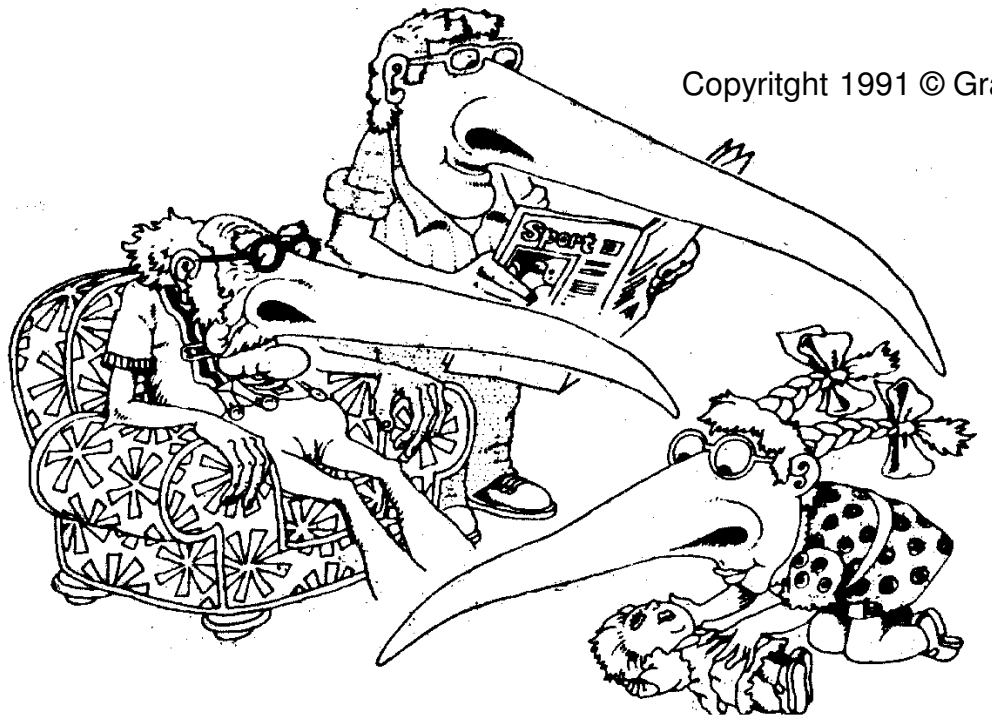
INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
PARAÍBA
Campus João Pessoa

Programação Orientada a Objetos

Fausto Maranhão Ayres

7
Herança

Herança



O que é Herança?

- É a “transmissão” de **atributos e métodos compilados** de uma (**super**)classe para uma (**sub**)classe, formando uma **hierarquia**

Exemplo de Hierarquia JFrame

java.lang.Object
java.awt.Component
java.awt.Container
java.awt.Window
java.awt.Frame
javax.swing.JFrame

JFrame recebe atributos e métodos das **5 classes** ancestrais, os quais são transmitidos de cada superclasse para cada subclasse da hierarquia

Métodos herdados por JFrame

https://docs.oracle.com/en/java/javase/17/docs/api/java.desktop/javax/swing/JFrame.html

Methods declared in class java.awt.Frame

addNotify, getCursorType, getExtendedState, getFrame
setExtendedState, setMaximizedBounds, setMenuBar, se

Methods declared in class java.awt.Window

addPropertyChangeListener, addPropertyChangeListener
createBufferStrategy, dispose, getBackground, getBuf
getListeners, getLocale, getModalExclusionType, getM
getWindowFocusListeners, getWindowListeners, getWind
isFocusCycleRoot, isFocused, isLocationByPlatform, i
removeWindowFocusListener, removeWindowListener, rem
setFocusableWindowState, setFocusCycleRoot, setIconI
setOpacity, setShape, setSize, setSize, setType, set

Methods declared in class java.awt.Container

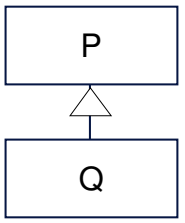
add, add, add, add, add, addContainerListener, apply
getAlignmentY, getComponent, getComponentAt, getComp
getMaximumSize, getMinimumSize, getMousePosition, ge
list, list, locate, minimumSize, paintComponents, pr
setFocusTraversalKeys, setFocusTraversalPolicy, setF

