Java - Polimorfismo

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP

Polimorfismo

- O **Polimorfismo** nos permite **programar genericamente** ao invés de programar especificamente
  - Nos permite criar programar que **processam objetos que compartilham uma mesma superclasse** como se todos fossem daquela classe;
  - Simplifica a programação

Polimorfismo (cont.)

- É possível fazer com que objetos se comportem de maneira adequada automaticamente
  - Sem conhecer o tipo do objeto;
  - Desde que os objetos pertençam a uma mesma hierarquia de herança
- O mesmo nome e assinatura de métodos podem ser utilizados para causar diferentes ações
  - Dependendo do tipo do objeto.

TestePolimorfismo.java

```
public class Super {
    public void print(){
        System.out.println("Chamada de superclasse");
    }
}
```

```
public class Sub extends Super{
    public void print(){
        System.out.println("Chamada da subclasse");
    }
}
```

TestePolimorfismo.java (cont.)

```
public class JavaPolimorfismo {
   public static void main(String[] args) {
        Super sup = new Super();
        Sub sub = new Sub();
        Super poli = new Sub();

        sup.print();
        sub.print();
        poli.print();
    }
}
```

TestePolimorfismo.java (cont.)

```
Chamada de superclasse
Chamada da subclasse
Chamada da subclasse
```

TestePolimorfismo.java (cont.)

- Quando uma **variável da superclasse contém uma referência a um objeto da subclasse**, e esta referência é utilizada para invocar um método, a versão da subclasse é utilizada
  - O compilador Java permite isto por causa da relação de herança;
  - Um objeto da subclasse é um objeto da superclasse
    - O contrário não é verdadeiro.

Classes e Métodos Abstratos

- Algumas classes são utilizadas apenas em níveis altos de hierarquias de herança, sem a necessidade de instanciá-las em nossos programas
  - São as superclasses abstratas;
  - São classes incompletas, “faltando peças”;
  - Mais genéricas

Classes e Métodos Abstratos (cont.)

- Para tornar uma classe abstrata, utilizamos a palavra **abstract**:

```
public abstract class Classe(); // classe abstrata
```

- As classes abstratas normalmente contém um ou mais métodos abstratos:

```
public abstract void metodo(); // metodo abstrato
```

Classes e Métodos Abstratos (cont.)

- Um método abstrato não possui implementação
  - Uma classe que possui um método abstrato deve ser declarada abstrata, mesmo que possua outros métodos concretos;
  - Cada subclasse concreta deve fornecer implementações para cada método abstrato da superclasse.

Classes e Métodos Abstratos (cont.)

- Construtores e métodos *static* não podem ser declarados abstratos
  - Construtores não podem ser herdados, logo, nunca seriam implementados;
  - Embora os métodos *static* sejam herdados, não podem ser sobrescritos. Já que os métodos *abstract* precisam ser sobrescritos, não faz sentido declarar um método *static* como *abstract*

Classes e Métodos Abstratos (cont.)

- Uma subclasse pode herdar a interface ou a implementação de uma superclasse
  - Hierarquias projetadas para **herança de implementação** tendem a ter a funcionalidade nos níveis mais altos da hierarquia
    - Cada nova subclasse herda um ou mais métodos que foram implementados na superclasse

Classes e Métodos Abstratos (cont.)

- Hierarquias projetadas para **herança de interface** tendem a ter a funcionalidade nos níveis mais baixos da hierarquia
  - Uma superclasse especifica um ou mais métodos abstratos que devem ser sobrescritos especificamente por cada subclasse

Exemplo Polimorfismo e *Downcast*

- Considere uma classe `Veículo` com duas classes derivadas `Automóvel` e `Bicicleta`
- Essas classes têm três métodos, definidos para veículos de forma geral e redefinidas mais especificamente para automóveis e bicicletas;

Exemplo Polimorfismo e *Downcast* (cont.)

- As funções são:
  - `VerificaLista()`: para verificar o que precisa ser analisado no veículo;
  - `Reparar()`: para realizar os reparos e a manutenção necessária
  - `Limpa()`: para realizar procedimentos de limpeza do veículo
- A aplicação `Oficina` define um objeto que recebe objetos da classe `Veículos`.
  - Para cada veículo recebido, a oficina executa na sequencia os três métodos da classe `Veículo`.

Exemplo Polimorfismo e *Downcast* (cont.)

```
public class Veiculo {
    public Veiculo() {
    }
    public void verificarLista(){
        System.out.println("Verifica Veiculo");
    }
    public void reparar(){
        System.out.println("Reparar Veiculo");
    }
    public void limpa(){
        System.out.println("Limpar Veiculo");
    }
}
```

Exemplo Polimorfismo e *Downcast* (cont.)

```
public class Automovel extends Veiculo{
    public Automovel() {
    }
    public void verificarLista(){
        System.out.println("Verifica Automovel");
    }
    public void reparar(){
        System.out.println("Reparar Automovel");
    }
    public void limpa(){
        System.out.println("Limpar Automovel");
    }
}
```

Exemplo Polimorfismo e *Downcast* (cont.)

```
public class Moto extends Veiculo{
    public Moto() {
    }
    public void verificarLista(){
        System.out.println("Verifica Moto");
    }
    public void reparar(){
        System.out.println("Reparar Moto");
    }
    public void limpa(){
        System.out.println("Limpar Moto");
    }
}
```

Exemplo Polimorfismo e *Downcast* (cont.)

```
public class Oficina {
    private int num = 0;
    private final Random r = new Random();
    public Oficina() {
    }
    public Veiculo proximo(){
        Veiculo v;
        num = r.nextInt(2);
        if (num == 0)
            v = new Automovel();
        else
            v = new Moto();
        return v;
    }
```

Exemplo Polimorfismo e *Downcast* (cont.)

```
    public void manter(Veiculo v){
        v.verificarLista();
        v.reparar();
        v.limpa();
    }

    public int getNum(){
        return num;
    }
}
```

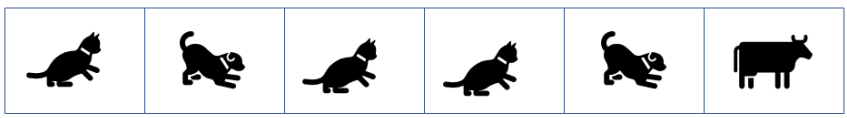
Exemplo Polimorfismo e *Downcast* (cont.)

```
public class DriverOficina {
    public static void main(String[] args) {
        Oficina of = new Oficina();
        Veiculo vec;
        for (int i = 0; i < 4; i++){
            vec = of.proximo();
            System.out.println(of.getNum());
            of.manter(vec);
        }
    }
}
```

