

Universidade Federal da Paraíba

Centro de Informática

---

Departamento de Informática

# Linguagem de Programação I

## Herança e Composição

▶ Tiago Maritan

▶ [tiago@ci.ufpb.br](mailto:tiago@ci.ufpb.br)

---



# Motivação

---

- ▶ Uma característica marcante das linguagens OO é a capacidade de **reutilizar código**
  - ▶ Aproveitar classes e métodos que já estejam escritos e bem testados;
- ▶ Com isso é possível:
  - ▶ **Reduzir o trabalho do programador**: escrever menos códigos;
  - ▶ **Diminuir os erros**: reuso de classes e métodos já bem testados;



# Motivação

---

- ▶ Hoje veremos os mecanismos básicos de reuso em POO:
  - ▶ **Composição (ou delegação)**
  - ▶ **Herança**



# Composição ou delegação

---

- ▶ Criação de **novas classes** usando **instâncias de classes existentes**
  - ▶ Nova classe será **composta** por uma **instância da classe base**
- ▶ **Ex: Considere 2 classes: Data e Hora**
  - ▶ Com essas classes podemos criar uma nova classe DataHora que representa uma data e uma hora simultaneamente

```
public class DataHora {  
    private Data estaData; // instância da classe Data  
    private Hora estaHora; // instância da classe Hora  
  
    // declaração de métodos  
}
```

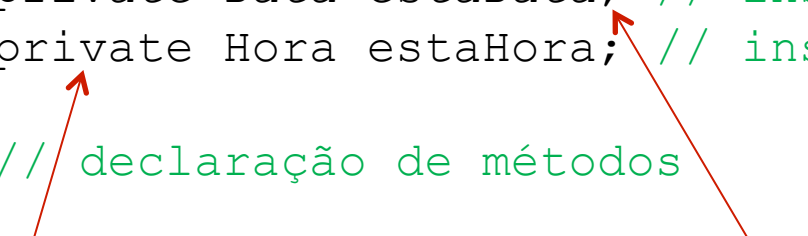


# Composição ou delegação

---

- ▶ Criação de **novas classes** usando **instâncias de classes existentes**
  - ▶ Nova classe será **composta** por uma **instância da classe base**
- ▶ **Ex: Considere 2 classes: Data e Hora**
  - ▶ Com essas classes podemos criar uma nova classe DataHora que representa uma data e uma hora simultaneamente

```
public class DataHora {  
    private Data estaData; // instância da classe Data  
    private Hora estaHora; // instância da classe Hora  
  
    // declaração de métodos  
}
```



**Classe DataHora composta por instâncias de Data e Hora**

---



# Composição ou delegação

---

- ▶ DataHora **pode reutilizar os métodos de** Data **e** Hora
  - ▶ Ex: Construtores **delegam** a outros inicialização dos campos

```
public class DataHora {  
    private Data estaData; // instância da classe Data  
    private Hora estaHora; // instância da classe Hora  
  
    public DataHora(byte hora, byte min, byte seg,  
        byte dia, byte mês, short ano) {  
        estaData = new Data(dia, mês, ano);  
        estaHora = new Hora(hora, min, seg);  
    }  
  
    public DataHora(byte dia, byte mês, short ano) {  
        estaData = new Data(dia, mês, ano);  
        estaHora = new Hora(0, 0, 0);  
    }  
}
```

# Composição ou delegação

```
public class Aluno {  
    private String nome;  
    private Data dataNascimento;  
    private int matricula;  
  
    Aluno(String n, Data d, int m) {  
        nome = n;  
        dataNascimento = d;  
        matricula = m;  
    }  
  
    public String toString() {  
        String res = " Matricula =" + matricula;  
        res += " Nome =" + nome;  
        res += " Data =" + dataNascimento;  
        return res;  
    }  
}
```