Methods declared in class java.awt.Component

action, add, addComponentListener, addFocusListener,
addMouseWheelListener, bounds, checkImage, checkImag
dispatchEvent, enable, enable, enableEvents, enableI
firePropertyChange, firePropertyChange, firePropertY
getCursor, getDropTarget, getFocusListeners, getFocu
getHierarchyListeners, getIgnoreRepaint, getInputMet

java.lang.Object
java.awt.Component
java.awt.Container
java.awt.Window
java.awt.Frame
javax.swing.JFrame

Declaração de Herança



superclasse

subclasse

Diagrama UML

```
public class Q extends P {...}
```

A **subclasse Q** recebe todos os atributos e métodos compilados de **Superclasse P** (exceto o construtor)

- **Java** só permite herança **simples** de uma única superclasse
- Outras LP permitem herança **múltipla**

Justificativa para Herança

- Qual a **vantagem** de uma classe herdar atributos e métodos de outra classe?
  - Facilidade de manutenção - não há necessidade de cópia de código fonte em várias classes
  - Usufruir da tecnologia de **Polimorfismo**

EXEMPLO 1 – BÁSICO

Duas classes *sem* Herança

- A classe Q possui muita semelhança com P

```java
public class P {
    public int a;
    public int b;
    public int c;

    public P(int a, int b, int c) {

        this.a = a;
        this.b = b;
        this.c = c;
    }
    public int somar() {
        return a+b+c;
    }
}
```

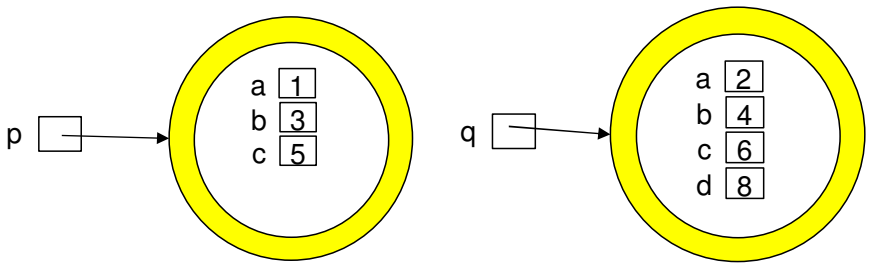
```java
public class Q {
    public int a;
    public int b;
    public int c;
    public int d;

    public Q(int a, int b, int c, int d) {

        this.a = a;
        this.b = b;
        this.c = c;
        this.d = d;
    }
    public int somar() {
        return a+b+c+d;
    }
}
```

semelhança

semelhança

```java
public class Teste {
    public static void main(String[] args) {
        P p = new P(1,3,5);
        Q q = new Q(2,4,6,8);
        System.out.println(p.somar());  //9
        System.out.println(q.somar());  //20
        System.out.println(p.a);        //1
        System.out.println(q.a);        //2
    }
```

Objetos instanciados de P e Q




```
public class Teste {
    public static void main(String[] args) {
        P p = new P(1,3,5);
        Q q = new Q(2,4,6,8);
        System.out.println(p.somar());  //9
        System.out.println(q.somar());  //20
        System.out.println(p.a);        //1
        System.out.println(q.a);        //2
    }
```


Usando Herança

declaração

Q recebe automaticamente atributos e métodos de P

```
public class P {
    public int a;
    public int b;
    public int c;

    public P(int a, int b, int c) {
        super();
        this.a = a;
        this.b = b;
        this.c = c;
    }
    public int somar() {
        return a+b+c;
    }
}
```

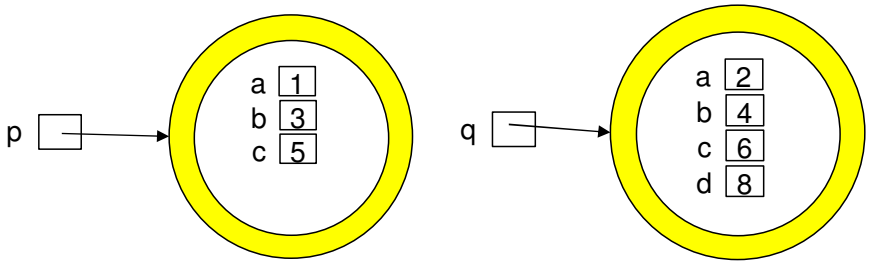
```
public class Q extends P {
    public int d;

    public Q(int a, int b, int c, int d) {
        super(a, b, c);
        this.d = d;
    }
    public int somar() {
        return a + b + c + d;
    }
}
```

