Programação Orientada a Objetos em

Flávio Miguel Varejão

Departamento de Informática

UFES

Reuso de Classes

- Composição (ou Agregação)
  - Uso de Classe como Atributo de Nova Classe
  - Reuso como Cliente
- Herança
  - Extensão de Classe Existente
  - Reuso como Implementador

Composição

```
class FonteDagua {
  private String s;
  FonteDagua () {
    System.out.println("FonteDagua ()");
    s = new String("Construida");
  }
  public String toString() { return s; }
}

public class Engarrafador {
  private String garrafa;
  FonteDagua fonte;
  int i;
  float f;
```

Composição

```
  void imprime() {
    System.out.println("garrafa = " + garrafa);
    System.out.println("i = " + i);
    System.out.println("f = " + f);
    System.out.println("fonte = " + fonte);
  }
  public static void main(String[] args) {
      Engarrafador x = new Engarrafador ();
       x.imprime ();
  }
}
```

```
garrafa = null
i = 0
f = 0.0
fonte = null
```

Composição

- Inicialização das Referências
  - onde os objetos são definidos
    - sempre inicializados antes do construtor
  - no construtor
  - quando o objeto é necessário
    - mais eficiente porque objeto só é inicializado se for necessário

Composição

```
class Sabao {
  private String s;
  Sabao() {
    System.out.println("Sabao()");
    // Inicialização no construtor
    s = new String("Construido");
  }
  public String toString() { return s; }
}

public class Banho {
  private String
    // Inicialização no ponto da definicao:
    s1 = new String("Feliz"),
    s2 = "Feliz",
    s3, s4;
  Sabao omo;
  int i;
  float brinquedo;
```

Composição

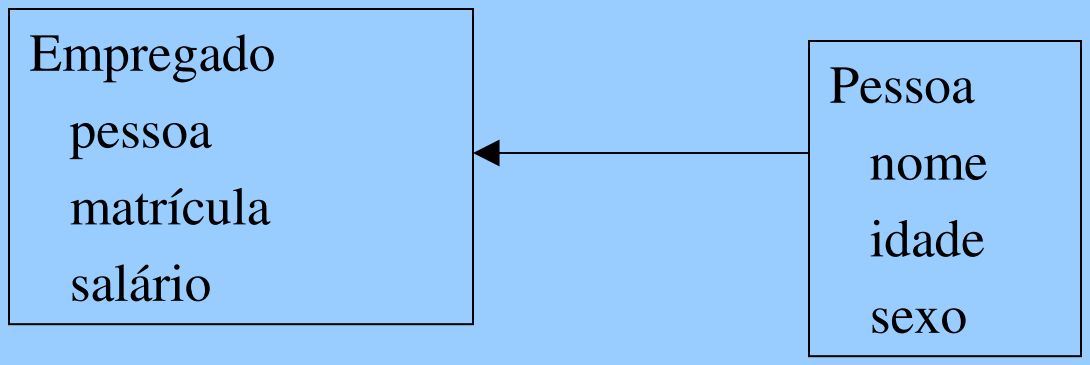
```
  Banho() {
    System.out.println("Dentro Banho()");
    s3 = new String("Maria");
    i = 47;
    brinquedo = 3.14f;
    omo = new Sabao();
  }
  void print() {
    // Inicializacao tardia:
    if(s4 == null) s4 = new String("Maria");
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
    System.out.println("s3 = " + s3);
    System.out.println("s4 = " + s4);
    System.out.println("i = " + i);
    System.out.println("brinquedo = " + brinquedo);
    System.out.println("omo = " + omo);
  }
  public static void main(String[] args) {
    Banho b = new Banho();
    b.print();
  }
}
```

Composição: Saída do Exemplo

```
Dentro Banho()
Sabao()
s1 = Feliz
s2 = Feliz
s3 = Maria
s4 = Maria
i = 47
brinquedo = 3.14
omo = Construido
```

Exercício

1 Criar classe Empregado que contenha os dados de uma Pessoa e mais dados de matrícula e salário



A Classe Empregado

```
public class Empregado {
   private Pessoa pessoa;
   private int matricula;
   private float salario;

   public Empregado (String n, int i, boolean sx,
                     int m, float s) {
      pessoa = new Pessoa (n, i, sx);
      matricula = m;
      salario = s;
   }
   public Empregado (Empregado e) {
      pessoa = new Pessoa (e.pessoa);
      matricula = e.matricula;
      salario = e.salario;
   }
```

