Polimorfismo

Prof. Ms. Peter Jandl Junior

J12B

Linguagem de Programação Orientada a Objetos

Ciência da Computação - UNIP – Jundiaí

POO::sobrecarga, herança, sobreposição e polimorfismo

- Na POO - Programação Orientada a Objetos são essenciais:
  - A sobrecarga,
  - A herança
  - A sobreposição e
  - O tratamento polimórfico.
- A rigor, todos estes mecanismos são manifestações diferentes do polimorfismo, que, de fato, é a característica chave na Orientação a Objetos.

Sobrecarga

- Mecanismo que possibilita existir métodos diferentes com mesmo nome.

Herança e Sobreposição

- Mecanismo que possibilita compartilhamento dinâmico de código.

Polimorfismo

- Característica que permite coexistência de muitas formas de representação de classes, seus elementos e seus objetos.

Classes abstratas

- Permite maior controle na criação de hierarquias de classes extensíveis.

Projeto de Classes

Herança::projeto:exemplo

- A representação de diferentes tipos de funcionários de uma empresa pode levar às seguintes conclusões:
  - **Funcionário**: todo aquele que trabalha na empresa, recebendo um salário para isso.
  - **Comissionado**: funcionário que recebe uma comissão sobre as vendas efetuadas.
  - **Gerente**: funcionário responsável pelas atividades dos funcionários lotados em um departamento específico.

Herança::projeto:exemplo

Características comuns são fatoradas na classe que se torna a superclasse da hierarquia:

- **Funcionario**

Características específicas determinam a construção de subclasses próprias:

- **Comissionado**
- **Gerente**

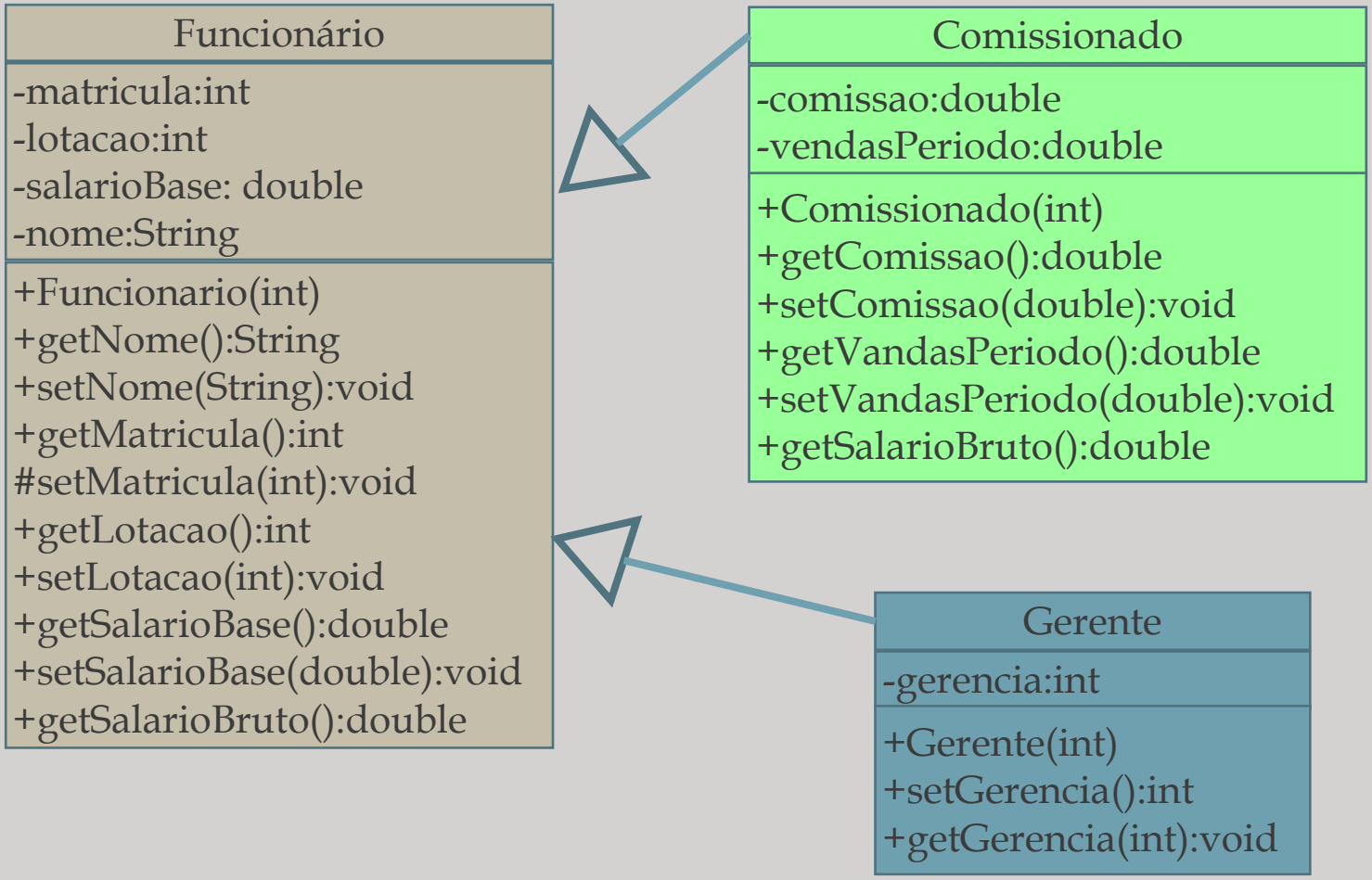
	Funcionario	Comissionado	Gerente
nome	☑	☑	☑
matricula	☑	☑	☑
salarioBase	☑	☑	☑
salarioBruto	☑	☑	☑
lotacao	☑	☑	☑
comissao		☑	
vendasPeriodo		☑	
gerencia			☑