Sobrescrita de método

```
public class Teste {
    public static void main(String[] args) {
        P p = new P(1,3,5);
        Q q = new Q(2,4,6,8);
        System.out.println(p.somar());  //9
        System.out.println(q.somar());  //20
        System.out.println(p.a);        //1
        System.out.println(q.a);        //2
    }
```

Objetos instanciados de P e Q

A herança faz o *reuso de código compilado*
Isso facilita a manutenção – cada classe cuida apenas do que é seu !




```
public class Teste {
    public static void main(String[] args) {
        P p = new P(1,3,5);
        Q q = new Q(2,4,6,8);
        System.out.println(p.somar());   //9
        System.out.println(q.somar());   //20
        System.out.println(p.a);         //1
        System.out.println(q.a);         //2
    }
```

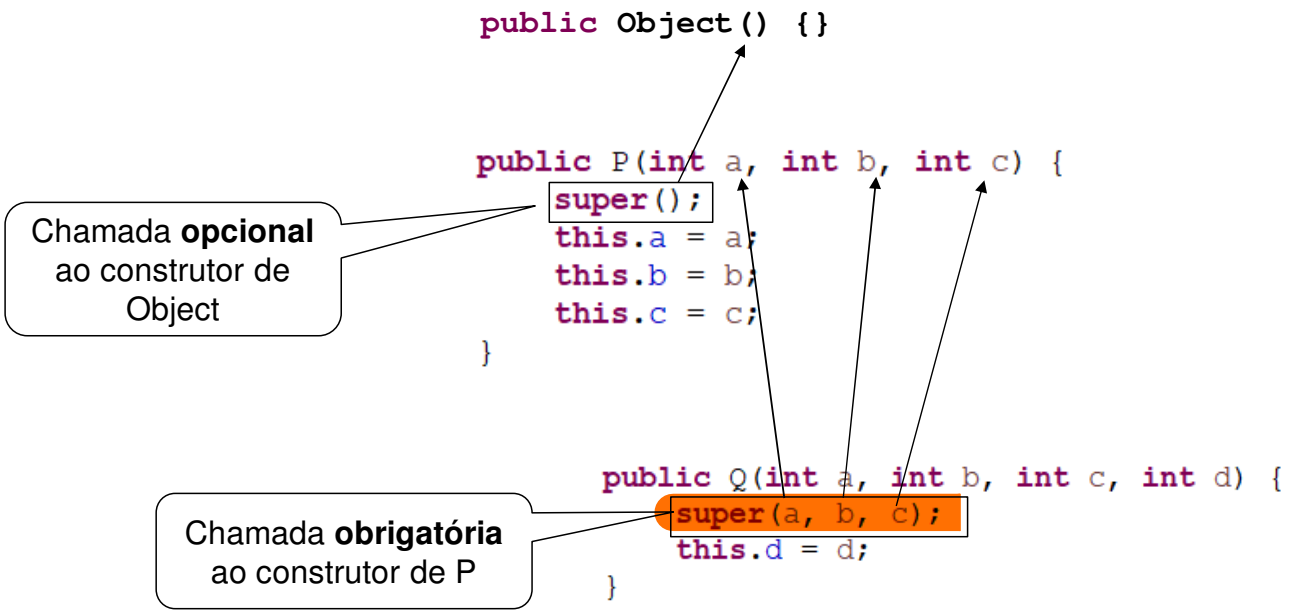
Observa-se que **a herança não altera o resultado do programa**

Conceitos que surgem com a Herança

- Chamada entre construtores
- Sobrescrita de método
- Polimorfismo de método
- Chamada de método da superclasse
- Atributos privados x protegidos
- Polimorfismo de variável
- Classe e método abstrato
- Listas polimórficas
- Casting de objetos