**Composto por uma  
instância de Data**



# Composição ou delegação

```
public class Aluno{  
    private String nome;  
    private Data dataNascimento;  
    private int matricula;
```

```
    Aluno(String n, Data d, int m){  
        nome = n;  
        dataNascimento = d;  
        matricula = m;  
    }
```


```
    public String toString(){  
        String res = " Matricula =" + matricula;  
        res += " Nome =" + nome;  
        res += " Data =" + dataNascimento;  
        return res;  
    }
```

```
}
```

**Chamada implícita ao método  
toString() de Data**



**Delega a classe Data a  
formatação de seus dados**





# Composição ou delegação

---

- ▶ Geralmente caracterizada por relações do tipo **“tem um”**
  - ▶ DataHora **“tem uma”** Data e **“tem uma”** Hora
- ▶ Vantagem do **reuso por composição**:
  - ▶ Uma nova classe DataHora foi criada **sem ser complexa**;
  - ▶ A complexidade (ex: verificar se a data e hora estão corretas, etc.) é implementada pelos métodos das classes Data e Hora



# Herança

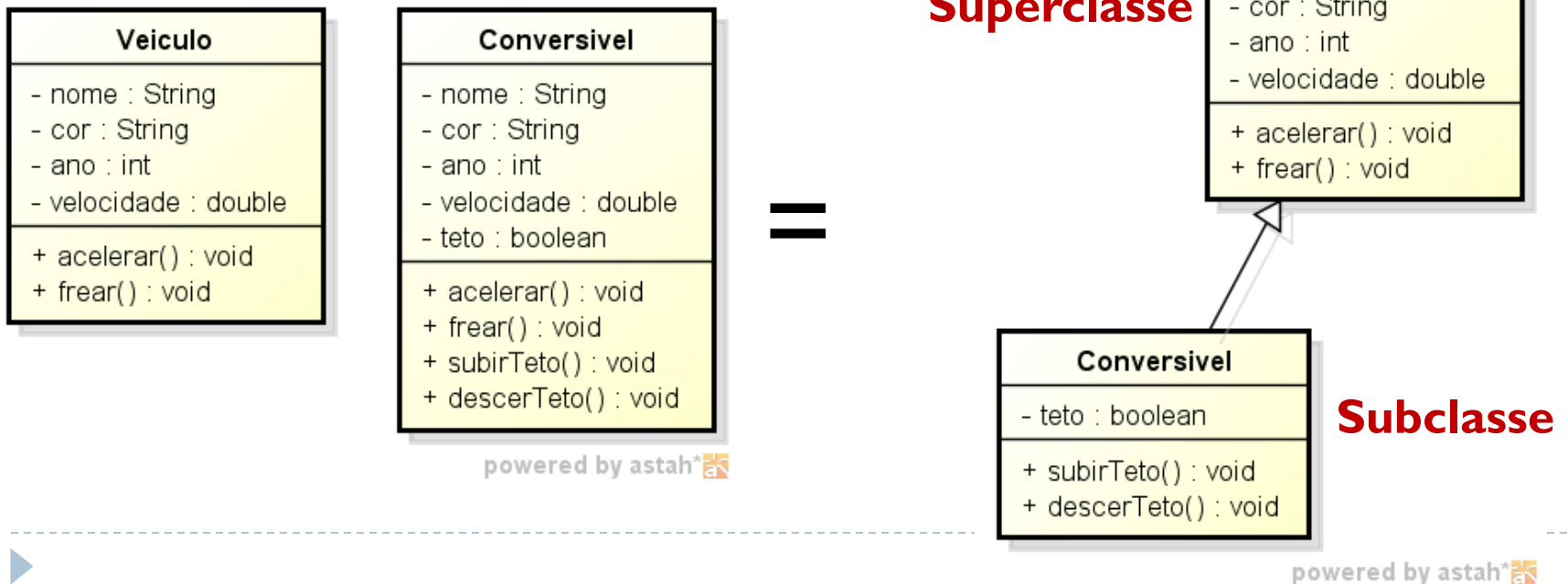
---

- ▶ Uma outra estratégia de reuso em POO
- ▶ **Novas classes** são criadas a partir das **classes existentes**
  - ▶ reutilizando seus **atributos** e **métodos não-privados** e
  - ▶ **adicionando novos recursos** que as novas classes exigem
- ▶ Exemplo:



# Herança

- ▶ Ex: A classe `Conversível` **pode herdar** da classe `Veiculo`
  - ▶ **Reutiliza atributos e métodos não-privados** de `Veiculo` e
  - ▶ Pode adicionar novos atributos e métodos: `teto`, `subirTeto()` e `descerTeto()`;



# Herança

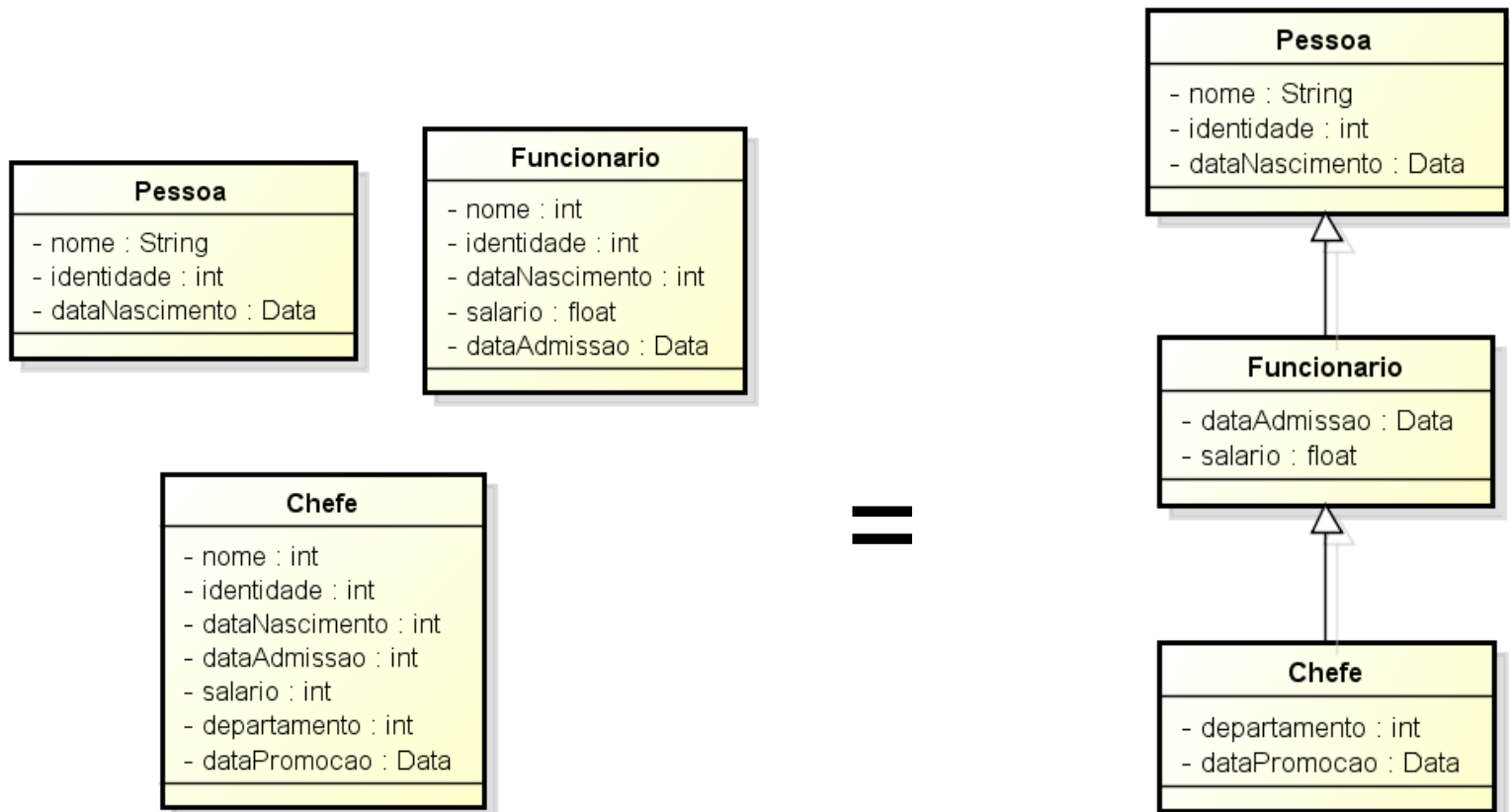
---

- ▶ Geralmente representam relações **“é um tipo de”**
  - ▶ Conversivel “é um tipo de” Veiculo
  - ▶ Funcionario “é um tipo de” Pessoa
  - ▶ Chefe “é um tipo de” Funcionario



# Herança

- **Ex2:** Ex: A classe `Funcionario` **pode herdar** da classe `Pessoa` e a classe `Chefe` **pode herdar** da classe `Funcionario`



# Herança


---

- ▶ Em Java, a palavra reservada **extends** denota herança
  - ▶ Indica que você está criando uma classe que deriva de uma classe existente

**Subclasse  
ou classe filha**

**Superclasse  
ou classe pai**

- ▶ Sintaxe:



```
public class Funcionario extends Pessoa
{
    // metodos e campos adicionais de Funcionario
}
```

```
public class Chefe extends Funcionario
{
    // metodos e campos adicionais de Chefe
}
```



# Herança

---

## ► Ex1:

```
public class Veiculo{
    protected String nome;
    protected String cor;
    protected int ano;
    protected double velocidade;

    public Veiculo(String n, String c, int a){
        nome = n; cor = c; ano = a;
        velocidade = 0.0;
    }

    public void acelerar(){
        velocidade++;
    }

    public void frear(){
        velocidade--;
    }
}
```

# Herança

---

## ► Ex1:

```
public class Conversivel extends Veiculo{
    private boolean teto;

    public Conversivel(String n, String c, int a){