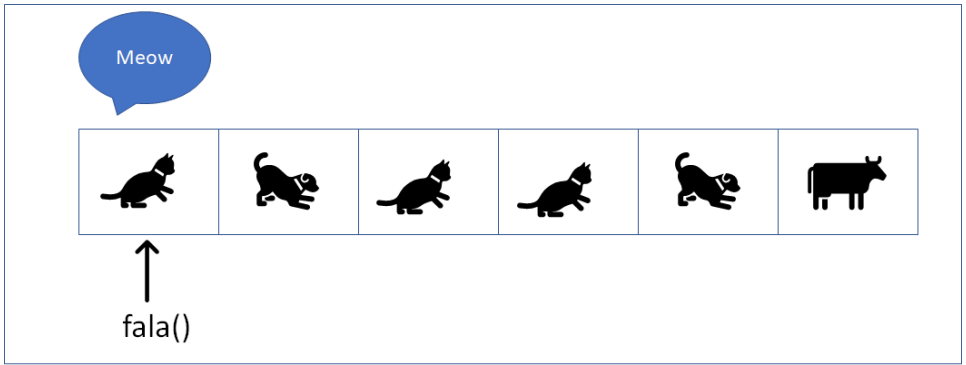
Exemplo Polimorfismo e *Downcast* (cont.)

```
1
Verifica Moto
Reparar Moto
Limpar Moto
1
Verifica Moto
Reparar Moto
Limpar Moto
0
Verifica Automovel
Reparar Automovel
Limpar Automovel
1
Verifica Moto
Reparar Moto
Limpar Moto
```

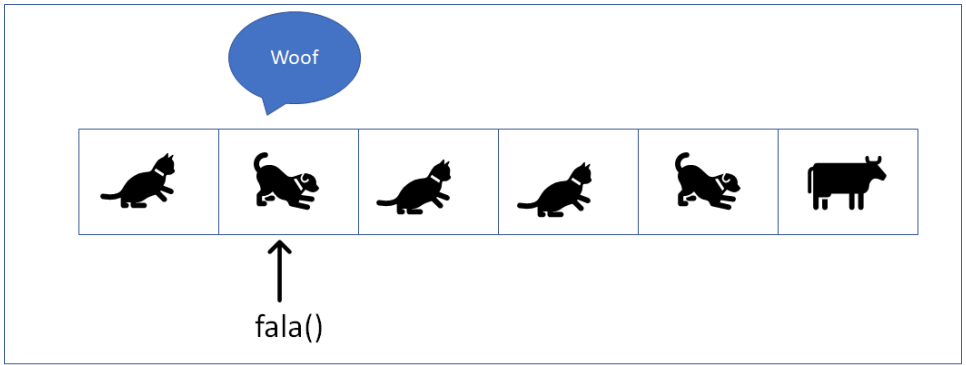
Exemplo Polimorfismo



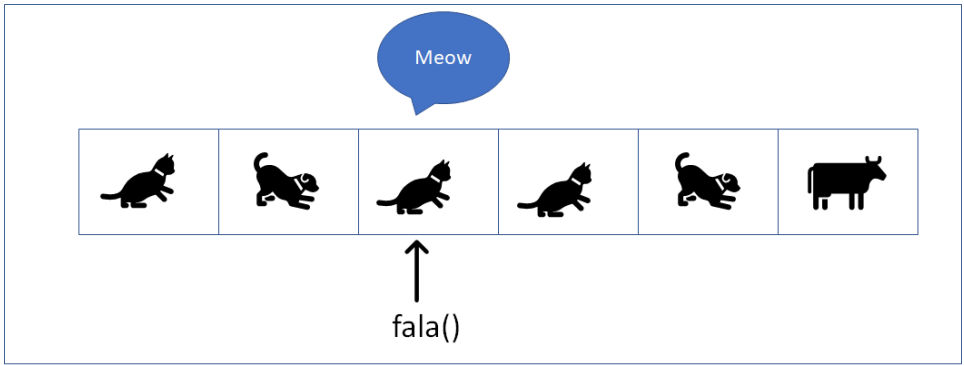
Exemplo Polimorfismo (cont.)



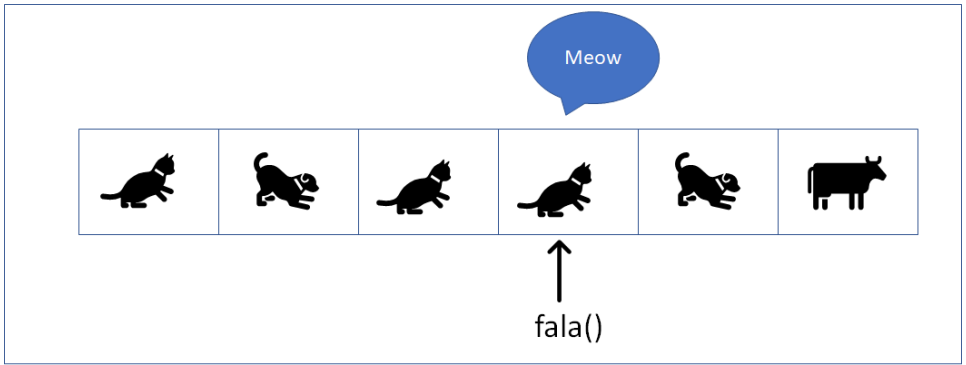
Exemplo Polimorfismo (cont.)



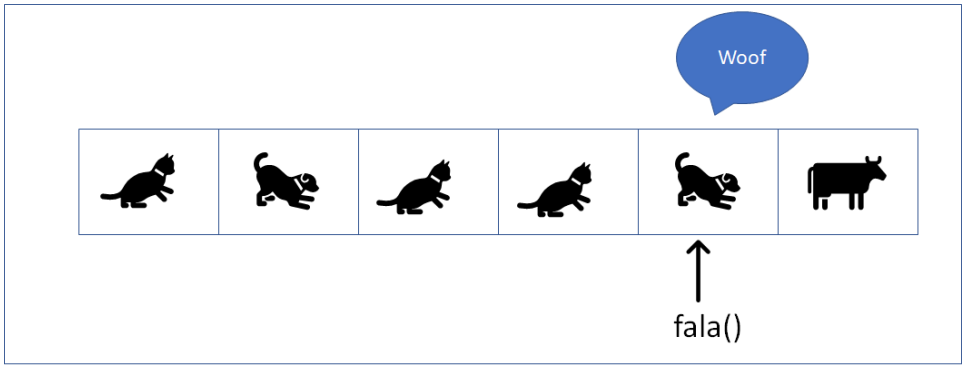
Exemplo Polimorfismo (cont.)



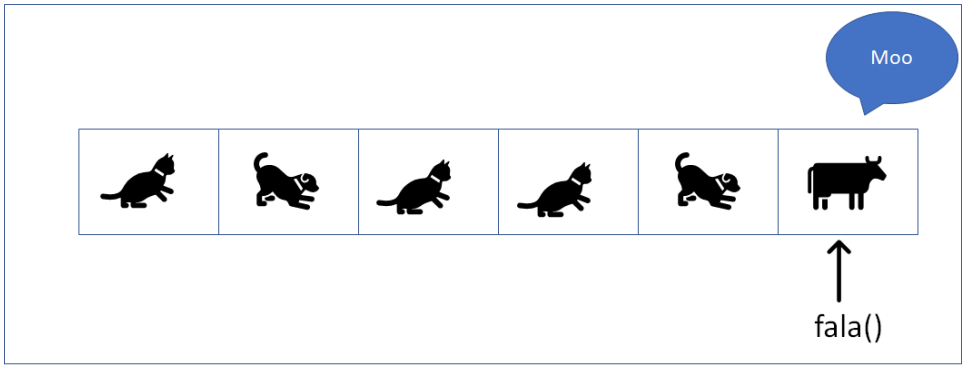
Exemplo Polimorfismo (cont.)