Chamada entre construtores

- O **construtor de cada subclasse,** no seu início, deve chamar o **construtor da superclasse,** usando o método **super(...)**



Sobrescrita (*override*) de método

- Ocorre quando um método é implementado na subclasse com a **mesma assinatura** da superclasse

Ex: existirão 2 versões de somar(), sendo que a segunda versão é a **sobrescrita** da primeira

Classe P

```
public int somar() {
    return a+b+c;
}
```

Classe Q

```
public int somar() {
    return a + b + c + d;
}
```

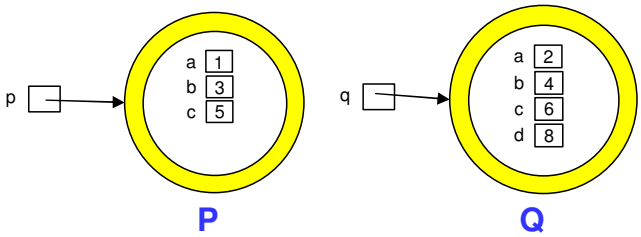
Versão sobrescrita

Polimorfismo de método

- É o **processo de descoberta** da versão correta do método durante a execução

```
public class Teste {
    public static void main(String[] args) {
        P p = new P(1,3,5);
        Q q = new Q(2,4,6,8);
        System.out.println(p.somar());   //9
        System.out.println(q.somar());   //20
```

Qual versão executar?



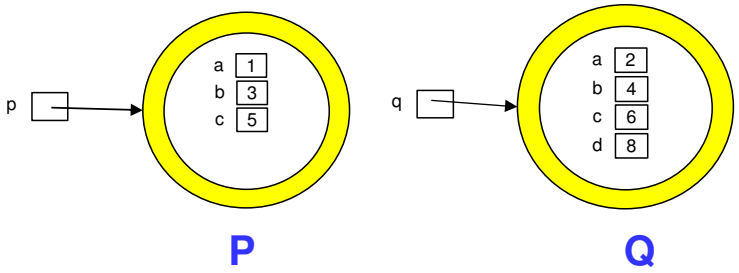

```
public int somar() {
    return a+b+c;
}
```

```
public int somar() {
    return a + b + c + d;
}
```


Polimorfismo de método

- A descoberta se baseia no **tipo do objeto** envolvido na chamada do método.

```
System.out.println(p.somar());   //9   somar() de P
System.out.println(q.somar());   //20  somar() de Q
```



Obs:
Nas LPOOs, o processo de descoberta é feito por **Ligação DINÂMICA** (**late binding**) por um ligador dinâmico

Nas LP antigas, o processo de descoberta é feito por **Ligação ESTÁTICA** *(***early binding***)* pelo compilador

Chamada a método da superclasse

Observe o cálculo da soma nas duas classes

```
public class P {
    public int a;
    public int b;
    public int c;

    public P(int a, int b, int c) {
        super();
        this.a = a;
        this.b = b;
        this.c = c;
    }
    public int somar() {
        return a+b+c;
    }
}
```

```
public class Q extends P {
    public int d;

    public Q(int a, int b, int c, int d) {
        super(a, b, c);
        this.d = d;
    }
    public int somar() {
        return a + b + c + d;
    }
}
```

Este cálculo já está pronto na classe P.
Como chama-lo ?