Características comuns de todos (**Funcionario**).

Características específicas de **Comissionado**.

Características específicas de **Gerente**.

Herança::projeto:exemplo



Herança::projeto:exemplo

Funcionario.java

```
public class Funcionario {
    private int matricula, lotacao;
    private double salarioBase;
    private String nome;

    public Funcionario (int m) {
            matricula = m;
    }
    public String getNome () {
            return nome;  }
    public void setNome (String n) {
            nome = n;  }
    public int getMatricula () {
            return matricula;  }
    protected void setMatricula (int m) {
            matricula = m;  }
    public int getLotacao () {
            return lotacao;  }
    public void setLotacao (int l) {
            lotacao = l;  }
    public double getSalarioBase () {
            return salarioBase;  }
    public void setSalarioBase (double s) {
            salarioBase = s;  }
    public double getSalarioBruto () {
            return salarioBase;  }
}
```

Herança::projeto:exemplo

- Na implementação da classe **Funcionario**:
  - O único construtor presente indica que a matrícula deve ser determinada na criação de um objeto **Funcionario**.
  - O método **setMatricula()** é protegido porque, em geral, não se altera a matrícula de um funcionário.
  - Note que **salarioBruto** não é um atributo porque não é um valor armazenado e sim calculado pelo método **getSalarioBruto()**.

Herança::projeto:exemplo

Comissionado.java

```
public class Comissionado
    extends Funcionario {
    private double comissao;
    private double vendasPeriodo;

    public Comissionado (int m) {
            super(m);
    }

    public double getComissao () {
            return comissao;  }
    public void setComissao (double c) {
            comissao = c;  }
    public double getVendasPeriodo () {
            return vendasPeriodo;  }
    public void set VendasPeriodo
                        (double v) {
            vendasPeriodo = v;  }
    public double getSalarioBruto () {
            return getSalarioBase() +
             comissao*vendasPeriodo;  }
}
```

Sobreposição para substituição de funcionalidade

Herança::projeto:exemplo

- Na implementação da classe **Comissionado**:
  - É obrigatória a presença de algum construtor que acione o único construtor presente na superclasse o qual requer a matrícula necessário para criação de um objeto **Funcionario**.
  - O método **getSalarioBruto()** da classe **Comissionado** substitui (*override*) o método existente na classe **Funcionario**, pois é diferente a forma de cálculo do salário bruto de funcionários comissionados .

Herança::projeto:exemplo

Gerente.java

```
public class Gerente
    extends Funcionario {
    private int gerencia;

    public Gerente (int m) {
            super(m);
    }
    public int getGerencia () {
            return gerencia;  }
    public void setGerencia (int g) {
            gerencia = g;  }
}
```

- Na implementação da classe **Gerente**:
  - É obrigatória a presença do construtor que aciona aquele presente na superclasse e que requer a matrícula necessário para criação de um objeto **Funcionario**.
  - O atributo **gerencia** e os respectivos métodos *getter* e *setter* são triviais, embora exclusivos desta classe.