        super(n, c, a); // veremos já
        teto = false;
    }

    public void subirTeto(){
        teto = true;
    }

    public void descerTeto(){
        teto = false;
    }
}
```



# Herança

---

## ► Ex1:

```
public class TesteVeiculoEConversivel {  
  
    public static void main(String args[]) {  
        Veiculo v1 = new Veiculo("Corsa", "Preto", 2010);  
        v1.acelerar(); // ok!  
        v1.frear();    // ok!  
  
        Conversivel c1 = new Conversivel("KA", "Preto", 2009);  
        c1.subirTeto(); // ok!  
        c1.descerTeto(); // ok!  
  
        c1.acelerar (); // e agora??  
        c1.frear();    // e agora??  
    }  
}
```



# Herança

---

## ► Ex2:

```
public class Pessoa {
    protected String nome;
    protected int identidade;
    protected Data nascimento;

    public Pessoa(String n, int i, Data d){
        nome = n;
        identidade = i;
        nascimento = d;
    }

    public String toString(){
        return "Nome:" + nome + "\nIdentidade:" +
            identidade + "\nNascimento:" + nascimento;
    }
}
```

# Herança

---

## ► Ex2:

```
public class Funcionario extends Pessoa {  
    protected Data admissao;  
    protected float salario;  
  
    public Funcionario(String nome, int id,  
        Data nasc, Data adm, float sal){  
  
        super(nome, id, nasc);  
        admissao = adm;  
        salario = sal;  
    }  
  
    public String toString(){  
        String res = super.toString() + "\n";  
        res += "Admissao:" + admissao;  
        res += "\nSalario" + salario;  
        return res;  
    }  
}
```