Exemplo Polimorfismo (cont.)



Exemplo Polimorfismo (cont.)



Exemplo Polimorfismo (cont.)

```
public abstract class Animal {
    public Animal(){System.out.println("Construtor Animal");}
    public abstract void fala();
}

public class Gato extends Animal {

    public Gato() {
        System.out.println("Construtor Gato");
    }

    public void fala(){
        System.out.println("Meow!");
    }
}
```

Exemplo Polimorfismo (cont.)

```
public class Cachorro extends Animal {
    public Cachorro(){
        System.out.println("Construtor Cachorro");
    }
    public void fala(){
        System.out.println("Goof!!");
    }
}

public class Vaca extends Animal {
    public Vaca(){
        System.out.println("Construtor Vaca");
    }
    public void fala(){
        System.out.println("Moo!!");
    }
}
```

Exemplo Polimorfismo (cont.)

```
public class DriverAnimal {

    public static void main(String[] args) {
        Random random = new Random();
        System.out.println("Instanciando animais\n");
        ArrayList<Animal> v = new ArrayList<>();
        for (int i = 0; i < 5; i++){
            Animal animal = switch(random.nextInt(3)) {
                case 0 -> new Gato();
                case 1 -> new Cachorro();
                case 2 -> new Vaca();
                default -> null;
            };

            v.add(animal);
            System.out.println("-----------------");
        }
```

Exemplo Polimorfismo (cont.)

```
        System.out.println("Percorrendo o vetor de animais\n");
        for (Animal animal : v){
            animal.fala();
        }
    }
}
```

Exemplo Polimorfismo (cont.)

```
Instancionado animais:

Construtor Animal
Construtor Vaca
-------------------
Construtor Animal
Construtor Gato
-------------------
Construtor Animal
Construtor Cachorro
-------------------
Construtor Animal
Construtor Cachorro
-------------------
Construtor Animal
Construtor Gato
-------------------
```

Exemplo Polimorfismo (cont.)

```
Percorrendo o vetor de animais

Moo!!
Meow!
Goof!!
Goof!!
Meow!
```

Métodos sobrecarregados em construtores

```
public class OverridableMethodConstructor {
    static abstract class A {
        A() { System.out.println("Constructing A");
            foo();
        }
        abstract void foo();
    }
    static class C extends A {
        C() {
            System.out.println("Constructing C");
        }
        void foo() {
            System.out.println("Using C");
        }
    }
    public static void main(String[] args) {
        C c = new C();
    }
}
```

Métodos sobrecarregados em construtores (cont.)

```
Constructing A
Using C
Constructing C
```

Exemplo Polimorfismo e *Downcast*

- Vamos revisitar o exemplo de uma empresa que possui diferentes tipos de empregados:
  - Horista
    - Recebe um valor fixo por hora trabalhada;
    - Após 40 horas, recebe 50 % a mais nas horas extras.
  - Assalariado
    - Recebe um salário fixo.

Exemplo Polimorfismo e *Downcast* (cont.)

- Comissionado
  - Recebe uma comissão aplicada sobre o valor das vendas realizadas.
- Comissionado Assalariado
  - Recebe um salário fixo mais uma comissão aplicada sobre o valor das vendas realizadas;

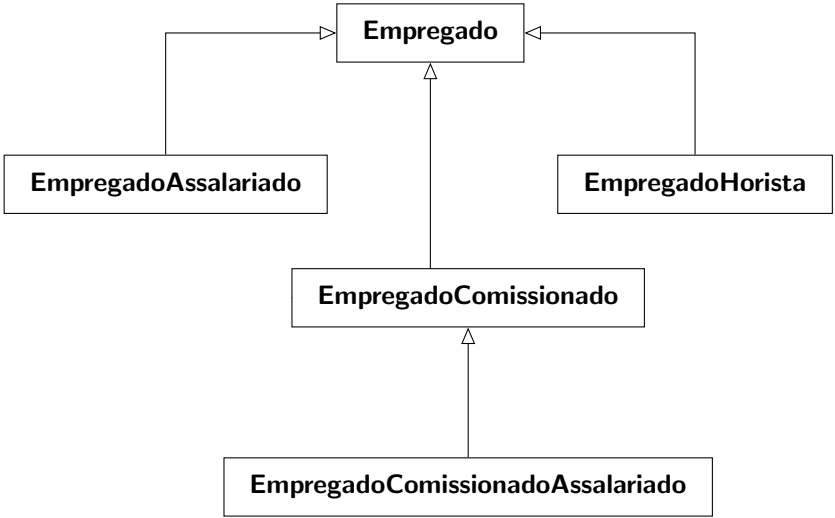
Exemplo Polimorfismo e *Downcast* (cont.)

- Especialmente para este período, deseja-se aplicar um bônus de 10 % para os funcionários assalariados comissionados;
- É um requisito que esta aplicação processe os funcionários polimorficamente

Exemplo Polimorfismo e *Downcast* (cont.)

- Vamos criar uma classe abstrata **Empregado**, que possua um método abstrato *salario()*
  - Calcula o salário de acordo com a implementação de cada subclasse;
  - Cada tipo de empregado será representado por uma subclasse;
  - A classe **EmpregadoComissionado** será superclasse direta da classe **EmpregadoComissionadoAssalariado**.

Exemplo Polimorfismo e *Downcast* (cont.)



Empregado.java

Empregado
–nome:string –sobrenome:string –documento:string
+Empregado(nome:string, sobrenome:string, documento:string) +setEmpregado(nome:string, sobrenome:string, documento:string) +toString() *+salario():double*

Empregado.java (cont.)

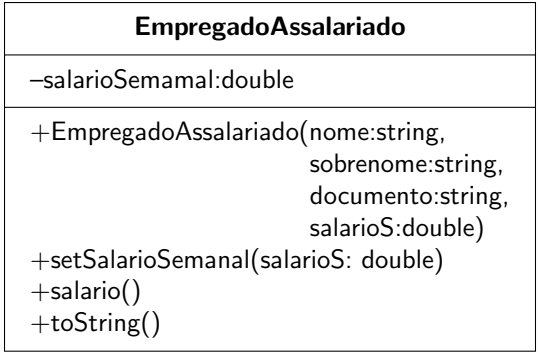
```
public abstract class Empregado {
    private String nome;
    private String sobrenome;
    private String documento;
    public Empregado(String nome,
                     String sobrenome,
                     String documento){
        setEmpregado(nome, sobrenome, documento);
    }
    public final void setEmpregado(String nome,
                                   String sobrenome,
                                   String documento){
        this.nome = nome;
        this.sobrenome = sobrenome;
        this.documento = documento;
    }
```

Empregado.java (cont.)

```
    public String toString(){
      return String.format("\n%s: %s\n%s: %s\n%s: %s",
                "Nome: ", nome,
                "Sobrenome", sobrenome,
                "Documento", documento);
    }

    public abstract double salario();
}
```

