nemo

ontology & conceptual modeling research group

Desenvolvimento OO com Java

6 – Polimorfismo

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 6

- Entrar em detalhes no conceito de polimorfismo, muito importante para a orientação a objetos:
  - Hierarquias polimórficas;
  - Classes e métodos abstratos;
  - Polimorfismo com `java.lang.Object`.
- Desta forma:
  - Aumentar ainda mais a capacidade de abstração para construção de softwares mais modulares e reutilizáveis.

Polimorfismo

- Do grego poli + morphos = múltiplas formas;
- Característica OO na qual se admite tratamento idêntico para objetos diferentes baseado em relações de semelhança;
- Em outras palavras, onde uma classe base é esperada, aceita-se qualquer uma de suas subclasses.

Relembrando o exemplo da parte 5

```
class Forma {
  public void desenhar() {
     System.out.println("Forma");
  }
}
class Circulo extends Forma {
  public void desenhar() {
     System.out.println("Círculo");
  }
}
class Quadrado extends Forma { /* ... */ }
class Triangulo extends Forma { /* ... */ }
```

Relembrando o exemplo da parte 5

```
public class Teste {
  private static void desenha(Forma[] fs) {
    for (int i = 0; i < fs.length; i++)
      fs[i].desenhar();
  }

  public static void main(String[] args) {
    Forma[] formas = new Forma[] {
      new Circulo(), new Forma(),
      new Quadrado(), new Triangulo()
    };
    desenha(formas);
  }
}
```

Relembrando o exemplo da parte 5

- Ampliação (*upcasting*) é a conversão implícita de uma subclasse para uma superclasse:

```
class Teste {
  public static void inverter(Forma f) {
     System.out.println("Inverte " + f);
  }

  public static void main(String[] args) {
     Circulo c = new Circulo();
     inverter(c);                    // Upcasting!
     Forma f = new Quadrado();       // Upcasting!
  }
}
```

Incrementando o exemplo

- O compilador realmente não sabe qual é o tipo. Veja um exemplo com geração aleatória:

```
public static void main(String[] args) {
  Forma f = null;
  switch((int)(Math.random() * 3)) {
     case 0: f = new Circulo();
     case 1: f = new Quadrado();
     case 2: f = new Triangulo();
     default: f = new Forma();
  }
  f.desenhar();
}
```

Esquecendo o tipo do objeto

- Quando realizamos ampliação, "esquecemos" o tipo de um objeto:

```
Forma f = new Quadrado();
```

- Não sabemos mais qual é a classe específica de `f`. Sabemos apenas que ele é uma forma;
- Por que fazer isso?

Métodos mais gerais

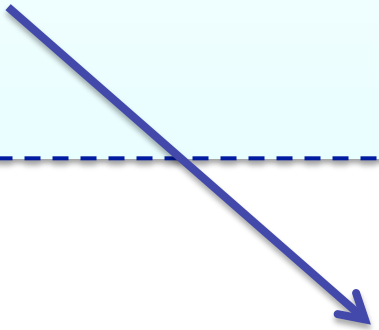
- Fazemos ampliação para escrevermos métodos mais gerais, para poupar tempo e esforço:

```
class Teste {
  public void inverter(Circulo f) {
     System.out.println("Inverte " + f);
  }
  public void inverter(Quadrado f) {
     System.out.println("Inverte " + f);
  }
  public void inverter(Triangulo f) {
     System.out.println("Inverte " + f);
  }
}
```

Amarração

- No entanto, se trabalhamos com Forma, como saber qual implementação executar quando chamamos um método?

```
public class Teste {
  private static void desenha(Forma[] fs) {
     for (int i = 0; i < fs.length; i++)
        fs[i].desenhar();
  }
}
```