A Classe Empregado

```
public String retornaNome() {
   return pessoa.retornaNome();
}
public int retornaIdade() {
   return pessoa.retornaIdade();
}
public boolean retornaSexo() {
   return pessoa.retornaSexo();
}
public int retornaMatricula() {
   return matricula;
}
public float retornaSalario() {
   return salario;
}
```

A Classe Empregado

```
    public String toString() {
       return pessoa + "\nMatricula: " +
              matricula + "\nSalario: " +
              salario;
    }
    public void imprimeConsole() {
       pessoa.imprimeConsole();
       System.out.println("Matricula: " +
              matricula + "\nSalario: " +
              salario);
    }
    public boolean menorMatricula(Empregado e) {
      return matricula < e.matricula;
    }
}
```

Regularidades neste Tipo de Composição

- Classe nova contém um Atributo de outra Classe

  Empregado contém Pessoa
- Mesmos Métodos

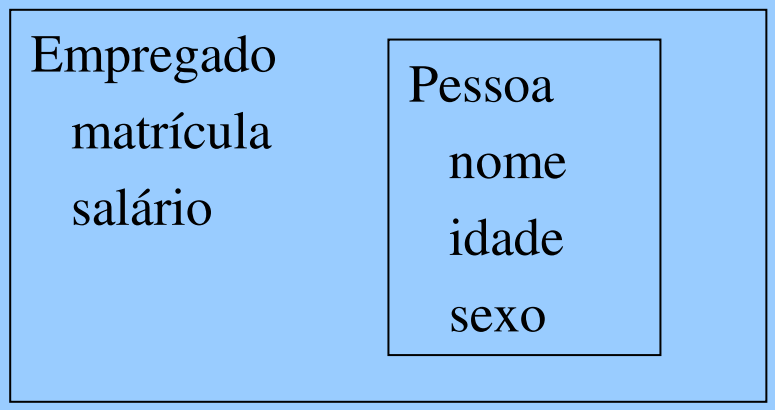
  retornaNome, retornSexo e retornaIdade
- Métodos Semelhantes

  Construtores, toString, imprimeConsole
- Métodos Adicionais

  retornaSalario, retornaMatricula, menorMatricula

Herança

- Composição implícita de objeto de outra classe
- Métodos idênticos são disponibilizados automaticamente
  - chamada dos métodos da outra classe é implícita
- Métodos semelhantes podem ser reescritos
- Métodos novos podem ser adicionados



Herança

- Todas classes herdam de Object
- Herança Simples
  - classes só herdam diretamente de uma outra classe
- Uso de cláusula *extends* para declarar uma classe como herdeira de outra
- Ao herdar, classe derivada adquire atributos e métodos da classe base
- Regra geral
  - atributos da classe base privados ou protegidos
  - métodos públicos

Herança

```
// Detergente.java
// Propriedades da Heranca

class Limpador {
  private String s = new String("Limpador");
  public void anexar(String a) { s += a; }
  public void diluir() { anexar(" diluir()"); }
  public void aplicar() { anexar(" aplicar()"); }
  public void esfregar() { anexar(" esfregar()"); }
  public void imprimir() { System.out.println(s); }
  public static void main(String[] args) {
    Limpador x = new Limpador();
    x.diluir(); x.aplicar(); x.esfregar();
    x.imprimir();
  }
}
```

Herança

```
public class Detergente extends Limpador {
  // Muda um método:
  public void esfregar() {
    anexar(" Detergente.esfregar()");
    super.esfregar(); // Chama a versao da classe base
  }
  // Adiciona métodos a interface:
  public void espumar() { anexar(" espumar()"); }
  // Testa a nova classe:
  public static void main(String[] args) {
    Detergente x = new Detergente();
    x.diluir();
    x.aplicar();
    x.esfregar();
    x.espumar();
    x.imprimir();
    System.out.println("Testando classe base:");
    Limpador.main(args);
  }
}
```

Herança

- Limpador e Detergente possuem main ()
  – código de teste integra a classe

  java Detergente

  java Limpador
- Detergente herda métodos de Limpador
- Inclusão de espumar ()

Herança e Sobrescrita

- Sobrescrita de esfregar ( )
  - redefine método na classe derivada
- Pode definir atributo com mesmo nome da classe base
  - pode ser de mesmo tipo ou tipo diferente
- Uso de *super*
  - para chamar método ou atributo da classe base

