Aula 6

Professores:

Carlos Bazílio
Isabel Rosseti

Polimorfismo, Conversão de Tipo e Amarração Tardia

Conteúdo:

- revisão da aula anterior
- motivação
- polimorfismo
- conversão de tipo
- amarração tardia
- exercício

Fundação CECIERJ
Consórcio cederj

Revisão da aula anterior

- herança
  - **extends**

```
class Point {
  int x, y;
  Point(int x0, int y0) {
    x = x0;
    y = y0;
  }
  void move(int dx, int dy){
    x += dx;
    y += dy;
  }
}
class Pixel extends Point {
  int color;
  Pixel(int x, int y, int c) {
      super(x, y);
      color = c;
    }
  }
```

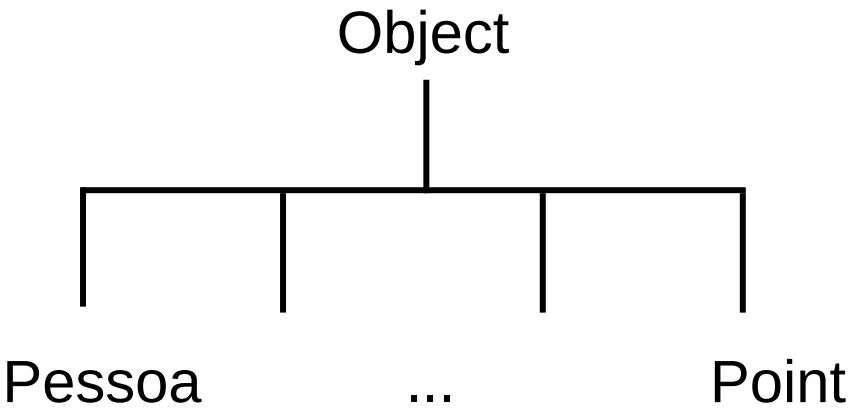
Revisão da aula anterior

- herança
  - **extends**
  - **super**

```
class Pixel extends Point {
   int color;
   Pixel(int x, int y, int c) {
     super(x, y);
     color = c;
   }
 }
```

Revisão da aula anterior

- herança
  - **extends**
  - **super**
- Object



Revisão da aula anterior

- herança
  - **extends**
  - **super**
- Object
- Sobrecarga
  - construtores
  - métodos

```
class Point {
  int x = 0;
  int y = 0;
  Point() {
  }
  Point(int x0, int y0) {
    x = x0;
    y = y0;
  }
  void move(int dx, int dy){
    x += dx;
    y += dy;
  }
}
```

Motivação

- como é possível criar um vetor capaz de guardar quaisquer objetos?
- como é possível recuperar corretamente cada elemento deste vetor?
- solução: polimorfismo

Polimorfismo

- polimorfismo é a capacidade de um objeto adquirir diversas formas
- esta capacidade decorre do mecanismo de herança
- porque, ao estendermos uma classe, **não** perdemos compatibilidade com a superclasse

Polimorfismo de **Pixel**

- a sub-classe de **Point**, **Pixel**, é compatível com **Point**
- ou seja, um **Pixel** *é* um ponto

```
class Point {
  int x, y;
  Point(int x0, int y0) {
    x = x0;
    y = y0;
  }
  void move(int dx, int dy){
    x += dx;
    y += dy;
  }
}
class Pixel extends Point {
  int color;
  Pixel(int x, int y, int c) {
      super(x, y);
      color = c;
    }
  }
```

Exemplo de polimorfismo

- sempre que precisarmos de um **Point**, podemos usar um **Pixel** em seu lugar
- podemos querer criar um vetor de pontos
- este vetor poderá conter pixels:

```
Point[] pontos = new Point[5]; // um vetor de pontos

pontos[0] = new Point();
pontos[1] = new Pixel(1,2,0); // um pixel é um ponto
```

