Universidade de São Paulo
Brasil

Aula 06 – (Re)utilização de Classes
Composição, Herança, e Classes Abstratas

MAC0321 - Laboratório de Programação Orientada a Objetos

Professor: Marcelo Finger **(mfinger@ime.usp.br)**

Departamento de Ciência da Computação
Instituto de Matemática e Estatística

Tópicos

1. Composição
2. Herança
3. Polimorfismo e sobrecarga
4. Classes abstratas vs Interfaces
5. Classes dentro de classes

Composição

Classe Pessoa

```
package br.usp.ime.mac321.aula06;
public class Pessoa {
    protected String nome;
    protected double peso;
    protected double altura;

    public Pessoa(String nome) {
        this.nome = nome;
    }

    public void imprime ( ) {
        System.out.println("Pessoa: "  + nome + ", peso: " + peso +
                           ", altura: " + altura);
    }
}
```

Formas de Reutilização de Classes

- Composição:
  - Criar objetos de uma classe existente na nova classe
  - Na classe Pessoa um dos membros é String
  - Reutiliza classes existentes

  Pergunta: o novo objeto tem objeto(s) da classe existente?
- Herança
  - Criar nova classe refinando o tipo de uma existente herdando os membros e os métodos da classe
  - Aproveitar a interface de uma classe existente

  Pergunta: o novo objeto é um objeto da classe existente ?

Herança

Herança em Java

Em Java, use palavra-chave extends

```
public class Cliente extends Pessoa {
     // além dos membros de Pessoa:
     private int conta;
     private String cpf;
     private double saldo;

     private static int próximaConta = 1000;

     // também colocamos novos métodos
     public void deposito (double val) { saldo += val;}
     public boolean saque (double val) {
          if (saldo-val>=0)
               { saldo -= val; return true;}
          return false;
     }
...
```

Atenção com a Herança !!

Perguntas para saber se a herança está sendo bem utilizada:

- Membros e métodos foram adicionados?
- Estes membros e métodos poderiam pertencer a classe base?
- A classe derivada é como a classe base?

Apesar da herança ser a estrela de POO, nem sempre ela é bem usada!!!

Observe as respostas às perguntas

- Membros e métodos foram adicionados?

  Sim
- Estes membros e métodos poderiam pertencer a classe base?

  Não, nem toda a pessoa tem saldo...
- A classe derivada é como a classe base?

  Não, um cliente é uma pessoa, mas tem outras características.

Construtores e Herança

O construtor da classe derivada ***usa*** o da classe base

1ª Opção: Se existe um construtor default, sem parâmetros este é chamado automaticamente

2ª Opção: Se não há construtor padrão, ou há necessidade de algo especial, o construtor desejado pode ser chamado

Tem que ser o primeiro comando do construtor, e se utiliza a palavra reservada super

Construtor para a classe Cliente

```
public Cliente(String nome, String cpf, int saldo) {
    super(nome); // tem que ser o primeiro comando;
    this.cpf = cpf;
    conta = ++próximaConta;
    this.saldo = saldo;
}
public Cliente(String nome, String cpf) {
    this(nome, cpf, 0);
}
```

Mudança nos métodos

O método imprime() de Pessoa não é adequado a um Cliente
Para resolver isto, redefinimos o método imprime()

```
@Override
public void imprime ( ) {
    System.out.println(nome + ", conta "+conta+", saldo: "+saldo+
            ", cpf: "+cpf);
    }
// se o objeto for Pessoa imprime peso, altura, etc
// se o objeto for Cliente imprime saldo, cpf, etc
public void imprimeSuper() {super.imprime();}
```

Palavras chave - protected

- Conforme visto anteriormente existem quatro qualificadores
  - private, “friendly”, protected e public
  - A palavra chave protected denota:
    - acessibilidade total dentro do mesmo package (friendly)
    - permite o acesso à classe derivada

Palavras chave - final

Pode ser usada para dados, métodos e classes

- Dados
  - indica constantes que nunca mudam
  - indica variáveis que não podem ser alteradas
  - indica argumentos que não podem ser alterados
- Métodos
  - métodos que não podem ser modificados pelos herdeiros
  - indicação para o compilador (inlining code)
- Classes
  - indica classes que não podem ter herdeiros
    - não use a não ser que você tenha MUITA certeza

Uso do static + final

- Pode ser usada para dados, métodos
  - Dados
    - constantes
    - variáveis
  - Métodos: de classe que não podem ser alterados por herdeiros

Polimorfismo

Polimorfismo

- Principais características de POO
  - encapsulamento
  - herança
  - polimorfismo
    - Separa a interface da implementação
    - Descoberta dinâmica de qual método deve ser usado
    - Funciona pois todos os métodos da classe base estão presentes nas derivadas!!