# Herança

## ► Ex2:

```
public class Chefe extends Funcionario {
    private String departamento;
    private Data promocao;

    public Chefe(String nome, int id,
                  Data nasc, Data adm, float sal,
                  String dep, Data prom){

        super(nome, id, nasc, adm, sal);
        departamento = dep;
        promocao = prom;
    }

    public String toString(){
        String res = super.toString() + "\n";
        res += "Departamento:" + departamento;
        res += "\nPromocao" + promocao;
        return res;
    }
}
```

# Herança

---

## ► Ex2:

```
public class TestaPessoas{
    public static void main(String args[]){
        Data d1 = new Data(01,11,1990);
        Data d2 = new Data(05,12,1980);
        Data d3 = new Data(01,05,1970);

        Pessoa p1 = new Pessoa("Tiago", 111111, d1);
        Funcionario f1 = new Funcionario("Mateus",
            111112, d1, d2, 1000.0);
        Chefe c1 = new Chefe("Rodrigo", 111113, d3, d2,
            1000.0, "DCE, d1);

        System.out.println(p1.toString());
        System.out.println(f1.toString());
        System.out.println(c1.toString());
    }
}
```

# Palavra reservada **super**

---

- ▶ Permite acessar métodos da superclasse
- ▶ Aumenta a **reutilização de código**
  - ▶ Se existem métodos na classe pai que podem efetuar parte do processamento, é melhor usar o código que já existe;

```
public class Funcionario extends Pessoa {  
    ...  
    public Funcionario(String nome, int id,  
        Data nasc, Data adm, float sal){  
        super(nome, id, nasc);  
    }  
  
    public String toString(){  
        String res = super.toString() + "\n";  
        ...  
    }  
}
```

# Palavra reservada **super**

---

- ▶ Permite acessar métodos da superclasse
- ▶ Aumenta a **reutilização de código**
  - ▶ Se existem métodos na classe pai que podem efetuar parte do processamento, é melhor usar o código que já existe;

```
public class Funcionario extends Pessoa {  
    ...  
    public Funcionario(String nome, int id,  
        Data nasc, Data adm, float sal){  
        super(nome, id, nasc) ;  
    }  
  
    public String toString(){  
        String res = super.toString() + "\n";  
        ...  
    }  
}
```

**Invoca construtor da  
classe pai Pessoa  
(Deve ser na 1ª linha)**

**Invoca método toString()  
da classe pai Pessoa**

# Palavra reservada **super**

---

## ► Algumas considerações:

1. Construtores da classe pai devem ser chamados na primeira linha...

**super (<parametros>)**

2. Métodos da classe pai podem ser chamados da seguinte forma:

**super . nomeMetodo (<parametros>)**





# Palavra reservada **super**

---

## ▶ Algumas considerações:

3. Apenas **métodos e construtores** da **superclasse imediata** podem ser invocados usando `super`
  - ▶ **Não existem construções do tipo “`super . super`”**
4. Quando programador não chama o construtor da superclasse, ele é chamado implicitamente.
  - ▶ Invoca implicitamente o construtor default
  - ▶ Se não existir construtor *default* => erro de compilação



# Object: A raiz da hierarquia

---

- ▶ Todas as classes herdam da classe Object, mesmo que não contenham a declaração de herança.
- ▶ Contém apenas métodos genéricos, que devem ser reimplementados pelas classes.

# Object: A raiz da hierarquia

Method Summary	
protected <u>O</u> <u>b</u> ject	<b><u>clone()</u></b> Creates and returns a copy of this object.
boolean	<b><u>equals()</u></b> (Object obj) Indicates whether some other object is "equal to" this one.
protected v oid	<b><u>finalize()</u></b> Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
<u>Class</u>	<b><u>getClass()</u></b> Returns the runtime class of an object.
int	<b><u>hashCode()</u></b> Returns a hash code value for the object.
void	<b><u>notify()</u></b> Wakes up a single thread that is waiting on this object's monitor.
void	<b><u>notifyAll()</u></b> Wakes up all threads that are waiting on this object's monitor.
<u>String</u>	<b><u>toString()</u></b> Returns a string representation of the object.
void	<b><u>wait()</u></b> Causes current thread to wait until another thread invokes the <u>notify()</u> method or the <u>notifyAll()</u> method for this object.
void	<b><u>wait()</u></b> (long timeout) Causes current thread to wait until either another thread invokes the <u>notify()</u> method or the <u>notifyAll()</u> method for this object, or a specified amount of time has elapsed.
void	<b><u>wait()</u></b> (long timeout, int nanos) Causes current thread to wait until another thread invokes the <u>notify()</u> method or the <u>notifyAll()</u> method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