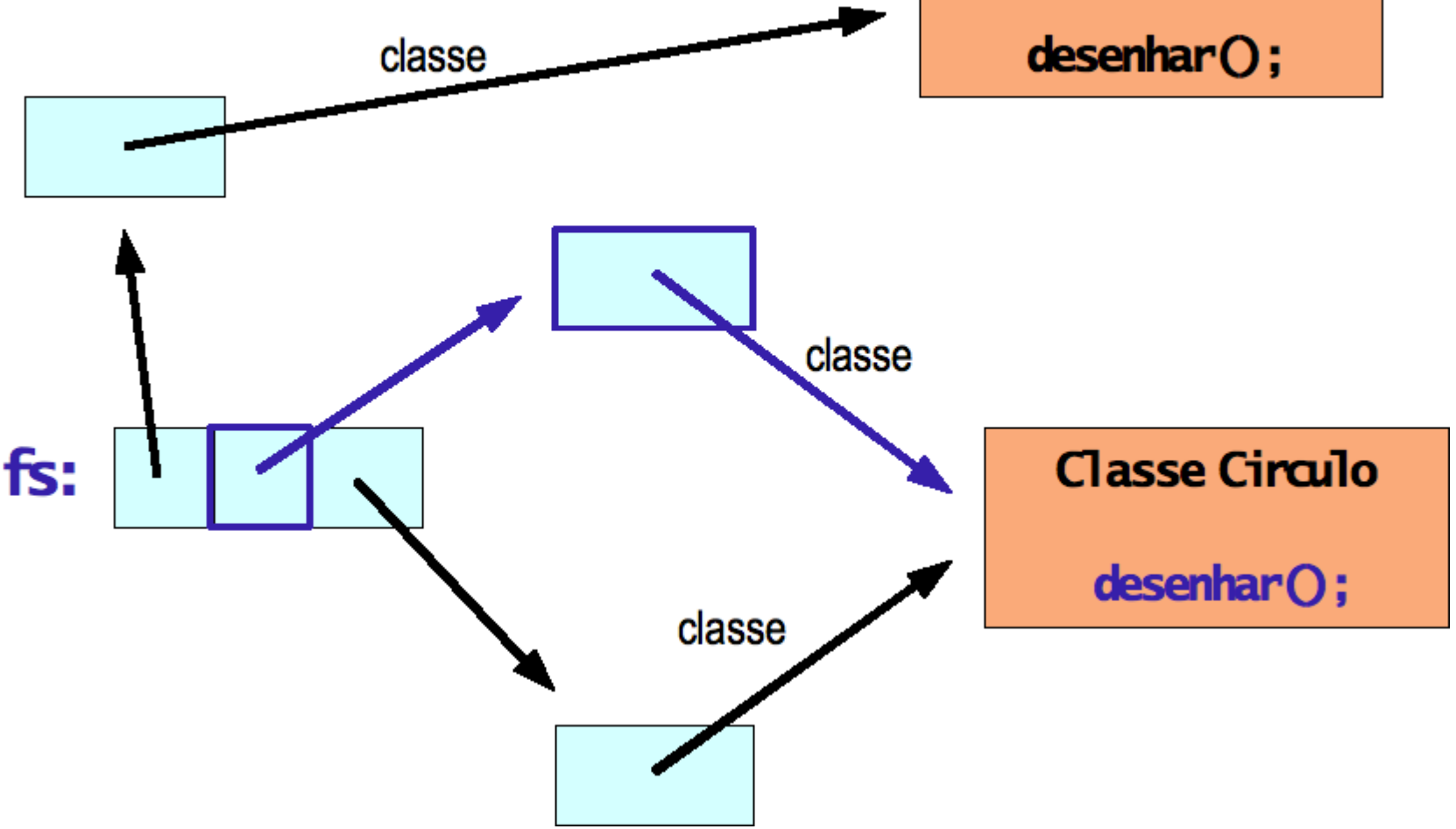
`fs[i]` é do tipo `Forma`.
Chamar sempre `Forma.desenhar()`?

Amarração tardia

- Em linguagens estruturadas, os compiladores realizam amarração em tempo de compilação;
- Em linguagens OO com polimorfismo, não temos como saber o tipo real do objeto em tempo de compilação;
- A amarração é feita em tempo de execução, também conhecida como:
  - Amarração tardia;
  - Amarração dinâmica; ou
  - Late binding.