Herança::projeto:exemplo

- Criação de funcionário definindo sua matrícula:
  - Funcionario f = new Funcionario(3579);
- Ajuste de atributos do funcionário:
  - f.setNome(“José Sá"); // nome
  - f.setLotacao(18); // depto lotado
  - f.setSalarioBase(850); // salário base

Herança::projeto:exemplo

- Cria funcionário comissionado definindo matrícula:
  - Comissionado v = new Comissionado(5678);
- Ajusta atributos do funcionário comissionado:
  - v.setNome("Pedro Oliveira"); // nome
  - v.setLotacao(18); // depto lotado
  - v.setSalarioBase(650); // salário base
- Ajusta atributos específicos dos comissionados:
  - v.setComissao(0.03); // % comissão
  - v.setVendasPeriodo(25431.00); // meta de vendas

Herança::projeto:exemplo

- Cria gerente definindo matrícula:
  - Gerente g = new Gerente(1234);
- Ajusta atributos do funcionário gerente:
  - g.setNome("José de Souza Santos"); // nome
  - g.setLotacao(3); // depto lotado
  - g.setSalarioBase(4500); // salário base
- Ajusta atributos específicos dos gerentes:
  - g.setGerencia(18); // depto gerenciado

Polimorfismo

O principal mecanismo da orientação a objetos.

Polimorfismo

- Característica mais importante de qualquer linguagem de programação OO.
- *Poli morfos* → muitas formas
- É um mecanismo pelo qual são admitidas várias formas para um mesmo tipo.
- Assim:

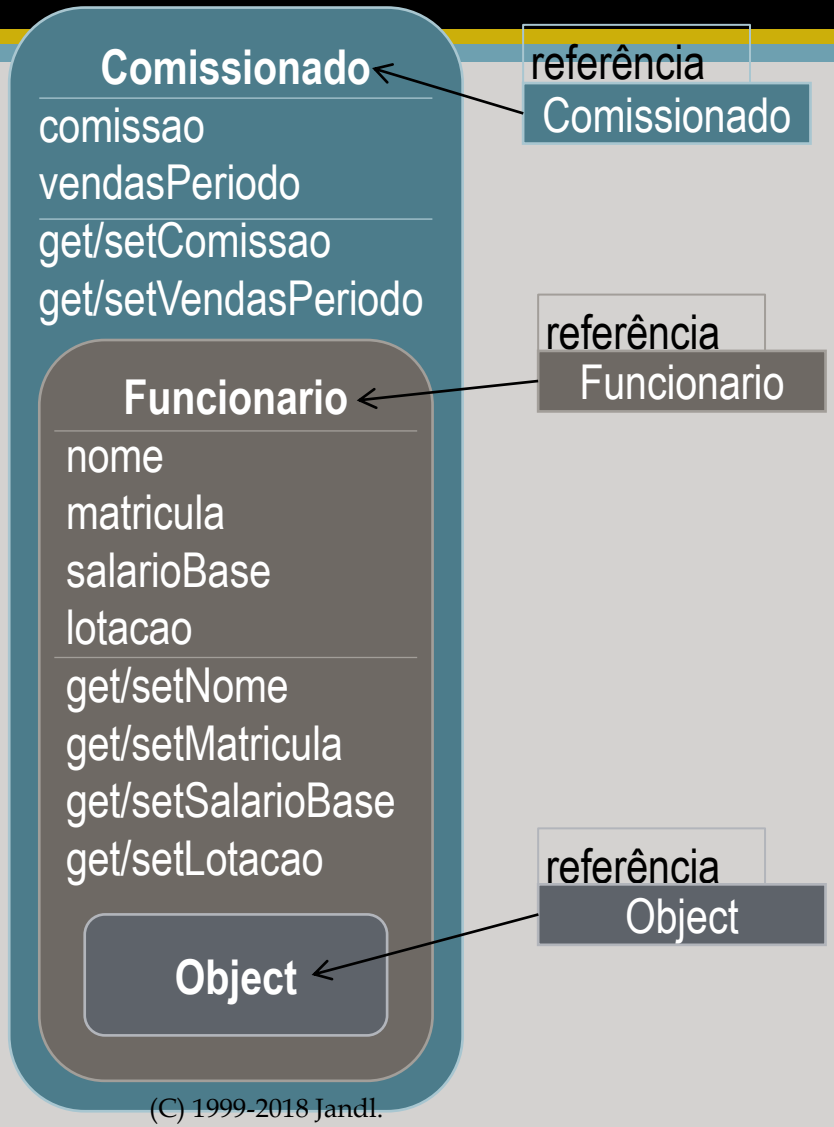
```
Object o;
o = new String("Polimorfismo");
o = new Integer(123456);
o = new FileReader("arquivoTexto.txt");
```

Uma variável do tipo Object pode armazenar referências de qualquer tipo de objeto!