Mais sobre polimorfismo

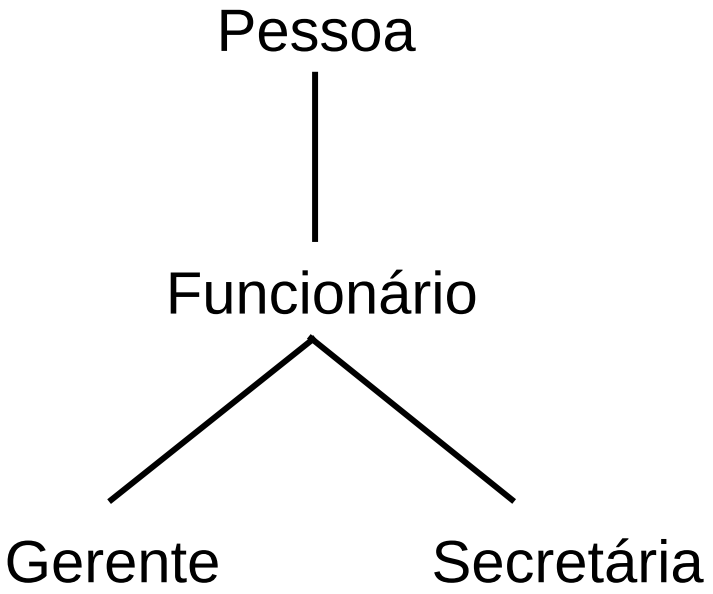
- um pixel pode ser usado sempre que se necessita um ponto

- porém, o contrário não é verdade: não podemos usar um ponto quando precisamos de um pixel

```
Point pt = new Pixel(0,0,1); // OK! pixel é ponto
Pixel px = new Point(0,0); // ERRO! ponto não é pixel
```

Conclusão sobre polimorfismo

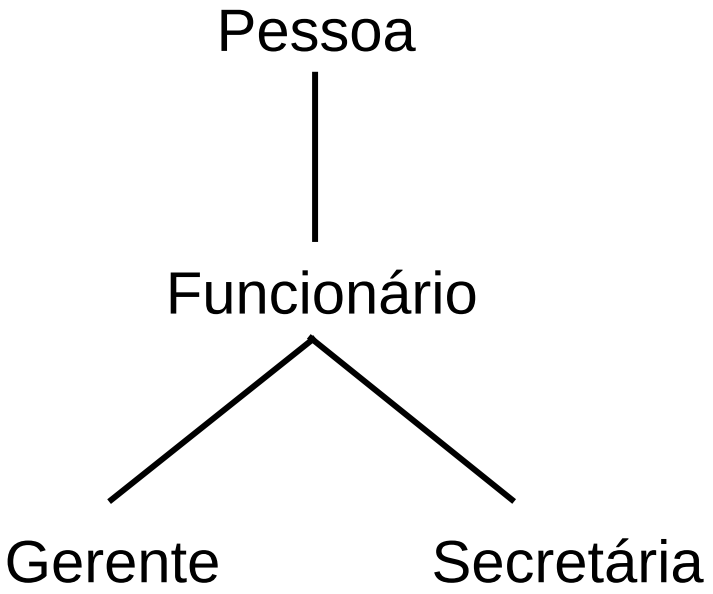
- polimorfismo é o nome formal para o fato de que quando precisamos de um objeto de determinado tipo, podemos usar uma versão mais especializada dele



Conversão de tipo em objetos

- podemos usar uma versão mais especializada quando esperamos um objeto da superclasse

- se precisarmos fazer a conversão ao tipo mais especializado teremos que fazê-lo explicitamente



Exemplo de conversão de tipo

- conversão explícita de um objeto de um tipo para outro:

```
Point pt = new Pixel(0,0,1); // OK! pixel é ponto
Pixel px = (Pixel)pt;        // OK! pt contém um pixel
pt = new Point();
px = (Pixel)pt;    // ERRO! pt agora contém um ponto
pt = new Pixel(0,0,0);
px = pt;           // ERRO! pt não é sempre um pixel
```