EmpregadoAssalariado.java



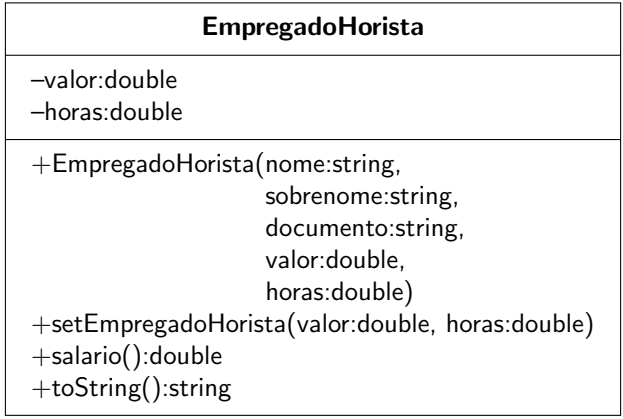
EmpregadoAssalariado.java (cont.)

```
public class EmpregadoAssalariado extends Empregado{
    private double salarioSemanal;
    public EmpregadoAssalariado(String nome,
                                String sobrenome,
                                String documento,
                                double salario){
        super(nome, sobrenome, documento);
        setSalarioSemanal(salario);
    }
```

EmpregadoAssalariado.java (cont.)

```
    public final void setSalarioSemanal(double salario){
        salarioSemanal = salario;
    }
    public double salario(){
        return salarioSemanal;
    }
    public String toString(){
        return String.format("%s\n%s: %s",
                super.toString(),
                "Salario semanal", salarioSemanal);
    }
}
```

EmpregadoHorista.java



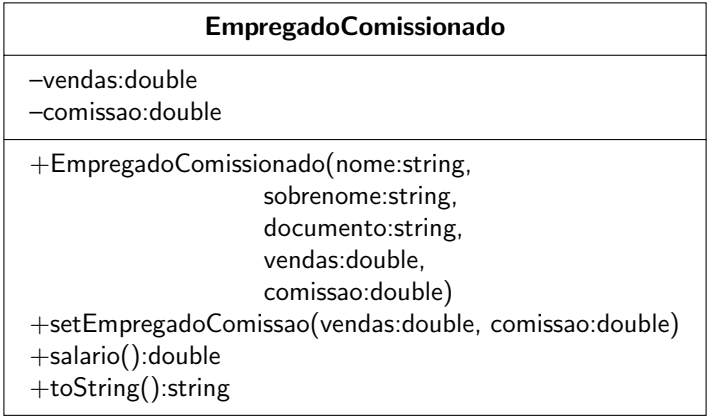
EmpregadoHorista.java (cont.)

```
public class EmpregadoHorista extends Empregado{
    private double valor;
    private double horas;
    public EmpregadoHorista(String nome,
                String snome,
                String doc,
                double valor,
                double hora){
        super(nome, snome, doc);
        setEmpregadoHorista(valor, hora);
    }
    public final void setEmpregadoHorista(double valor, double hora){
        this.valor = valor;
        this.horas = hora;
    }
```

EmpregadoHorista.java (cont.)

```
    public double salario(){
        if (horas <= 40)
            return valor * horas;
        else
            return 40*valor+(horas-40)*valor*1.5;
    }
    public String toString(){
        return String.format("%s\n%s: %.2f\n%s: %.2f",
                super.toString(),
                "Horas Trabalhadas", horas,
                "Valor por hora", valor);
    }
}
```

EmpregadoComissionado.java



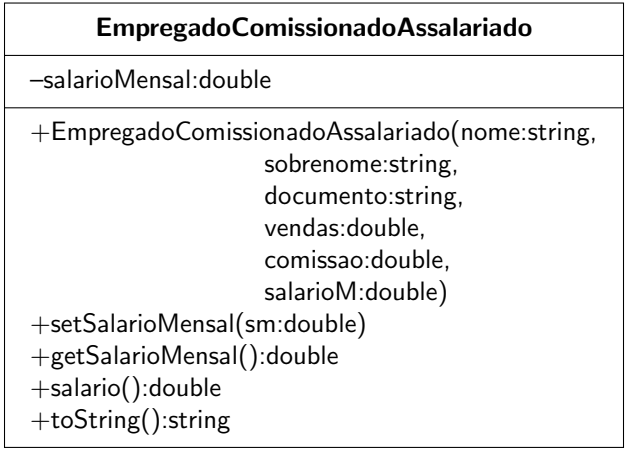
EmpregadoComissionado.java (cont.)

```
public class EmpregadoComissionado extends Empregado{
  private double vendas;
  private double comissao;
  public EmpregadoComissionado(String nome,
              String sobrenome,
              String doc,
              double v,
              double c){
      super(nome, sobrenome, doc);
      setEmpregadoComissionado(v, c);
  }
  public final void setEmpregadoComissionado(double v, double c){
      vendas = v;
      comissao = c;
  }
```

EmpregadoComissionado.java (cont.)

```
    public double salario(){
        return vendas * comissao;
    }
    public String toString(){
        return String.format("%s\n%s: %.2f\n%s: %.2f",
                super.toString(),
                "Total em vendas", vendas,
                "Comissao", comissao);
    }
}
```

EmpregadoComissionadoAssalariado.java



EmpregadoComissionadoAssalariado.java (cont.)

```
public class EmpregadoComissionadoAssalariado extends
    EmpregadoComissionado{
     private double salarioMensal;
    public EmpregadoComissionadoAssalariado(String nome,
                String sobrenome,
                String doc,
                double v,
                double c,
                double sm){
        super(nome, sobrenome, doc, v, c);
        setSalarioMensal(sm);
     }
     public final void setSalarioMensal(double sm){
        salarioMensal = sm;
     }
```

EmpregadoComissionadoAssalariado.java (cont.)

```
    public double getSalarioMensal(){
        return salarioMensal;
    }
    public double salario(){
        return getSalarioMensal()+super.salario();
    }
    public String toString(){
        return String.format("%s\n%s: %.2f",
                super.toString(),
                "Salario Mensal", getSalarioMensal());
    }
}
```

DriverEmpregado.java

```
public class DriverEmpregado{

    public static void main(String[] args) {
        EmpregadoAssalariado ea =
            new EmpregadoAssalariado("John", "Smith",
                "1111111-1", 800.00);
        EmpregadoHorista eh =
            new EmpregadoHorista("Karen", "Price",
                "2222222-2", 16.75, 40);
        EmpregadoComissionado ec =
            new EmpregadoComissionado("Sue", "Jones",
                "3333333-3", 10000, 0.06);
        EmpregadoComissionadoAssalariado eca =
            new EmpregadoComissionadoAssalariado("Bob", "Lewis", "
                44444444-4", 5000, 0.04, 300);
```

DriverEmpregado.java (cont.)

```
System.out.println("Processando individualmente\n");

System.out.printf("%.2f\n", ea.salario());
System.out.printf("%.2f\n", eh.salario());
System.out.printf("%.2f\n", ec.salario());
System.out.printf("%.2f\n", eca.salario());

Empregado vetor[] = new Empregado[4];

vetor[0] = ea;
vetor[1] = eh;
vetor[2] = ec;
vetor[3] = eca;
```