Polimorfismo

- Permite que um objeto seja transparentemente tratado como sendo do seu tipo (i.e., ***como foi instanciado***) ou como qualquer outro tipo ascendente (superclasses do qual é derivado).
- Admite múltiplos tratamentos para um mesmo objeto.
- Possibilita a ***generalização***!
  Ou seja, é a operação inversa da herança, que promove a *especialização*.

Polimorfismo:como funciona



- Um objeto cuja classe é derivada de outra é como uma composição de objetos em camadas.
- O mais externo é o tipo real do objeto (no caso, **Comissionado**).
- O mais interno é sempre **Object**.
- O tipo da referência usada para acessar o objeto determina qual camada será efetivamente acessada.

Polimorfismo:exemplo

- A família de classes originada por **Funcionario**:

  Gerente g2 = new Gerente(2345);

- Seguem operações válidas (de *upcasting*):

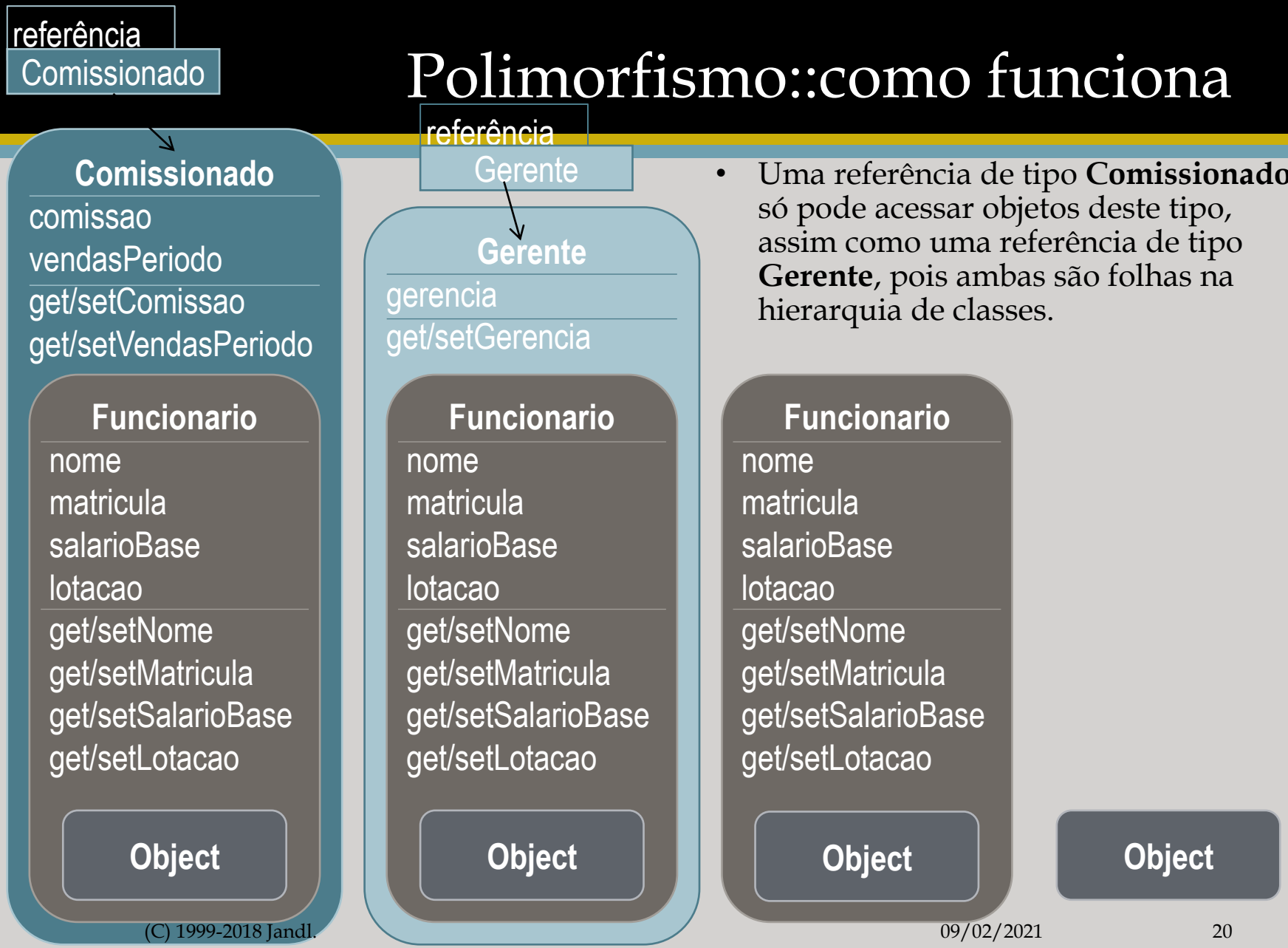
  Funcionario f2 = g2;