Mais sobre conversão de tipo

- note que a conversão de tipo só pode ser resolvida em tempo de execução
- só quando o programa estiver executando é que poderemos saber o valor que uma dada variável terá
- assim, poderemos decidir se a conversão é válida ou não

instanceof

- comando de Java que permite verificar a classe real de um objeto:

```
if (pt instanceof Pixel) {
 Pixel px = (Pixel)pt;
 ...
}
```

Ampliando o exemplo

- vamos aumentar a classe **Point** para fornecer um método que imprima na tela uma representação textual do ponto

```
class Point {
   ...
   void print() {
     System.out.println("Point ("+x+","+y+")");
   }
}
```

Ampliando o exemplo

Point e **Pixel** possuem um método que imprime o *ponto* representado

```
Point pt = new Point();       // ponto em (0,0)
Pixel px = new Pixel(0,0,0); // pixel em (0,0)

pt.print(); // Imprime: "Point (0,0)"
px.print(); // Imprime: "Point (0,0)"
```

Ampliando o exemplo

- a implementação desse método não é boa para um pixel pois não imprime a cor

- vamos *redefinir* o método print em **Pixel**

```
class Pixel extends Point {
 ...
 void print() {
  System.out.println("Pixel("+x+","+y+","+color+")");
 }
}
```

Ampliando o exemplo

- com essa modificação, a classe **Pixel** agora possui um método que imprime o *pixel* de forma correta:

```
Point pt = new Point();       // ponto em (0,0)
Pixel px = new Pixel(0,0,0); // pixel em (0,0)

pt.print(); // Imprime: "Point (0,0)"
px.print(); // Imprime: "Pixel (0,0,0)"
```

Amarração tardia (ou *late binding*)

- no exemplo do vetor de pontos:
  - cada classe possui sua própria codificação para o método **print**
  - ao percorrermos o vetor imprimindo os pontos, as versões corretas dos métodos são usadas
  - isso acontece porque as linguagens OO usam um recurso chamado amarração tardia ou *late binding*

Amarração tardia na prática

Graças a esse recurso, agora temos:

```
Point[] pontos = new Point[5];
pontos[0] = new Point();
pontos[1] = new Pixel(1,2,0);

pontos[0].print(); // Imprime: "Point (0,0)"
pontos[1].print(); // Imprime: "Pixel (1,2,0)"
```

Definição de amarração tardia

- é a capacidade de adiar a resolução de um método até o momento no qual ele deve ser efetivamente chamado

Mais sobre amarração tardia

- ou seja, a resolução do método acontecerá em tempo de execução, ao invés de em tempo de compilação

Mais sobre amarração tardia

- no momento da chamada, o método utilizado será o definido pela classe *real* do objeto

Amarração tardia x eficiência

- o uso de amarração tardia pode trazer perdas no desempenho dos programas
  - porque a cada chamada de método um processamento adicional deve ser feito
  - processamento adicional: descobrir a classe real do objeto

Amarração tardia x eficiência

- esse fato levou várias linguagens OO a permitir a construção de métodos *constantes*

- ou seja, métodos cujas implementações não podem ser redefinidas nas sub-classes

Valores constantes

- Java permite declarar um atributo ou uma variável local que, uma vez inicializada, tenha seu valor fixo

- para isso, usamos o modificador **final**:

```
class Banco {
   final int MAX_TAM = 10;
   ...
}
```

OBS: Na verdade, constante já foi vista na solução dos exercícios das aulas 3 e 5 (classe Banco).

Métodos constantes em Java

- para criarmos um método constante em Java devemos utilizar, também, o modificador **final**:

```
public class A {
   public final int f() {
      ...
   }
}
```

Classes constantes em Java

- uma classe inteira pode ser definida **final**
- nesse caso, em particular, a classe não pode ser estendida

```
public final class A {
   ...
}
```

Exercício: enunciado

Projete e implemente uma classe empregado. Todo empregado tem um nome, um salário e o ano de contratação. Para um dado empregado, deve ser possível: obter seu nome, obter seu salário, aumentar seu salário, obter o ano de contratação e imprimir seus dados na tela.