DriverEmpregado.java (cont.)

```
System.out.println("Processando polimorficamente");

for(Empregado elem : vetor){
    System.out.println(elem);
    System.out.println();
    //if (elem.getClass().equals(EmpregadoComissionadoAssalariado.class))
    //if ( elem.getClass().getSimpleName().equals(.EmpregadoComissionadoAssalariado")
        )
    if (elem instanceof EmpregadoComissionadoAssalariado){
        EmpregadoComissionadoAssalariado tmp = (
            EmpregadoComissionadoAssalariado)elem;
        tmp.setSalarioMensal( tmp.getSalarioMensal()*1.1 );
        System.out.println(tmp);
    }
    System.out.println("Salario: " + elem.salario());
    System.out.println();
}
```

DriverEmpregado.java (cont.)

```
        for (Empregado elem : vetor){
            System.out.println("O empregado e "
                + elem.getClass().getSimpleName());
        }
    }
}
```

DriverEmpregado.java (cont.)

```
Processando individualmente

800.00
670.00
600.00
500.00

Processando polimorficamente

Nome: : John
Sobrenome: Smith
Documento: 1111111-1
Salario semanal: 800.0

Salario: 800.0
```

DriverEmpregado.java (cont.)

```
Nome: : Karen
Sobrenome: Price
Documento: 2222222-2
Horas Trabalhadas: 40.00
Valor por hora: 16.75

Salario: 670.0


Nome: : Sue
Sobrenome: Jones
Documento: 3333333-3
Total em vendas: 10000.00
Comissao: 0.06

Salario: 600.0
```

DriverEmpregado.java (cont.)

```
Sobrenome: Lewis
Documento: 44444444-4
Total em vendas: 5000.00
Comissao: 0.04
Salario Mensal: 300.00

Nome: : Bob
Sobrenome: Lewis
Documento: 44444444-4
Total em vendas: 5000.00
Comissao: 0.04
Salario Mensal: 330.00
Salario: 530.0

O empregado e EmpregadoAssalariado
O empregado e EmpregadoHorista
O empregado e EmpregadoComissionado
O empregado e EmpregadoComissionadoAssalariado
```

Resolução Dinâmica

- Todas as chamadas aos métodos *toString()* e *salario()* são adequadas ao objeto que as chama
  - Resolvido em tempo de execução, baseado na classe do objeto;
  - **Resolução Dinâmica (Dynamic Binding)**

Operador *instanceof*

- O operador **instanceof** é utilizado para determinar se um objeto é de uma determinada classe

```
currentEmployee instanceof BasePlusCommissionEmployee
```

  - O valor retornado é *true* caso o objeto pertença à classe ou caso herde direta ou indiretamente da classe;
  - Retorna *false* caso contrário

Downcast

- Os métodos *getSalario()* e *setSalario()* são definidos apenas na subclasse **EmpregadoComissionadoAssalariado**
  - Não é possível invocá-lo em outras classes;
  - Uma vez determinado que se trata de um objeto da classe **EmpregadoComissionadoAssalariado**, é necessário realizar a conversão para o tipo adequado
    - De superclasse para subclasse;
    - Downcast

Downcast (cont.)

- Atribuir uma variável de uma superclasse a uma variável de uma subclasse sem realizar *cast* explícito resulta em erro de compilação;
- Antes de realizar um *downcast*, sempre realize um teste antes com o operador *instanceof*
- Durante a realização de um *downcast*, se o objeto não possuir um relacionamento com o objeto do *cast* será lançada uma exceção: **ClassCastException**

Método *getClass*

- Todos os objetos em Java sabem qual é a sua própria classe e podem acessar esta informação através do método *getClass()*
  - Todas as classes o herdam da classe **Object**
  - Retorna um objeto;
  - Neste objeto, invocamos o método *getName()*, que retorna o nome da classe que o objeto representa.

Criando e Utilizando Interfaces

- Vamos estender nosso exemplo anterior
  - Suponha que a mesma empresa deseja realizar pagamentos diversos utilizando uma mesma aplicação
    - Pagamento de empregados e também de notas fiscais de fornecedores;
    - Cada pagamento depende do tipo do empregado ou do conteúdo da nota fiscal.

Criando e Utilizando Interfaces (cont.)

- Apesar de aplicado a elementos distintos (empregados e notas fiscais), ambas as operações envolvem o pagamento de algo
  - Seria possível processar elementos distintos polimorficamente?
  - Seria possível implementar um conjunto de métodos comuns em classes não relacionadas?

Criando e Utilizando Interfaces (cont.)

- As **interfaces** de Java oferecem exatamente esta capacidade
  - Padronizam as formas em que elementos como sistemas e pessoas podem interagir entre si;
  - Por exemplo, os controles de um rádio definem a interface entre o usuário e os componentes internos
    - Permite que os usuários realizem um conjunto restrito de operações;
    - Diferentes rádios implementam os controles de forma diferente.

Criando e Utilizando Interfaces (cont.)

- A interface especifica o que deve ser feito
  - Quais operações.
  - Porém, não especifica como deve ser feito.
- Objetos de *software* também se comunicam via interfaces
  - Uma interface descreve um conjunto de métodos que podem ser invocados a partir de um objeto
    - Dizem ao objeto que uma tarefa deve ser realizada

Criando e Utilizando Interfaces (cont.)

- A declaração de uma interface começa com a palavra **interface** e contém apenas constantes e métodos abstratos
  - Todos os membros devem ser públicos;
  - Não devem especificar nenhum detalhe de implementação
    - Como variáveis ou métodos concretos.

Criando e Utilizando Interfaces (cont.)

- Logo, todos os métodos são implicitamente *public abstract*
  - Todos os atributos são implicitamente *public*, *static* e *final*
- De acordo com a especificação da linguagem Java
  - Não se declara métodos de uma interface com os modificadores *public* e *abstract*
  - Constantes não são declaradas com os modificadores *public*, *static* e *final*
  - São redundantes em interfaces.

Criando e Utilizando Interfaces (cont.)

- Para utilizar uma interface, uma classe concreta deve especificar que a implementa
  - Definido pela palavra **implements**
  - Deve declarar cada método da interface, com a mesma assinatura
  - Uma classe que não implementa todos os métodos de uma interface é abstrata, e assim deve ser declarada.

Criando e Utilizando Interfaces (cont.)

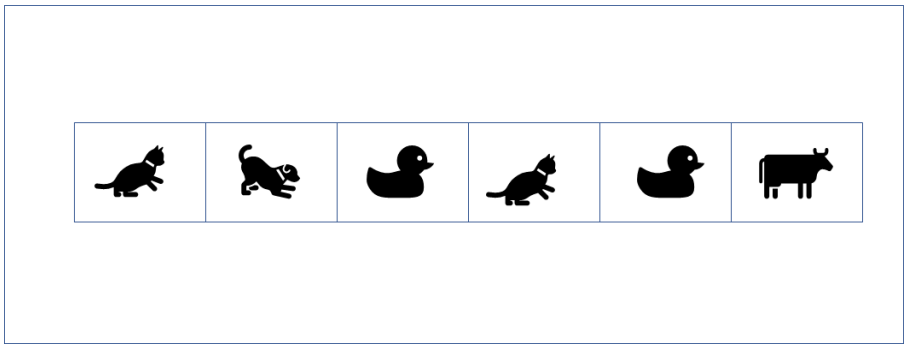
- Interfaces são utilizadas quando classes não relacionadas precisam compartilhar métodos e constantes
  - Objetos destas classes podem ser processados polimorficamente
  - Objetos de classes que implementam uma mesma interface respondem aos mesmos métodos.