# Hierarquia de classes em Java

---

## Class Hierarchy

```
oclass java.lang.Object
  oclass java.lang.Boolean (implements java.io.Serializable)
  oclass java.lang.Character (implements java.lang.Comparable, java.io.Serializable)
  oclass java.lang.Character.Subset
    oclass java.lang.Character.UnicodeBlock
  oclass java.lang.Class (implements java.io.Serializable)

  . . .
  oclass java.lang.Math
  oclass java.lang.Number (implements java.io.Serializable)
    oclass java.lang.Byte (implements java.lang.Comparable)
    oclass java.lang.Double (implements java.lang.Comparable)
    oclass java.lang.Float (implements java.lang.Comparable)
    oclass java.lang.Integer (implements java.lang.Comparable)
    oclass java.lang.Long (implements java.lang.Comparable)
    oclass java.lang.Short (implements java.lang.Comparable)

    . . .
```

# Sobrescrita (ou sobreposição)

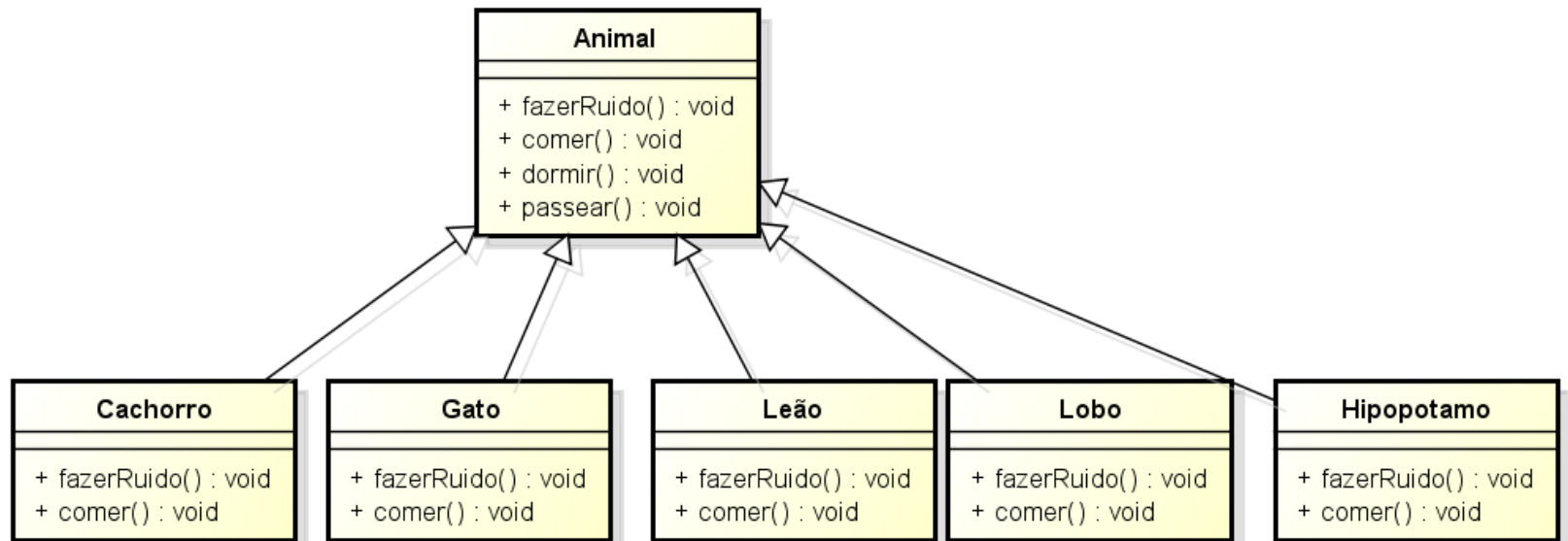
---

- ▶ Em POO, uma subclasse pode redefinir um atributo ou um método da superclasse com a mesma assinatura.
- ▶ Sobrescrita de atributos:
  - ▶ Um campo declarado na subclasse oculta o campo de mesmo nome da superclasse. Não é muito útil.
- ▶ Sobreposição de métodos:
  - ▶ Um método declarado na subclasse oculta o método com a mesma assinatura declarado na superclasse.
  - ▶ O método oculto da superclasse, se não for privado, pode ser invocado através da palavra `super`.
  - ▶ Ex: método `toString()` nas classes `Funcionario` e `Chefe`;