  Funcionario f3 = new Comissionado(5432);

  // Funcionario é superclasse das classes

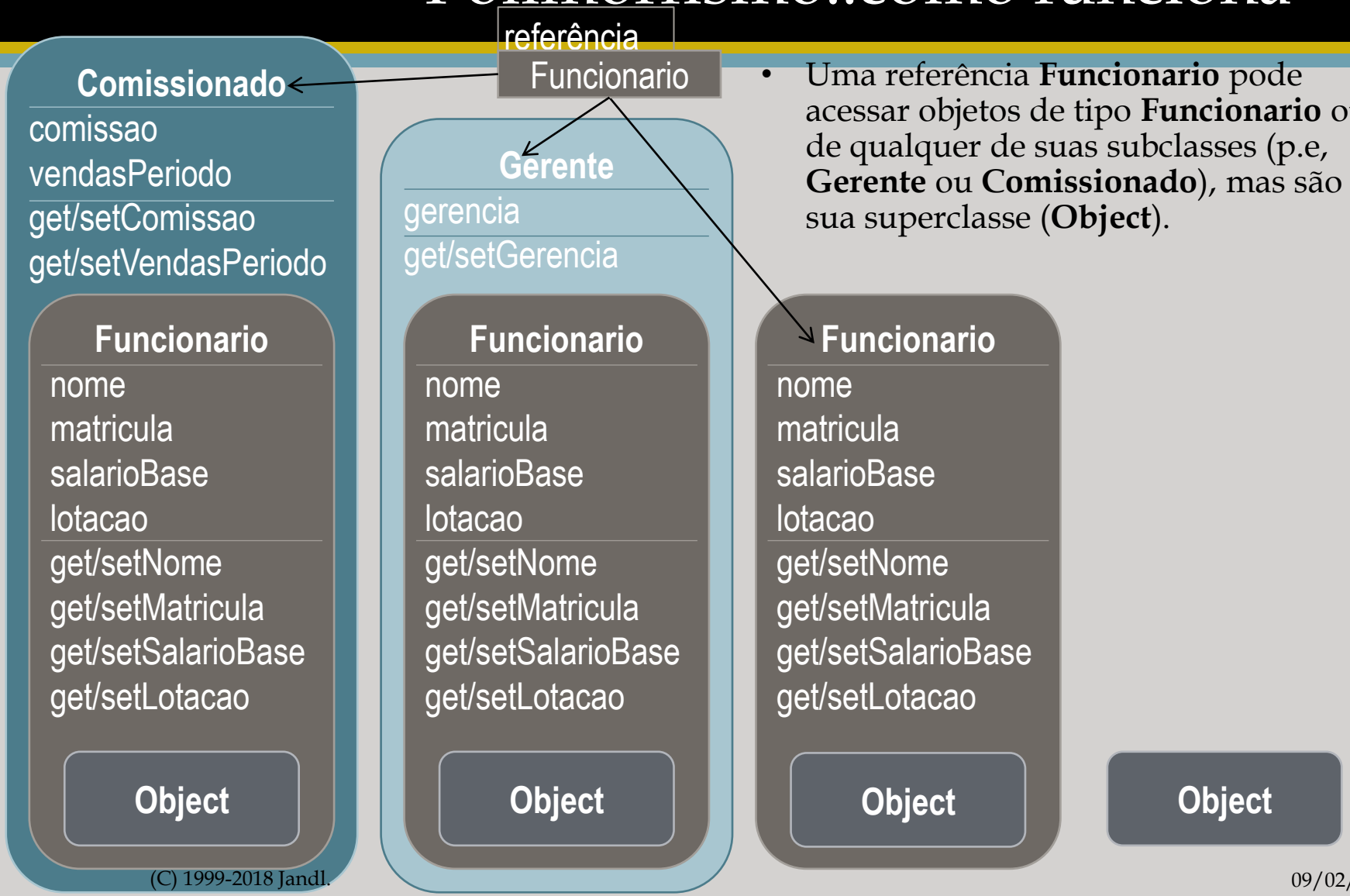
  // Gerente e Comissionado

Polimorfismo::como funciona