Exemplo de Polimorfismo

```
class Animal {
  void nasce() {
    println("Nasceu 1 animal");
  }
  void cresce() {
    println("Cresceu 1 animal");
  }
}
class Mamifero extends Animal {
  void nasce()  {
    println("Nasceu 1 mamifero");
  }
  void cresce() {
    println("Cresceu 1 mamifero");
  }
}
```

```
public class Homem extends Mamifero  {
  void nasce()  {
    println("Nasceu um homem");
  }
  void cresce()  {
    println("Cresceu um homem");
  }
  public static void main(String []argc) {
    Animal x = new Animal();
    x.nasce();
    Homem h = new Homem();
    h.nasce();
    x = h;       // OK, pois Homem deriva de Animal
    x.nasce(); // mas o que faz???
  }
}
```

Exemplo de polimorfismo

```
// pode se criar a seguinte classe
class Vida  {
  public static void ciclo(Animal qualquer) {
        qualquer.nasce();
        for(int i=0;i<5;i++)
              qualquer.cresce();
  }
}

// e as seguintes chamadas no metodo main()
Vida.ciclo(new Animal());
Vida.ciclo(new Mamifero());
Vida.ciclo(new Homem());
```

A decisão sobre qual método de nome nasce/cresce deve ser chamado só é feita durante a execução!! Quem manda é o new

O método ciclo funciona mesmo para classes derivadas que vierem a ser criadas!!

Confusão comum

- Overloading (sobrecarga) *x* Overriding(sobrepujar)
  - Sobrecarga - quando a diferença entre diversos métodos se dá através de mudança nos parâmetros
  - Sobrepujar o método - quando métodos da classe base são alterados nas classes derivadas.

Classes Abstratas

Classes abstratas

- Quando tivermos os animais (répteis, peixes, etc) não serão mais necessários objetos Animal
  - Animal poderia ser uma classe abstrata
    - Não se pode criar objetos desta classe (nada de **new**)
    - Não é necessário definir os métodos (apenas protótipos)
    - Pode se derivar classes desta
      - os métodos da classe abstrata DEVEM ser definidos na derivada
    - Pode-se definir métodos completamente em uma classe abstrata

Exemplo de classe abstrata

```
abstract class Veiculo  {
        private double preco;
        public void setPreço(double valor) {
                preco = valor;
        }
        public double getPreço() {
                return preco;
        }
        abstract public void move();
        abstract public void tamanho();
}
```

Exemplo de classe abstrata

```
class Carro extends Veiculo {
      public void move() {System.out.println("Por vias pavimentadas");}
      public void tamanho() {System.out.println("Entre 2 e 4 metros");}
}

class Aviao extends Veiculo  {
      public void move() {System.out.println("Pelo ar");}
      public void tamanho() {System.out.println("Entre 4 e 200 metros");}
}
class TecoTeco extends Aviao  {
      public void move() {System.out.println("Pelo ar, mas baixo");}
      public void tamanho() {System.out.println("Entre 4 e 6 metros");}
}
```

Exemplo de classe abstrata

```
public class TestaVeiculos {
    public static void main(String [] argc) {
        Veiculo[] v = new Veiculo[5]; //vetor com 5 veiculos
        v[0]=new Carro();
        v[1]=new Aviao();
        v[2]=new TecoTeco();
        v[3]=new Aviao();
        v[4]=new TecoTeco();
        for(int i=0;i<5;i++) {
            System.out.print("O veiculo "+i+" se move: ");
            v[i].move();
        }
    }
}
```

Herança e o método finalize()

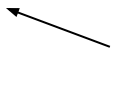
- O método finalize desaloca recursos quando o coletor de lixo libera o espaço de um objeto
- Quando a classe base tem o método finalize
  - Este deve ser protected (acesso à classe derivada)
  - Este deve ser chamado na classe derivada
  - De preferência, finalizar os objetos na mesma ordem de criação