# Sobreposição

## ► Exemplo:

- Cada tipo de Animal faz um ruído diferente e come algo diferente.



# Sobrescrita de métodos

---

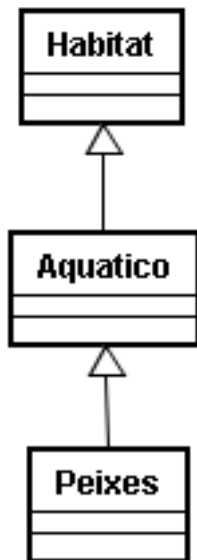
## ► Considerações:

1. Um método **public** (subclasse) pode sobrepor um método `private` (superclasse);
2. Um método **private** (subclasse) **não** pode sobrepor um método **public** (superclasse);
3. Um método estático não pode ser sobreposto;
4. Um método **final** é herdado pelas subclasses, mas não pode ser sobreposto.

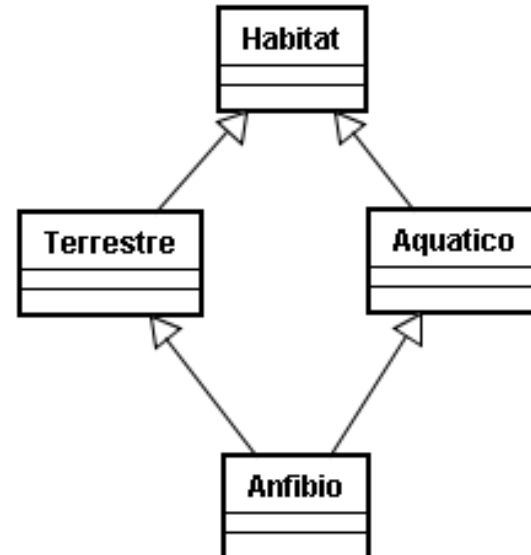
# Herança simples vs Herança múltipla

- ▶ **Herança simples:** classe é derivada de uma única superclasse.
- ▶ **Herança múltipla:** classe é derivada de mais de uma superclasse.
- ▶ **Java possui suporte apenas a herança simples**
  - ▶ É possível implementá-la através do uso de **interfaces**;

## Herança Simples



## Herança Múltipla





# Herança vs Composição

---

## ► Composição: **tem-um**

- Quando se quer as características de uma classe, mas não seus campos e métodos;
- O *componente* auxilia na implementação da funcionalidade da nova classe.

## ► Herança: **é-um**

- Além de definir seus próprios atributos e métodos, a subclasse também herda atributos e/ou métodos da superclasse.

Universidade Federal da Paraíba

Centro de Informática

---

Departamento de Informática

# Linguagem de Programação I

## Herança e Composição

▶ Tiago Maritan

▶ [tiago@ci.ufpb.br](mailto:tiago@ci.ufpb.br)

---