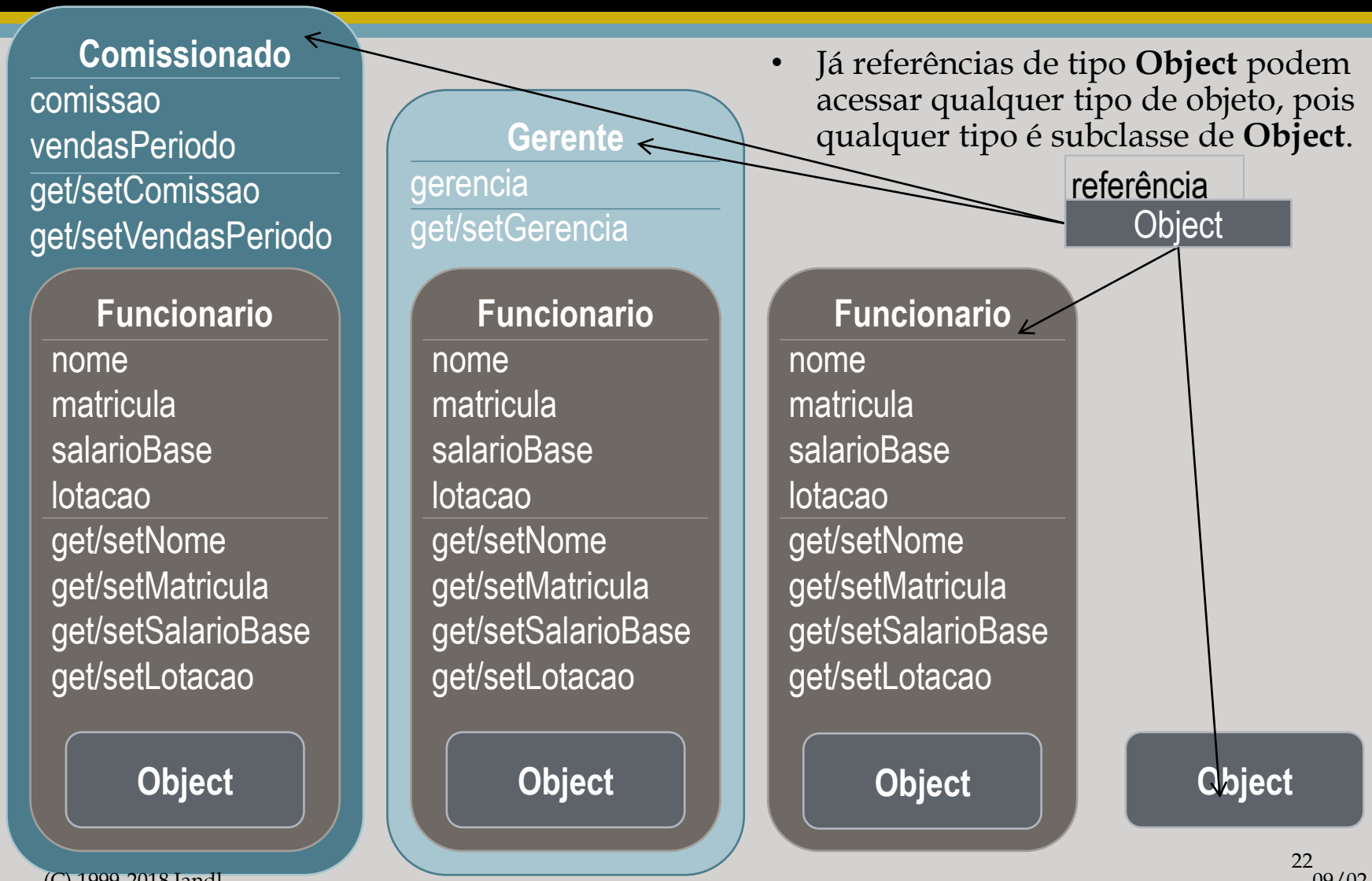
- Uma referência de tipo **Comissionado** só pode acessar objetos deste tipo, assim como uma referência de tipo **Gerente**, pois ambas são folhas na hierarquia de classes.