Um gerente realiza todas as operações realizadas por um empregado, mas diferentemente de empregado, ele possui uma secretária e um vetor de dois subordinados. Nesta classe, deve ser possível aumentar o salário de todos os seus subordinados, inclusive de sua secretária.

A secretária também possui todas as responsabilidades de empregado e deve anotar os cinco últimos nomes de pessoas que entraram em contato com seu gerente.

Escreva um programa de teste que crie uma loja, e, para esta loja, deve-se criar um vetor de empregados, incluindo um gerente e uma secretária. Efetue as operações possíveis para todas as classes existentes.

OBS: Na verdade, o final do enunciado deveria ser "Escreva um programa de teste que cria um vetor de empregados, incluindo um gerente e uma secretária. Efetue as operações possíveis para todas as classes existentes."

Exercício: solução (classe Empregado)

```
class Empregado{
  String nome;
  float salario;
  short ano;
```

Exercício: solução (classe Empregado)

```
class Empregado{
  String nome;
  float salario;
  short ano;

  Empregado(String n, float s,
            short a){
   nome = n;
   salario = s;
   ano = a;
  }
```

Exercício: solução (classe Empregado)

```
class Empregado{
  String nome;
  float salario;
  short ano;

  Empregado(String n, float s,
            short a){
   nome = n;
   salario = s;
   ano = a;
  }
  String obterNome{return nome;}
```

Exercício: solução (classe Empregado)

```
class Empregado{
  String nome;
  float salario;
  short ano;

  Empregado(String n, float s,
            short a){
   nome = n;
   salario = s;
   ano = a;
  }
  String obterNome{return nome;}


  float obterSalario{
   return salario;
}
```

Exercício: solução (classe Empregado)

```
class Empregado{
  String nome;
  float salario;
  short ano;

  Empregado(String n, float s,
            short a){
   nome = n;
   salario = s;
   ano = a;
  }
  String obterNome{return nome;}


  float obterSalario{
   return salario;
}
```

```
short obterAno{return ano;}
```

Exercício: solução (classe Empregado)

```
class Empregado{
  String nome;
  float salario;
  short ano;

  Empregado(String n, float s,
            short a){
   nome = n;
   salario = s;
   ano = a;
  }
  String obterNome{return nome;}


  float obterSalario{
   return salario;
}
```

```
short obterAno{return ano;}


void aumentaSal(float t){
 salario *= (1 + t);
}
```

Exercício: solução (classe Empregado)

```
class Empregado{
  String nome;
  float salario;
  short ano;

  Empregado(String n, float s,
            short a){
   nome = n;
   salario = s;
   ano = a;
  }
  String obterNome{return nome;}


  float obterSalario{
   return salario;
}
```

```
short obterAno{return ano;}


void aumentaSal(float t){
 salario *= (1 + t);
}

public String toString(){
 return "Empregado";
}
```

Exercício: solução (classe Empregado)

```
class Empregado{
  String nome;
  float salario;
  short ano;

  Empregado(String n, float s,
            short a){
   nome = n;
   salario = s;
   ano = a;
  }
  String obterNome{return nome;}


  float obterSalario{
   return salario;
}
```

```
  short obterAno{return ano;}


  void aumentaSal(float t){
   salario *= (1 + t);
  }

  public String toString(){
   return "Empregado";
  }


  void imprime(){
   System.out.println(this +
   ":" + nome + " " + ano +
   " " + salario);
  }
}
```

Exercício: solução (classe Secretaria)

```
class Secretaria extends
   Empregado{
   String[] contatos;
   int pos_livre = 0;
```

Exercício: solução (classe Secretaria)

```
class Secretaria extends
   Empregado{
   String[] contatos;
   int pos_livre = 0;

   Secretaria(String n, float
   s, short a){
     super(n, s, a);
     contatos = new String[5];
   }
```

