { c = _c; f = _f; }
    public double getValor(double x) {
        return c * f.getValor(x);
    }
    public String getFormula() { return ""+c+"*"+f.getFormula(); }
}
```

- Podemos criar uma nova função a partir da soma de várias funções:

```
public class Soma implements Funcao {
    Funcao[] fs;
    public Soma(Funcao... _fs) { fs = _fs; }
    public double getValor(double x) {
        double v = 0.0;
        for(Funcao f: fs) { v = v + f.getValor(x); }
        return v;
    }
    public String getFormula() { ... }
}
```

Varargs

- No construtor de Soma, usamos a notação `Funcao...` para indicar que queremos receber um vetor de funções
- Essa é uma notação especial que permite não apenas passar um vetor explicitamente, mas também passá-lo de forma implícita, com cada elemento como um argumento diferente:

```
Soma s = new Soma(f1, f2, f3);
```

```
new Soma ( new Funcao [] { f1, f2, f3 } )
```

- Essa notação se chama “varargs”, e só podemos usar ela no último parâmetro de um construtor, método ou função

Decoradores e Compósitos

- Apenas as funções Const, Pot e Exp são implementações primitivas de Funcao no nosso exemplo
- As outras são funções construídas em cima de outras funções, que aproveitam o polimorfismo para ter um número ilimitado de combinações recursivas
- Elas também são demonstrações de outros dois padrões de programação OO muito comuns, baseados em polimorfismo: decoradores e *compósitos*

Decorador

- Um [decorador](#) é um objeto que modifica o comportamento de outro objeto, expondo a mesma interface
- Um decorador implementa uma interface, e contém uma instância dessa mesma interface, delegando seus métodos para essa instância, mas sempre acrescentando alguma coisa
- Escala é um exemplo de decorador
- Outro é a Derivada de uma função
- A biblioteca padrão de Java faz uso extenso de decoradores no seu sistema de entrada e saída

```
public class Derivada implements Funcao {
    double dx = 0.000001;
    Funcao f;
    public Derivada(Funcao _f) { f = _f; }
    public double getValor(double x) {
        return (f.valor(x+dx)-f.valor(x))/dx;
    }
    public String getFormula() { ... }
}
```

Compósito

- Um compósito é uma composição em que as partes são do mesmo tipo do todo
- O compósito implementa a mesma interface das suas partes, e as implementações de seus métodos são uma combinação dos resultados de delegar os métodos para as partes
- Soma é um compósito, assim como a função $f \circ g$ formada pela *composição* de duas funções
- Compósitos são uma escolha natural para representar estruturas em árvore: uma janela de uma interface gráfica é um exemplo bem complexo de um compósito

```
public class Composta implements Funcao {
    Funcao f, g;
    public Composta(Funcao _f, Funcao _g) { f = _f; g = _g;}
    public double getValor(double x) { return g.valor(f.valor(x)); }
    public String getFormula() { ... }
}
```

Recursão estrutural

- Decoradores e compósitos são exemplos de *recursão estrutural*
- Recursão estrutural aparece sempre que uma ou mais partes de um objeto são similares ao todo
- Chamamos métodos que operam sobre essas partes de *métodos recursivos*
- Os métodos `getValor` das classes `Escala`, `Derivada`, `Soma` e `Composta` são exemplos de métodos recursivos