Criando e Utilizando Interfaces (cont.)

- Outra aplicação de interfaces é na substituição de classes abstratas em que não há implementação padrão a ser herdada
  - Métodos ou atributos
  - Interfaces normalmente são públicas, logo, podem ser declaradas em arquivos próprios, com extensão *.java*

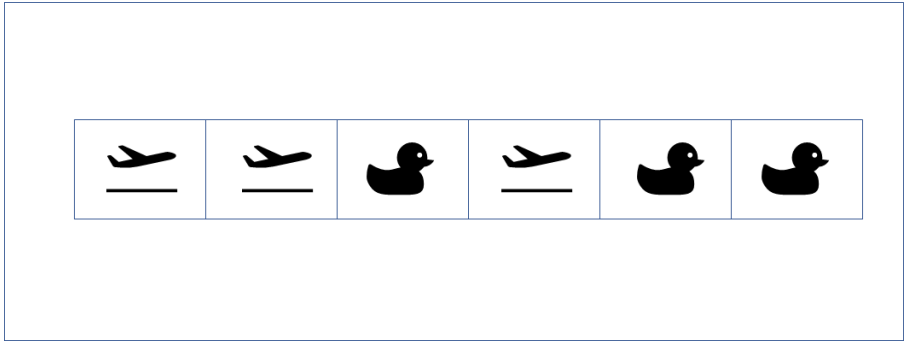
Exemplo 1

Podemos adicionar uma nova classe: Pato



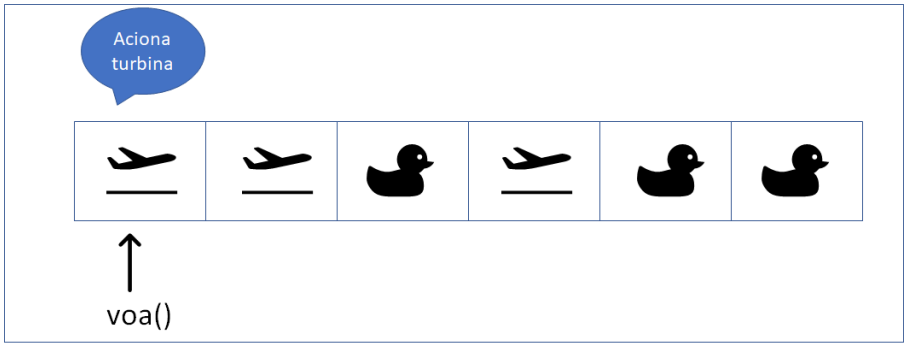
Exemplo 1 (cont.)

Qual é a relação destas classes?



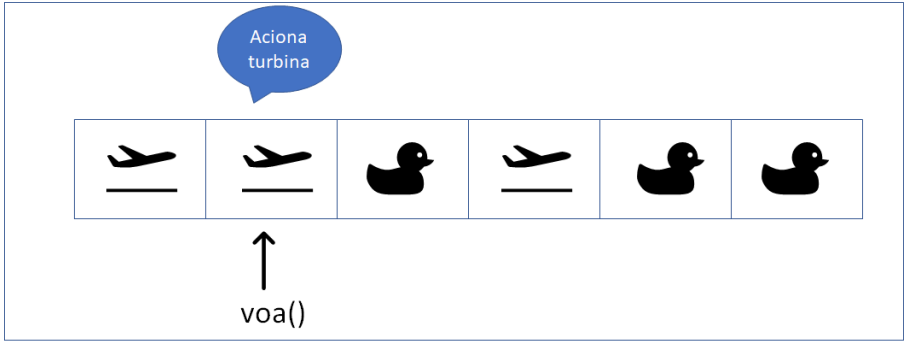
Exemplo 1 (cont.)

Qual é a relação destas classes?



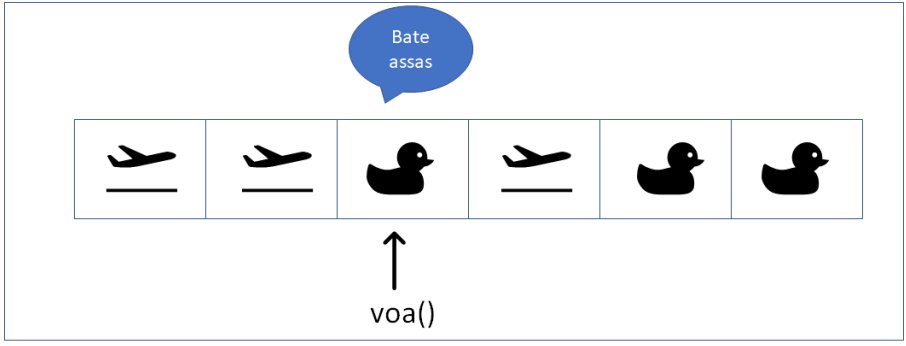
Exemplo 1 (cont.)

Qual é a relação destas classes?



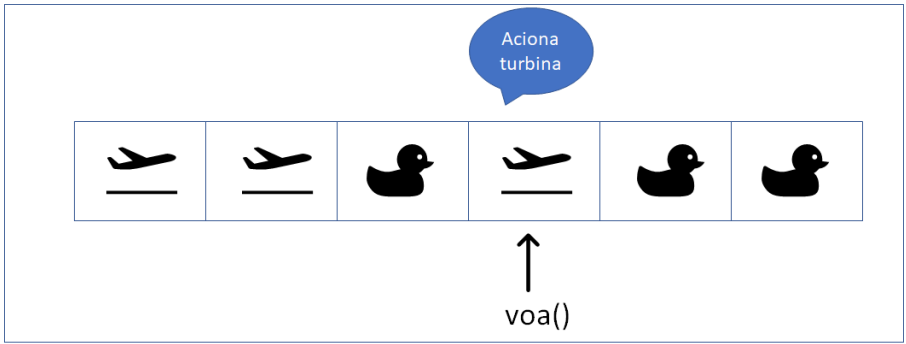
Exemplo 1 (cont.)

Qual é a relação destas classes?



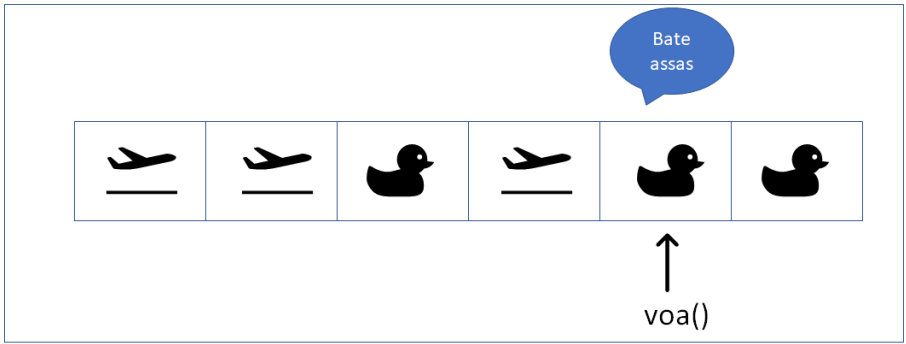
Exemplo 1 (cont.)

Qual é a relação destas classes?



Exemplo 1 (cont.)

Qual é a relação destas classes?