Amarração tardia

```
fs[1].desenhar();
```



Quando usar

- Amarração dinâmica é menos eficiente;
- No entanto, ela que permite o polimorfismo;
- Java usa sempre amarração dinâmica;
- A exceção: se um método é final, Java usa amarração estática (pois ele não pode ser sobrescrito);
- Você não pode escolher quando usar um ou outro. É importante apenas entender o que acontece.

Benefícios do polimorfismo

- Extensibilidade:
  - Podemos adicionar novas classes sem alterar o método polimórfico.

```
class Retangulo extends Forma {
  public void desenhar() {
     System.out.println("Retangulo");
  }
}

class Quadrado extends Retangulo {
  public void desenhar() {
     System.out.println("Quadrado");
  }
}
```

Benefícios do polimorfismo

```
class Reta extends Forma {
  public void desenhar() {
     System.out.println("Reta");
  }
}

public class Teste {
  public void inverter(Forma f) {
     System.out.println("Inverte " + f);
  }
  public static void main(String[] args) {
     Forma f = new Reta();
     f.desenhar();
     inverter(f);
  }
}
```

Benefícios do polimorfismo

- A interface de todos é definida pela classe base;
- Novas classes possuem a mesma interface, portanto o sistema já sabe lidar com elas;
- Mesmo que todas as classes já existam de princípio, poupa-se tempo e esforço, codificando um método único para todas.

Sobrescrita vs. sobrecarga

- Cuidado para não confundir:

```
class Forma {
  public void aumentar(int t) {
     System.out.println("Forma.aumentar()");
  }
}

class Linha extends Forma {
  // Foi feita sobregarga e não sobrescrita!
  public void aumentar(double t) {
     System.out.println("Linha.aumentar()");
  }
}
```

Sobrescrita vs. sobrecarga

- A confusão gera um erro muito difícil de descobrir:

```
public class Teste {
  private static void aumentar(Forma f) {
     f.aumentar(10);
  }

  public static void main(String[] args) {
     Linha l = new Linha();
     aumentar(l); // Forma.aumentar()
  }
}
```

Classes e métodos abstratos

- Algumas vezes classes no topo da hierarquia são muito gerais:
  - O que é uma forma?
  - Como se desenha uma forma?
  - Como se aumenta uma forma?
- Tais operações não fazem sentido. Queremos apenas definir que elas existam, mas não implementá-las;
- A solução: métodos abstratos.

Classes e métodos abstratos

- Uma classe que possui métodos abstratos deve ser declarada como abstrata:

```
abstract class Forma {
  public abstract void desenhar();
}

class Circulo extends Forma {
  public void desenhar() {
     System.out.println("Círculo");
  }
}
```

Classes abstratas

- Não permitem criação de instâncias (objetos):
  - Um método abstrato não possui implementação, portanto não pode ser chamado;
  - Podemos ter classes abstratas sem métodos abstratos, mas não o contrário.
- Para ser útil, deve ser estendida:
  - Suas subclasses devem implementar o método ou declararem-se como abstratas.
- Servem para definir interfaces e prover algumas implementações comuns.

Classes e métodos abstratos

- Algumas regras:
  - Métodos estáticos não podem ser abstratos;
  - Construtores não podem ser abstratos;
  - Classes abstratas podem ter construtores (lembre-se que são chamados pelas subclasses!);
  - Métodos abstratos não podem ser privados.