Polimorfismo::como funciona



- Uma referência **Funcionario** pode acessar objetos de tipo **Funcionario** ou de qualquer de suas subclasses (p.e, **Gerente** ou **Comissionado**), mas são de sua superclasse (**Object**).

Polimorfismo::como funciona



- Já referências de tipo **Object** podem acessar qualquer tipo de objeto, pois qualquer tipo é subclasse de **Object**.

Referências para Objetos

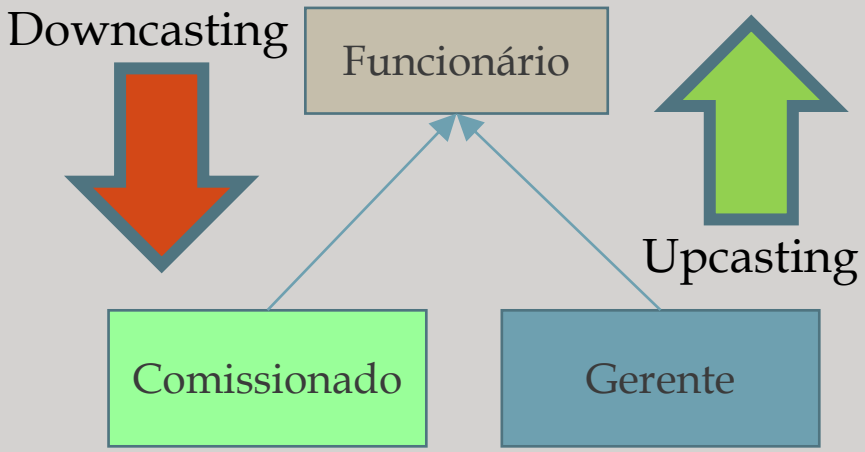
- Vale sempre observar que:
  - Uma referência é uma variável de tipo objeto;
  - Uma referência aponta para **UM** objeto de cada vez;
  - Basta trocar o valor de uma referência para que ela aponte para **OUTRO** objeto;
  - O valor **null** indica que a referência não aponta qualquer objeto válido.
  - Várias referências diferentes podem apontar para o mesmo objeto.

Referências para Objetos

- No Java, em particular:
  - As referências não são endereços de memória, mas identificadores de objetos definidos pela máquina virtual Java (JVM).
  - Por meio desses identificadores, a JVM pode verificar quando os objetos estão sendo referenciados (*uso aparente*).
  - Quando um objeto deixa de ser referenciado (i.e., quando não existem referências apontando para o objeto) ele é marcado pelo *Automatic Garbage Collector* para descarte (remoção da memória).

Polimorfismo::
Upcasting & Downcasting

- **Upcasting**
  Operação de tratamento polimórfico que ocorre quando se utiliza uma referência de tipo ancestral para o tratamento de um objeto.
- **Downcasting**
  Operação de coerção que ocorre quando se utiliza um referência de tipo descendente para o tratamento de um objeto.



Polimorfismo::*Upcasting*

- Quando uma referência de uma classe C recebe um objeto de alguma subclasse de C:
  - Funcionario f3 = new Comissionado(5432);
  - Object obj = new String("Upcasting");
- Representa uma operação implícita de *up type casting* ou coerção (conversão de tipo).
- O *upcasting/coerção:*
  - possibilita o uso mais genérico de um objeto;
  - só é válida quando utiliza subclasse do tipo real do objeto.

Polimorfismo::*Upcasting*

- Permite que métodos operem sobre (características comuns de) famílias de objetos ao invés de tipos específicos.
- Flexibiliza o armazenamento de objetos em *arrays* e outras estruturas de dados.
- Possibilita que métodos retornem objetos de tipos diferentes (mas de uma mesma família).
- Facilita alterações no projeto de sistemas.