Exemplo 1 (cont.)

```
public interface Voador {
    void voa();
}

public class Aviao implements Voador {
    public Aviao() {
        System.out.println("Construtor Aviao");
    }

    public void voa(){
        System.out.println("Aciona turbina!");
    }
}
```

Exemplo 1 (cont.)

```
public class Pato extends Animal implements Voador {
    public Pato() {
        System.out.println("Construtor Pato");
    }

    public void fala(){
        System.out.println("Quack quack!");
    }

    public void voa() {
        System.out.println("Bate as assas");
    }
}
```

Exemplo 1 (cont.)

```
public class DriverInterface {

    public static void main(String[] args) {
        Random random = new Random();
        System.out.println("Instanciando objetos voadores");
        ArrayList<Voador> v = new ArrayList<>();
        for (int i = 0; i < 5; i++){
            Voador voador;
            voador = switch(random.nextInt(2)) {
                case 0 -> new Pato();
                default -> new Aviao();
            };

            v.add(voador);
            System.out.println("---------------");
        }
```

Exemplo 1 (cont.)

```
        System.out.println("Percorrendo o vetor de objetos voadores");
        for (Voador voador : v){
            voador.voa();
        }
    }
}
```

Exemplo 1 (cont.)

```
Instanciando objetos voadores
Construtor Animal
Construtor Pato
_______________

Construtor Aviao
_______________

Construtor Aviao
_______________

Construtor Animal
Construtor Pato
_______________

Construtor Animal
Construtor Pato
_______________
```

Exemplo 1 (cont.)

```
Percorrendo o vetor de objetos voadores
Bate as assas
Aciona turbina!
Aciona turbina!
Bate as assas
Bate as assas
```

Exemplo 2

- Vejamos o exemplo de uma interface chamada **Pagavel**
  - Descreve a funcionalidade de qualquer objeto capaz de ser pago e, portanto, fornece um método para determinar a quantidade a ser paga

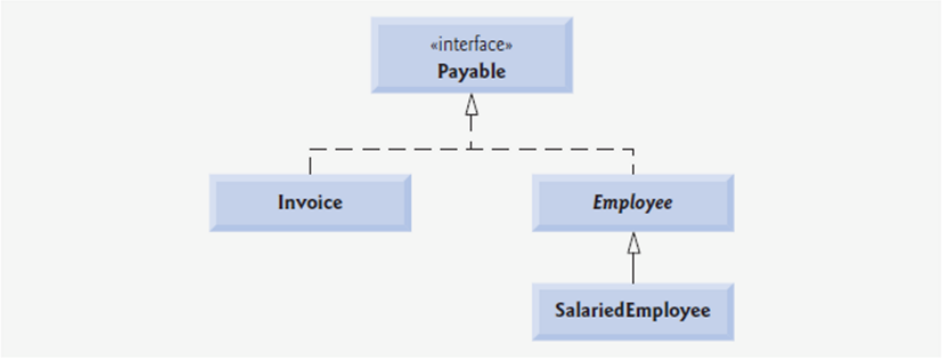
Exemplo 2 (cont.)

- A interface **Pagavel** determina a quantia a ser paga a um empregado ou uma nota fiscal
  - Metodo double getValorAPagar()
  - Uma versão genérica do método salario()
    - Pode ser aplicada a diferentes classes.

Exemplo 2 (cont.)

- Depois de declararmos esta interface, definiremos a classe **NotaFiscal**, que a implementará;
- Também modificaremos a classe **Empregado** para implementar a interface
  - A subclasse **EmpregadoAsssalariado** também será modificada.

Exemplo 2 (cont.)



- O relacionamento entre uma classe e uma interface em UML é chamado de realização
  - Uma classe realiza os métodos de uma interface;

Exemplo 2 (cont.)

- Representado por uma linha pontilhada com seta vazada.

Pagavel.java

```
public interface Pagavel
{
    // calcula o pagamento, não possui implementação
    double getValorAPagar();
}
```

NotaFiscal.java

```
public class NotaFiscal implements Pagavel {
    private String numeroPeca;
    private String descricaoPeca;
    private int quantidade;
    private double precoPorItem;

    public NotaFiscal(String peca, String descricao, int numero, double
         preco){
        numeroPeca = peca;
        descricaoPeca = descricao;
        setQuantidade(numero);
        setPrecoPorItem(preco);
    }

    public void setQuantidade(int numero){
        quantidade = numero > 0 ? numero : 0;
    }
```

NotaFiscal.java (cont.)

```
public int getQuantidade(){
    return quantidade;
}

public void setPrecoPorItem(double preco){
    precoPorItem = preco > 0 ? preco : 0.0;
}

public double getPrecoPorItem(){
    return precoPorItem;
}

public String getNumeroPeca() {
    return numeroPeca;
}

public void setNumeroPeca(String numeroPeca) {
    this.numeroPeca = numeroPeca;
}
```

NotaFiscal.java (cont.)

```
public String getDescricaoPeca() {
    return descricaoPeca;
}

public void setDescricaoPeca(String descricaoPeca) {
    this.descricaoPeca = descricaoPeca;
}

public String toString(){
    return String.format("\n%s \n%s: %s (%s) \n%s: %d \n%s: $%.2f",
        "Nota", "peca numero",
        getNumeroPeca(), getDescricaoPeca(),
        "quantidade", getQuantidade(),
        "preco por item", getPrecoPorItem());
}
```

NotaFiscal.java (cont.)

```
    public double getValorAPagar(){
        return getQuantidade() * getPrecoPorItem();
    }
}
```

Empregado.java

```
public abstract class Empregado implements Pagavel {
    private String nome;
    private String sobrenome;
    private String documento;

    public Empregado(String n, String s, String d){
        nome = n;
        sobrenome = s;
        documento = d;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
```

Empregado.java (cont.)

```
public String getSobrenome() {
    return sobrenome;
}

public void setSobrenome(String sobrenome) {
    this.sobrenome = sobrenome;
}

public String getDocumento() {
    return documento;
}

public void setDocumento(String documento) {
    this.documento = documento;
}
```

Empregado.java (cont.)

```
    public String toString(){
        return String.format("\n%s %s\n%s: %s",
                getNome(), getSobrenome(),
                "documento", getDocumento());
    }
}
```

EmpregadoAssalariado.java

```
public class EmpregadoAssalariado extends Empregado {
    private double salarioMensal;

    public EmpregadoAssalariado(String n, String s, String d, double
        salario) {
        super(n, s, d);
        this.salarioMensal = salario;
    }

    public double getSalarioMensal() {
        return salarioMensal;
    }

    public void setSalarioMensal(double salario) {
        this.salarioMensal = salario < 0.0? 0.0 : salario;
    }
```

EmpregadoAssalariado.java (cont.)

```
    public String toString(){
        return String.format("\n%s %s\n%s: $%.2f",
            "Empregado Assalariado", super.toString(),
            "Salario", getSalarioMensal());
    }

    public double getValorAPagar(){
        return getSalarioMensal();
    }
}
```