Downcasting

Usando as classes Veiculo e Carro:

```
class CarroCorrida extends Carro {
        public void move() {System.out.println("Por pistas de corrida");}
        public void tamanho() {System.out.println("Entre 2 e 4 metros");}
        public void seguranca() {System.out.println(“cinto 5 pontos”);}
}

// e se no programa temos:
Veiculo x = new CarroCorrida();
// como chamar o metodo seguranca ???
// downcasting, da classe base a classes derivadas
((CarroCorrida) x).seguranca(); // OK, mas pode dar erro de runtime se malfeito
```



downcasting

Interfaces x Classes Abstratas

- Interfaces são como classes abstratas
  - mas oferecem mais flexibilidade !
  - Interfaces são classes abstratas puras
    - só contém métodos abstratos
      - todos automaticamente públicos
  - Pode conter membros
    - mas são todos static e final
- interface fornece apenas a forma, e não a implementação

Exemplo de uso

```
interface TudoQueSeMove {
  double VELOCIDADEMAX = 300000; // todo membro é static e final
  void move();                                  // todo membro e método é public
  void mudaVelocidade(double vel);
}
class Veiculo implements TudoQueSeMove {
// agora obrigatoriamente tenho que definir os métodos (públicos !)
private double velocidade=0;
public void move(){
  System.out.println("Vai de um lugar a outro");
}
public void mudaVelocidade(double vel) {
  if (vel<=VELOCIDADEMAX) // o membro da interface está disponível
        velocidade=vel;
}
```

Afinal qual a diferença entre

A interface:

```
interface TudoQueSeMove {
  double VELOCIDADEMAX = 300000;
  void move();
  void mudaVelocidade(double vel);
}
```

A classe abstrata:

```
abstract class TudoQueSeMove {
  static final double VELOCIDADEMAX = 300000;
  abstract public void move();
  abstract public void mudaVelocidade(double vel);
}
```

Interface vs. Classe Abstrata

Uma classe derivada só pode ter 1 classe base

- Mas pode implementar várias interfaces !
- Desta forma, um objeto x é um a, b e c.

```
// Tipoa é uma classe, Tipob e Tipoc são interfaces
Class Tipox extends Tipoa implements Tipob, Tipoc {...}
Tipox x = new Tipox();
Tipoa a; Tipob b; Tipoc c;
a = x; b=x; c=x; // OK pois objetos do Tipox são também do Tipoa, Tibob e Tipoc
```

Interfaces públicas podem agrupar constantes

Herança múltipla: exemplo

```
interface PodeVoar {
  void voa();
}
interface TransportaPassageiros {
  double precoPorPessoa();
}
class Aviao extends Veiculo implements PodeVoar, TransportaPassageiros {
  public void voa() {
        …
  }
  public double precoPorPessoa () {
        …
  }
```

Herança múltipla: exemplo

```
public class Compania {
  static void voaParaAlgumLugar(PodeVoar a) {
        a.voa();
  }
  static void levaGente(TransportaPassageiros b) {
        b.precoPorPessoa();
  }
  static void revisaVeiculo(Veiculo v) {
        // executa métodos relativos à veículos
  }
  public static void main(String [] args) {
        Aviao aviao1 = new Aviao();
        voaParaAlgumLugar(aviao1);
        levaGente(aviao1);
        revisaVeiculo(aviao1);
  }
}
```

Agrupando constantes

```
public interface Months {
  int    JANUARY = 1, FEBRUARY = 2, MARCH = 3,
    APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
    AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
    NOVEMBER = 11, DECEMBER = 12;
}
```

Classes Internas

Classes dentro de classes

- *Inner class* - definição de uma classe dentro de outra
  - geralmente a classe exterior possui um método que retorna uma referência para a classe interior
- São geralmente usadas para implementar uma interface dentro de uma classe de forma totalmente transparente
  - com uma referência para a classe base ou interface, pode-se esconder o tipo exato da classe derivada (inner classes podem ser private)

Classes dentro de classes

- Principais características
  - os objetos criados dentro da classe podem ser referenciados fora (desde que sejam derivados)
  - os objetos interiores tem livre acesso aos membros do objeto que estão na classe interna.

Exemplo de classe Interna

```
public class Qualquer {
 // métodos
 void f(){
      Testador t = new Testador();
      t.teste();
 }
 class Testador{
  public void teste(){
     for(int i = 0; i < 10; i++)
         System.out.println("Funciona");
      }
  }
}
```

Lista de exercícios

No computador com o Eclipse

Entrega até o final do dia

MAC321
Lab POO

- Professor: Marcelo Finger

  E-mail: mfinger@ime.usp.br