Polimorfismo::*Upcasting*

Um *array* de tipo **Funcionario** pode conter objetos de qualquer tipo derivado de **Funcionario**.

```
Funcionario folha[] = new Funcionario[3];
Funcionario f = new Funcionario(1234);
f.setSalarioBase(850);
folha[0] = f;
Comissionado c = new Comissionado(2345);
f.setSalarioBase(650);
folha[1] = c;
folha[2] = new Gerente(3456);
folha[2].setSalarioBase(4500);
```

Nos elementos [1] e [2] do *array* (cujo tipo declarado é **Funcionario**) ocorre o *downcasting*.

Polimorfismo::*Upcasting*

```
double folhaBaseTotal = 0;

for(int i=0; i<folha.length; i++) {
    folhaBaseTotal =
       folhaBaseTotal + folha[i].getSalarioBase();

}

System.out.println("Folha total = " +
   folhaBaseTotal);
```

O processamento do *array* de tipo **Funcionario** permite acessar objetos de qualquer tipo derivado de **Funcionario**.

A despeito do tipo real, todas as características do tipo **Funcionario** estão disponíveis por meio das referências existentes no *array* de tipo **Funcionario**.

Polimorfismo::*Upcasting*

Um método pode receber um *array* de tipo **Funcionario**.

```
double getFolhaBaseTotal(Funcionario[] folha ){
    double folhaBaseTotal = 0;
   for(int i=0; i<folha.length; i++) {
      folhaBaseTotal =
          folhaBaseTotal + folha[i].getSalarioBase();
   }
   return folhaBaseTotal;
}
```

Como antes, a despeito do tipo real, todas as características do tipo **Funcionario** estão disponíveis por meio das referências existentes no *array* de tipo **Funcionario**.

Polimorfismo::*Downcasting*

- Quando uma referência de uma classe C é transformada em um objeto de alguma de suas superclasses:
  - Comissionado c = (Comissionado) f3;
  - String down = (String) obj;
- Representa uma operação explícita de *down type casting* ou coerção (conversão de tipo).
- O *downcasting* possibilita transformar uma referência genérico no tipo real do objeto (ou outro mais próximo).

Polimorfismo::*Downcasting*

- Só é válido quando utiliza a classe do tipo real do objeto (de sua instanciação) ou outra subclasse intermediária).
- Se a referência não pode ser transformada na tipo indicado, o *downcasting* provoca o lançamento da exceção: **ClassCastException**.
- Exemplos inválidos:
  - **Funcionario** f4 = new **Gerente** (6789);
  - **Comissionado** c = (**Comissionado**) f4;

Polimorfismo::operador *instanceof*

- O operador especial booleano **instanceof** permite verificar, em tempo de execução, se um objeto é de um tipo específico ou de suas subclasses.
- Sintaxe:
  - **<objeto> instanceof <Tipo>**
  - retornando:
    - **true** quando o objeto é uma instância direta do tipo indicado ou de alguma de suas subclasses.
    - **false** quando o objeto não é uma instancia do tipo indicado ou de qualquer uma de suas subclasses.

Polimorfismo::operador *instanceof*

- O uso do operador **instanceof** permite evitar erros em operações de *downcasting*.
  - Um exemplo:

```
Funcionario func = new Gerente(5678);
if (func instanceof Gerente) {
    Gerente g = (Gerente) func; // downcasting
    System.out.println("Gerente depto: " +
                                    g.getGerencia() );
}
```

  - Outro exemplo:

```
Integer n = new Integer (123456789);
if (n instanceof Number) {
    count= n.intValue();
}
```

Polimorfismo::uso na API Java

- Polimorfismo é usado extensivamente na API do Java. Apenas considerando **Object**:
  - Métodos herdados da classe **Object**:
    - getClass(), notify(), notifyAll(), toString(), wait().
  - Métodos que tomam argumentos do tipo **Object**:
    - Object.equals(Object), AWTEvent.setSource(Object), ObjectOutputStream.writeObject(Object), etc.
  - Métodos que retornam objetos do tipo **Object**:
    - JComboBox.getSelectedItem(), DefaultMutableTreeNode.getUserObject(), ObjectInputStream.readObject()

Recomendações de Estudo

- Java – Guia do Programador, 3ª Edição, P. JANDL Jr, Novatec, 2015.
- Java 6- Guia de Consulta Rápida, P. JANDL Jr, Novatec, 2008.
- Java 5- Guia de Consulta Rápida, P. JANDL Jr, Novatec, 2006.
- Introdução ao Java, P. JANDL Jr, Berkeley, 2002.