Herança e Inicialização

```
class Arte {
  Arte() {
    System.out.println("Construtor Arte");
  }
}
class Desenho extends Arte {
  Desenho() {
    System.out.println("Construtor Desenho");
  }
}
public class Cartao extends Desenho {
  Cartao() {
    System.out.println("Construtor Cartao");
  }
  public static void main(String[] args) {
    Cartao x = new Cartao();
  }
}
```

Construtores com Argumentos

```
class Jogo {
  Jogo(int i) {
    System.out.println("Construtor Jogo");
  }
}
class JogoTabuleiro extends Jogo{
  JogoTabuleiro(int i) {
    super(i);
    System.out.println("Construtor JogoTabuleiro");
  }
}
public class Xadrez extends JogoTabuleiro {
  Xadrez() {
    super(11);
    System.out.println("Construtor Xadrez");
  }
  public static void main(String[] args) {
    Xadrez x = new Xadrez();
  }
}
```

Composição + Herança

```
class Prato
  Prato (int i) {
    System.out.println("Construtor Prato");
  }
}
class PratoJantar extends Prato {
  PratoJantar(int i) {
    super(i);
    System.out.println(
      "Construtor PratoJantar");
  }
}
class Utensilio {
  Utensilio(int i) {
    System.out.println("Construtor Utensilio");
  }
}
```

Composição + Herança

```
class Colher extends Utensilio {
  Colher(int i) {
    super(i);
    System.out.println("Construtor Colher");
  }
}
class Garfo extends Utensilio {
  Garfo(int i) {
    super(i);
    System.out.println("Construtor Garfo");
  }
}
class Faca extends Utensilio {
  Faca(int i) {
    super(i);
    System.out.println("Construtor Faca");
  }
}
class Costume{
  Costume(int i) {
    System.out.println("Construtor Costume");
  }
}
```

Composição + Herança

```
public class PreparacaoMesa extends Costume {
  Sopa sp;
  Garfo frk;
  Faca kn;
  PratoJantar pl;
  PreparacaoMesa(int i) {
    super(i + 1);
    sp = new Colher(i + 2);
    frk = new Garfo(i + 3);
    kn = new Faca(i + 4);
    pl = new PratoJantar(i + 5);
    System.out.println(
      "Construtor PreparacaoMesa");
  }
  public static void main(String[] args) {
    PreparacaoMesa x = new PreparacaoMesa(9);
  }
}
```

Restauração

- Java não possui destrutores
  - coletor de lixo desaloca memória
- Algumas vezes é necessário executar ações para restaurar estado anterior ou limpeza
  - não pode confiar no coletor
  - necessário criar método próprio de restauração na classe e chamá-lo explicitamente
  - exemplo do contador de objetos
- Regra geral é
  - chamar método de restauração na ordem inversa da criação

Restauração

```
class Forma {
  Forma(int i) {
    System.out.println("Construtor Forma");
  }
  void limpeza() {
    System.out.println("Limpeza Forma");
  }
}
class Circulo extends Forma {
  Circulo(int i) {
    super(i);
    System.out.println("Desenhando Circulo");
  }
  void limpeza() {
    System.out.println("Apagando um Circulo");
    super.limpeza();
  }
}
```

Restauração

```
class Triangulo extends Forma {
  Triangulo(int i) {
    super(i);
    System.out.println("Desenhando um Triangulo");
  }
  void limpeza() {
    System.out.println("Apagando um Triangulo");
    super.limpeza();
  }
}
class Linha extends Forma {
  private int comeco, fim;
  Linha(int comeco, int fim) {
    super(comeco);
    this.comeco = comeco;
    this.fim = fim;
    System.out.println("Desenhando uma Linha: " +
            comeco + ", " + fim);
  }
```

Restauração

```
  void limpeza() {
    System.out.println("Apagando uma Linha: " +
           comeco + ", " + fim);
    super.limpeza();
  }
}
public class SistemaCAD extends Forma {
  private Circulo c;
  private Triangulo t;
  private Linha[] linhas = new Linha[10];
  SistemaCAD (int i) {
    super(i + 1);
    for(int j = 0; j < 10; j++)
      linhas[j] = new Linha(j, j*j);
    c = new Circulo(1);
    t = new Triangulo(1);
    System.out.println("Construtor Combinado");
  }
```

Restauração

```
  void limpeza() {
    System.out.println("SistemaCAD.limpeza()");
    // A ordem de remocao e o reverso
    // da ordem de inicializacao
    t.limpeza();
    c.limpeza();
    for(int i = linhas.length - 1; i >= 0; i--)
      linhas[i].limpeza();
    super.limpeza();
  }
  public static void main(String[] args) {
    SistemaCAD x = new SistemaCAD (47);
    // …
    x.limpeza();
  }
}
```