Classes e métodos abstratos

```
// Classe abstrata pura.
abstract class Forma {
  public abstract void desenhar();
  public abstract void aumentar(int t);
}

// Classe abstrata.
abstract class Poligono extends Forma {
  private int lados;

  public Poligono(int lados) {
     this.lados = lados;
  }

  public int getLados() { return lados; }
  public abstract void pintar(int cor);
}
```

Classes e métodos abstratos

```
// Classe concreta.
class Retangulo extends Poligono {
  public Retangulo() {
     super(4);
  }
  public void desenhar() {
     System.out.println("Retangulo.desenhar");
  }
  public void aumentar(int t) {
     System.out.println("Retangulo.aumentar");
  }
  public void pintar(int cor) {
     System.out.println("Retangulo.pintar");
  }
}
```

Polimorfismo com `java.lang.Object`

- Relembrando a interface comum a todas as classes escritas em Java:
  - `clone()`: cria uma cópia do objeto (uso avançado);
  - `equals(Object o)`: verifica se objetos são iguais;
  - `finalize()`: chamado pelo GC (não é garantia);
  - `getClass()`: retorna a classe do objeto;
  - `hashCode()`: função hash;
  - `notify()`, `notifyAll()` e `wait()`: para uso com threads;
  - `toString()`: converte o objeto para uma representação como `String`.

O método equals()

- Compara dois objetos. Retorna `true` se forem iguais, `false` se não forem;
- Permitem polimorfismo em grande escala:
  - Podemos criar uma classe conjunto que armazena objetos de qualquer classe, desde que sejam objetos diferentes;
  - Podemos implementar um método que permite dizer se um objeto está no conjunto, se um conjunto está contido em outro, etc.

O método equals()

```
class Valor {
  int i;
  public Valor(int i) { this.i = i; }
}

public class Teste {
  public static void main(String[] args) {
     Integer m = new Integer(100);
     Integer n = new Integer(100);
     System.out.println(m == n);          // false
     System.out.println(m.equals(n));     // true
     Valor v = new Valor(100);
     Valor u = new Valor(100);
     System.out.println(v.equals(u));     // false
  }
}
```

O método equals()

```
class Valor {
  int i;
  public Valor(int i) { this.i = i; }
  public boolean equals(Object o) {
     return (o instanceof Valor)
               && (((Valor)o).i == i);
  }
}

public class Teste {
  public static void main(String[] args) {
     Valor v = new Valor(100);
     Valor u = new Valor(100);
     System.out.println(v.equals(u));   // true
  }
}
```

O método `toString()`

- Retorna uma representação em `String` do objeto em questão;
- Permite polimorfismo em grande escala:
  - Se quisermos imprimir um objeto de qualquer classe, ele será chamado;
  - Se quisermos concatenar um objeto de qualquer classe com uma `String`, ele será chamado.

O método toString()

```
class Valor {
  int i;
  public Valor(int i) { this.i = i; }
}

public class Teste {
  public static void main(String[] args) {
     Integer m = new Integer(100);
     System.out.println(m);  // 100

     Valor v = new Valor(100);
     System.out.println(v);  // Valor@82ba41
  }
}
```

O método toString()

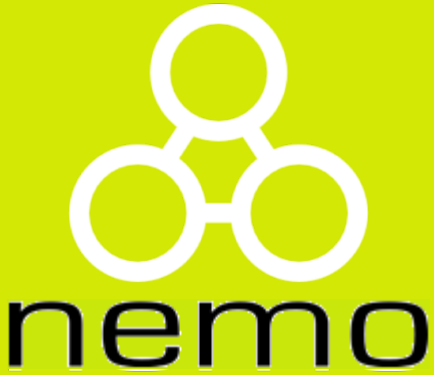
```
class Valor {
  int i;
  public Valor(int i) { this.i = i; }
  public String toString() { return "" + i;}
}

public class Teste {
  public static void main(String[] args) {
     Integer m = new Integer(100);
     System.out.println(m);  // 100

     Valor v = new Valor(100);
     System.out.println(v);  // 100
  }
}
```

Conclusões

- O polimorfismo permite a criação de métodos mais gerais, graças à amarração tardia;
- Hierarquias polimórficas são mais extensíveis;
- Classes e métodos abstratos auxiliam na criação de modelos mais precisos;
- A classe `Object`, ancestral de todas as classes Java, promove polimorfismo em grande escala com seus métodos.



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