TesteInterfacePagavel.java

```
public class TesteInterfacePagavel {

    public static void main(String[] args) {
        Pagavel pagavel[] = new Pagavel[4];

        pagavel[0] = new NotaFiscal("01234", "banco",
            2, 375.0);
        pagavel[1] = new NotaFiscal("56789", "pneu", 4,
            79.95);
        pagavel[2] = new EmpregadoAssalariado("John",
            "Smith", "111-11-1111",800.00);
        pagavel[3] = new EmpregadoAssalariado("Lisa",
            "Barnes", "888-88-8888",1200.00);
```

TesteInterfacePagavel.java (cont.)

```
        System.out.println("Notas Fiscais e Empregados processados
            polimorficamente");

        for (Pagavel elem : pagavel){
            System.out.printf("\n%s \n%s: $%.2f\n",
                elem.toString(),"Pagamento Devido",
                elem.getValorAPagar());
        }
    }

}
```

TesteInterfacePagavel.java (cont.)

```
Notas Fiscais e Empregados processados polimorficamente

Nota
peca numero: 01234 (banco)
quantidade: 2
preco por item: $375.00
Pagamento Devido: $750.00


Nota
peca numero: 56789 (pneu)
quantidade: 4
preco por item: $79.95
Pagamento Devido: $319.80
```

TesteInterfacePagavel.java (cont.)

```
Empregado Assalariado
John Smith
documento: 111-11-1111
Salario: $800.00
Pagamento Devido: $800.00

Empregado Assalariado
Lisa Barnes
documento: 888-88-8888
Salario: $1200.00
Pagamento Devido: $1200.00
```

Exemplo

- O relacionamento **é um** existe entre superclasses e subclasses e entre interfaces e classes que as implementam
  - Quando um método recebe por parâmetro uma variável de superclasse ou de interface, o **método processa o objeto recebido polimorficamente**

Exemplo (cont.)

- Através de uma referência a uma superclasse ou interface, podemos **polimorficamente invocar qualquer método definido na superclasse ou interface**
  - Através de uma referência a uma interface, podemos invocar polimorficamente qualquer método definido na interface

Exemplo (cont.)

- Todos os métodos da classe **Object** podem ser invocados a partir de uma referência a uma interface

Declarando Constantes em Interfaces

- Como dito anteriormente, uma interface pode ser utilizada para declarar constantes
  - Implicitamente, são public, static e final, o que não precisa ser declarado no código.

Declarando Constantes em Interfaces (cont.)

- Um uso popular é declarar um conjunto de constantes que possa ser utilizado em várias declarações de classes
  - Uma classe pode utilizar as constantes importando a interface;
  - Refere-se ao nome da interface e ao nome da constante, separados por **.**;
  - Se for realizado *static import*, o nome da interface pode ser omitido

Declarando Constantes em Interfaces (cont.)

```
public interface Constants
{
        int ONE = 1;
        int TWO = 2;
        int THREE = 3;
}
```

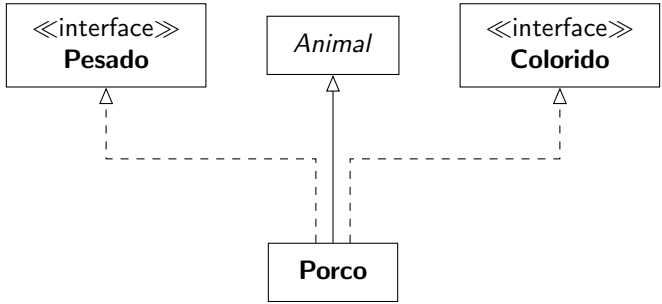
Herança Múltipla e Interfaces

- Java não fornece suporte à herança múltipla
  - No entanto, uma classe pode implementar mais do que uma interface;
  - Desta forma, é possível determinar que uma classe "absorva" o comportamento de diferentes interfaces
  - Ainda é necessário implementar as interfaces.

Herança Múltipla e Interfaces (cont.)

- Para nosso exemplo, suponha que possuímos interfaces **Pesado** e **Colorido**, uma classe abstrata **Animal** e queremos implementar uma classe **Porco**
  - Os objetos da classe **Porco** devem saber seu *peso* e sua *cor*, além de serem animais;
  - Desta forma, podemos herdar diretamente da classe **Animal** e implementar as duas interfaces.

Herança Múltipla e Interfaces (cont.)



Herança Múltipla e Interfaces (cont.)

```
public abstract class Animal {
    public abstract void fazerBarulho();
}

public interface Pesado {
    double obterPeso();
}

public interface Colorido {
    String obterCor();
}
```

Herança Múltipla e Interfaces (cont.)

```
public class Porco extends Animal implements Pesado, Colorido {
    public void fazerBarulho(){
        System.out.println("Oinc!");
    }

    // Implementação da interface Pesado
    public double obterPeso(){
        return 50.0;
    }

    //Implementação da interface Colirido
    public String obterCor(){
        return "Preto";
    }
```

Herança Múltipla e Interfaces (cont.)

```
    //Uma propriedade só do porco
    public boolean enlameado(){
        return true;
    }

    public String toString(){
        return String.format("\n%s: %s \n%s: %f \n%s %s",
                "Cor", obterCor(),
                "Peso", obterPeso(),
                "Enlameado?", enlameado());
    }
}
```

Herança Múltipla e Interfaces (cont.)

```
public class PorcoDriver {

    public static void main(String[] args) {
        // TODO code application logic here
        Porco porco = new Porco();
        System.out.println(porco);
        porco.fazerBarulho();
    }
}
```

Herança Múltipla e Interfaces (cont.)

```
Cor: Preto
Peso: 50.000000
Enlameado? true
Oinc!
```

Interfaces Comuns da API Java

- As interfaces da API java permite que usemos nossas próprias classes dentro de frameworks fornecidos pelo Java
- Algumas das mais populares são:
  - *Comparable*
  - *Serializable*
  - *Runnable*
  - *GUI event-listener interfaces*
  - *Swing Constants*

Interfaces Comuns da API Java (cont.)

- **Comparable**
  - Permite que objetos de classes que a implementam sejam comparados;
  - Contém apenas um método:
    - *compareTo()*
  - Compara o objeto que o invocou com um objeto passado como argumento;
  - Deve retornar:
    - Um número negativo, caso o primeiro objeto seja menor;
    - Zero, caso os objetos sejam iguais;
    - Um número positivo, caso o primeiro objeto seja maior.

Interfaces Comuns da API Java (cont.)

- **Serializable**
  - Utilizada para identificar classes cujos objetos podem ser escritos (serializados) ou lidos (desserializados) de algum tipo de armazenamento (arquivo, banco de dados), ou transmitidos através da rede.

Interfaces Comuns da API Java (cont.)

- **Runnable**
  - Implementada por qualquer classe cujos objetos devem estar aptos a executarem em paralelo utilizando uma técnica chamada de *multithreading*
  - Contém um método: *run()*
  - Descreve o comportamento do objeto quando executado.

Interfaces Comuns da API Java (cont.)

- **GUI event-listener interfaces**
  - Toda interação com uma interface gráfica é chamada de *evento*
    - O código utilizado para responder é chamado de manipulador de evento (*event handler*).
  - Os manipuladores de eventos são declarados em classes que implementam uma interface *event-listener* adequada

Interfaces Comuns da API Java (cont.)

- **Swing Constants**
  - Contém um conjunto de constantes utilizado em programação de interfaces gráficas
    - Posicionamento de elementos GUI na tela

FIM