Exercício: solução (classe Secretaria)

```
class Secretaria extends
   Empregado{
   String[] contatos;
   int pos_livre = 0;

   Secretaria(String n, float
   s, short a){
     super(n, s, a);
     contatos = new String[5];
   }

   void guardaNome(String n){
     contatos[pos_livre++] = n;
     if(pos_livre == 5)
       pos_livre = 0;
  }
```

Exercício: solução (classe Secretaria)

```
class Secretaria extends
   Empregado{
   String[] contatos;
   int pos_livre = 0;

   Secretaria(String n, float
   s, short a){
     super(n, s, a);
     contatos = new String[5];
   }


   void guardaNome(String n){
     contatos[pos_livre++] = n;
     if(pos_livre == 5)
       pos_livre = 0;
  }
```

```
   public String toString(){
     return "Secretaria";
   }
}
```

Exercício: solução (classe Gerente)

```
class Gerente extends Empregado{
  Secretaria secret;
  Empregado[] subord;
```

Exercício: solução (classe Gerente)

```
class Gerente extends Empregado{
  Secretaria secret;
  Empregado[] subord;


  Gerente(String n, float s,
   short a, Secretaria sec,
   Empregado e1, Empregado e2){
   super(n, s, a);
   subord = new Empregado[2];
   secret = sec;
   subord [0] = e1;
   subord [1] = e2;
  }
```

Exercício: solução (classe Gerente)

```
class Gerente extends Empregado{
  Secretaria secret;
  Empregado[] subord;


  Gerente(String n, float s,
   short a, Secretaria sec,
   Empregado e1, Empregado e2){
   super(n, s, a);
   subord = new Empregado[2];
   secret = sec;
   subord [0] = e1;
   subord [1] = e2;
  }

  public String toString(){
   return "Gerente";
  }
```

Exercício: solução (classe Gerente)

```
class Gerente extends Empregado{
  Secretaria secret;
  Empregado[] subord;


  Gerente(String n, float s,
   short a, Secretaria sec,
   Empregado e1, Empregado e2){
   super(n, s, a);
   subord = new Empregado[2];
   secret = sec;
   subord [0] = e1;
   subord [1] = e2;
  }

  public String toString(){
   return "Gerente";
  }
```

```
void aumento(){
 secret.aumentaSal(0.1F);
 int i;
 for(i = 0; i < 2; i++)
  subord[i].aumentaSal(0.15F);
}
```

Exercício: solução (classe Gerente)

```
class Gerente extends Empregado{
  Secretaria secret;
  Empregado[] subord;


  Gerente(String n, float s,
   short a, Secretaria sec,
   Empregado e1, Empregado e2){
   super(n, s, a);
   subord = new Empregado[2];
   secret = sec;
   subord [0] = e1;
   subord [1] = e2;
  }

  public String toString(){
   return "Gerente";
  }
```

```
  void aumento(){
   secret.aumentaSal(0.1F);
   int i;
   for(i = 0; i < 2; i++)
    subord[i].aumentaSal(0.15F);
  }


  void imprime(){
   super.imprime();
   secret.imprime();
   int i;
   for(i = 0; i < 2; i++)
    subord[i].imprime();
  }
}
```

Exercício: solução (classe Teste)

```
class Teste{
  public static void main(String[] args){
    Empregado[] lista = new Empregado[4];
    lista[0] = new Secretaria("Maria", 1000F, 2000);
    lista[1] = new Empregado("Jose", 500F, 1998);
    lista[2] = new Empregado("Joao", 700F, 2001);
    lista[3] = new Gerente("Vitor", 5000F, 1999, (Secretaria)
   lista[0], lista[1], lista[2]);
    for(int i = 0; i < 4; i++)
      lista[i].imprime();
    ((Gerente) lista[3]).aumento();
    for(int i = 0; i < 4; i++)
      lista[i].imprime();
  }
}
```