Composição x Herança

- Ambos permitem colocar objetos dentro de uma nova classe
- Composição
  - quer as características de uma classe existente, mas não sua interface
- Herança
  - quer as características e a interface
- Regra Geral
  - tem-um -> composição
  - é-um -> herança

Classe Empregado com Herança

```
public class Empregado extends Pessoa {
   private int matricula;
   private float salario;

   public Empregado (String n, int i, boolean sx,
                     int m, float s) {
      super(n, i, sx);
      matricula = m;
      salario = s;
   }
   public Empregado (Empregado e) {
      super(e.retornaNome(), e.retornaIdade(),
            e.retornaSexo()); // private de Pessoa
      matricula = e.matricula;
      salario = e.salario;
   }
```

Classe Empregado com Herança

```
public int retornaMatricula() {
   return matricula;
}
public float retornaSalario() {
   return salario;
}
public boolean menorMatricula(Empregado e) {
  return matricula < e.matricula;
}
public boolean menorSalario(Empregado e) {
  return salario < e.salario;
}
public String toString() {
  return super.toString() + "\nMatricula: " +
     matricula + "\nSalario: " + salario;
}
```

Classe Empregado com Herança

```
    public void imprimeConsole() {
      super.imprimeConsole();
      System.out.println("Matricula: " +
         matricula + "\nSalario: " + salario);
    }
}
```

Cuidados com Herança

- Em geral, não deve ser feita inicialização de atributos da classe base no construtor da classe derivada
- Ausência de construtor padrão na classe base
- Sobrescrita X Novo Método
  - xpto (Pessoa p)
  - xpto (Empregado e)

Atributos Públicos na Composição

- Reduz trabalho do implementador da classe
- Pode ser mais eficiente
- sacrifica reuso de código usuário
- Em geral deve usar *private*

Atributos Públicos na Composição

```
class Motor {
  public void ligar() {}
  public void parar() {}
}
class Pneu {
  public void encher (int psi) {}
}
class Janela {
  public void fechar() {}
  public void abrir() {}
}
class Porta {
  public Janela janela = new Janela();
  public void abrir() {}
  public void fechar() {}
}
```

Atributos Públicos na Composição

```
public class Automovel {
  public Motor motor = new Motor();
  public Pneu[] pneus = new Pneu[4];
  public Porta esquerda = new Porta(),
       direita = new Porta();           // 2-portas
  public Automovel() {
    for(int i = 0; i < 4; i++)
      pneus[i] = new Pneu();
  }
  public static void main(String[] args) {
    Automovel carro = new Automovel();
    carro.esquerda.janela.fechar();
    carro.pneus[0].encher(72);
  }
}
```

Membros Protegidos

```
class Aloprada {
  private int i;
  protected int ler() { return i; }
  protected void ajustar (int ii) { i = ii; }
  public Aloprada (int ii) { i = ii; }
  public int valor (int m) { return m*i; }
}
public class Maluca extends Aloprada {
  private int j;
  public Maluca (int jj) { super(jj); j = jj; }
  public void mudar(int x) { ajustar(x); }
}
```

Vantagens da Herança

- Reuso
- Desenvolvimento Incremental
  - Erros ficam concentrados no novo código
- Polimorfismo
  - Objeto da classe derivada pode ser usado onde se espera objeto da classe base

Relacionamento É-UM

```
class Instrumento{
  public void tocar() {}
  static void afinar(Instrumento i) {
    // ...
    i.tocar();
  }
}
public class Sopro extends Instrumento{
  public static void main(String[] args) {
    Sopro flauta = new Sopro();
    Instrumento.afinar(flauta); // Upcasting
  }
}
```

Dados Finais

```
class Valor {
  int i = 1;
}
public class DadoFinal {
  // Podem ser constantes em tempo de compilacao
  final int i1 = 9;
  static final int CONST_2 = 99;
  // Constante publica comum:
  public static final int CONST_3 = 39;
  // Não podem ser constantes em tempo de compilacao:
  final int i4 = (int)(Math.random()*20);
  static final int i5 = (int)(Math.random()*20);
  Valor v1 = new Valor();
  final Valor v2 = new Valor();
  static final Valor v3 = new Valor();
  // Arrays:
  final int[] a = { 1, 2, 3, 4, 5, 6 };
```

Dados Finais

```
  public void print(String id) {
    System.out.println(
      id + ": " + "i4 = " + i4 +
      ", i5 = " + i5);
  }
  public static void main(String[] args) {
    DadoFinal fd1 = new DadoFinal();
    //! fd1.i1++; // Erro: não pdoe mudar o valor
    fd1.v2.i++; // Objeto não e constante!
    fd1.v1 = new Valor(); // OK -- não final
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // Objeto não e constante!
    //! fd1.v2 = new Valor(); // Erro: Não
    //! fd1.v3 = new Valor(); // pode mudar referencia
    //! fd1.a = new int[3];
    fd1.print("fd1");
    System.out.println("Criando novo DadoFinal");
    DadoFinal fd2 = new DadoFinal();
    fd1.print("fd1");
    fd2.print("fd2");
  }
}
```

Dados Finais Não Inicializados

```
class Viagem { }
class DadoFinalLivre {
  final int i = 0; // Final inicializado
  final int j; // Final nao inicializado
  final Viagem p; // Referencia final nao inicializada
  DadoFinalLivre () {
    j = 1; // Inicializa Final
    p = new Viagem();
  }
  DadoFinalLivre (int x) {
    j = x; // Inicializa Final
    p = new Viagem();
  }
  public static void main(String[] args) {
    DadoFinalLivre bf = new DadoFinalLivre();
  }
}
```

Argumentos Finais

```
class Argumento {
  public void naoFazNada() {}
}
public class ArgumentosFinais {
  void com(final Argumento g) {
    //! g = new Argumento(); // Ilegal  -- g e final
  }
  void sem(Argumento g) {
    g = new Argumento(); // OK -- g não final
    g.naoFazNada();
  }
  // void f(final int i) { i++; } // Não pode mudar
  // Você pode somente ler dos dados primitivos:
  int g(final int i) { return i + 1; }
  public static void main(String[] args) {
    ArgumentosFinais bf = new ArgumentosFinais();
    bf.sem(null);
    bf.com(null);
  }
}
```

Métodos Finais

- Prevenir mudança na herança
- Eficiência através de inline
- Métodos *private* são implicitamente *final*
  - não há sobrescrita - cria novo método
  - *final* não acrescenta nada a declaração *private*

Métodos Finais

```
class ComFinais {
  private final void f() {
    System.out.println("ComFinais.f()");
  }
  // Mesma coisa
  private void g() {
    System.out.println("ComFinais.g()");
  }
}
class SobrePrivado1 extends ComFinais {
  private final void f() {
    System.out.println("SobrePrivado1.f()");
  }
  private void g() {
    System.out.println("SobrePrivado1.g()");
  }
}
```

Métodos Finais

```
class SobrePrivado2 extends SobrePrivado1 {
  public final void f() {
    System.out.println("SobrePrivado2.f()");
  }
  public void g() {
    System.out.println("OverridingPrivate2.g()");
  }
}
public class IlusaoFinal {
  public static void main(String[] args) {
    SobrePrivado2 op2 = new SobrePrivado2();
    op2.f();
    op2.g();
    // Você pode fazer "upcast"
    SobrePrivado1 op1 = op2;
    // Mas você não pode chamar os métodos
    //! op1.f();
    //! op1.g();
    // O mesmo aqui
    ComFinais wf = op2;
    //! wf.f();
    //! wf.g();
  }
}
```

Classes Finais

```
class CerebroPequeno {}
final class Dinossauro {
  int i = 7;
  int j = 1;
  CerebroPequeno x = new CerebroPequeno();
  void f() {}
}
//! class Tiranossauro extends Dinossauro {}
public class Jurassico {
  public static void main(String[] args) {
    Dinossauro n = new Dinossauro();
    n.f();
    n.i = 40;
    n.j++;
  }
}
```

Inicialização com Herança

```
class Inseto {
  int i = 9;
  int j;
  Inseto() {
    prt("i = " + i + ", j = " + j);
    j = 39;
  }
  static int x1 = prt("static Inseto.x1 inicializado");
  static int prt(String s) {
    System.out.println(s);
    return 47;
  }
}
```

Inicialização com Herança

```
public class Besouro extends Inseto {
  int k = prt("Besouro.k inicializado");
  Besouro() {
    prt("k = " + k);
    prt("j = " + j);
  }
  static int x2 = prt("static Besouro.x2
inicializado");
  public static void main(String[] args) {
    prt("Besouro construtor");
    Besouro b = new Besouro();
  }
}
```

Inicialização com Herança
Saída

```
static Inseto.x1 inicializado
static  Besouro.x2 inicializado
Besouro construtor
i = 9, j = 0
Besouro.k inicializado
k = 47
j = 39
```