Chamada a método da superclasse

- Utiliza-se a variável padrão **super** para chamar um método da superclasse

Superclasse P

```
public int somar() {
    return a+b+c;
}
```

Subclasse Q

```
public int somar() {
    return super.somar() + d;
}
```

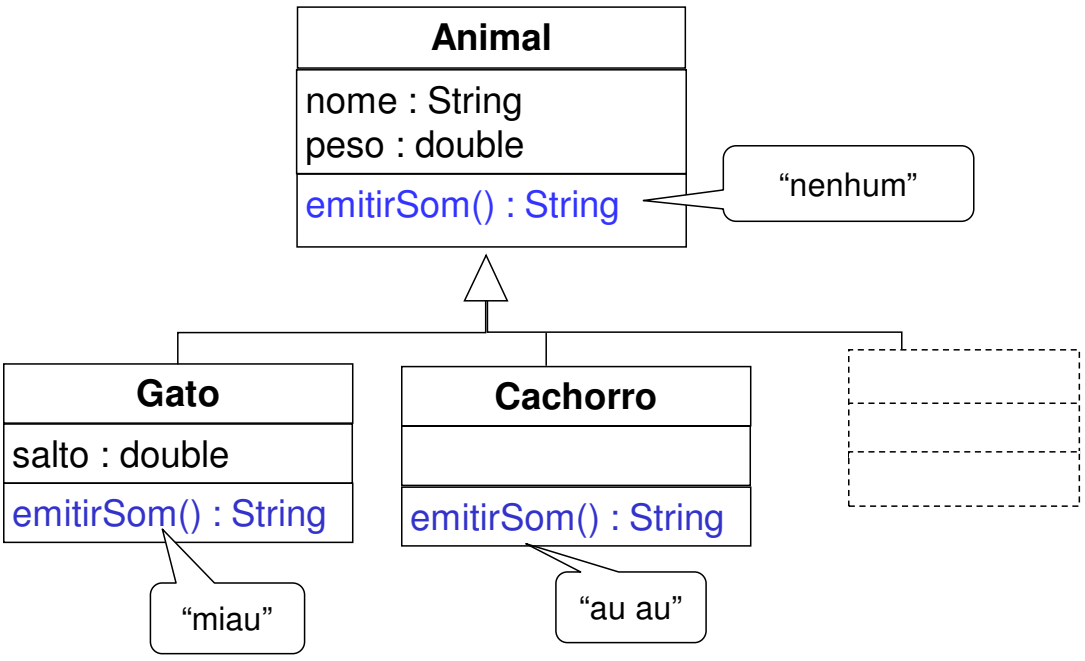
Equivale a:

```
return a + b + c + d;
```

Cuidado: se tirar a variável **super** haverá uma recursão

EXEMPLO 2
HIERARQUIA ANIMAL

Hierarquia Animal



Classe Animal

```
public class Animal {
    private String nome;
    private double peso;

    public Animal(String nome, double peso) {
        super();
        this.nome = nome;
        this.peso = peso;
    }
    public String emitirSom() {
        return "nenhum som";
    }
    public String toString() {
        return  " nome= " + nome +
                ", peso= " + peso +
                ", som= "+  emitirSom();
    }
```


Classe Cachorro

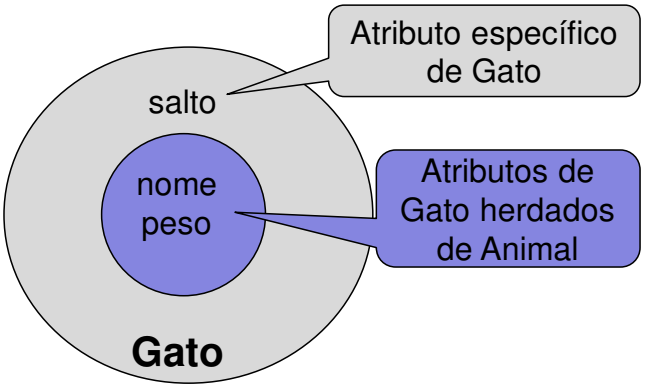
```
public class Cachorro extends Animal {
    public Cachorro(String nome, double peso) {
        super(nome, peso);
    }

    @Override
    public String emitirSom(){
        return "au au";
    }
}
```

Sobrescrita de emitirSom()

Classe Gato

```
public class Gato extends Animal {
    private int salto;

    public Gato(String nome, double peso, int salto) {
        super(nome, peso);
        this.salto = salto;
    }
    @Override
    public String emitirSom() {
        return  "miau!";
    }

    public int getSalto() {
        return salto;
    }
```

Sobrescrita de emitirSom()



@Override

- Faz com que o compilador valide a sobrescrita

Animal:

```
public String emitirSom() {
    return "nenhum som";
}
```

Assinatura original

Cachorro:

```
@Override
public String emitirSom(){
    return "au au";
}
```

válido

```
@Override
public String emitirsom(){
    return "au au";
}
```

Não ok

Teste

```
//instanciar os objetos
Gato fifi =       new Gato("fifi", 5, 3);
Cachorro rex =    new Cachorro("rex", 15);

//exibir nome, peso e som dos objetos com toString()
System.out.println(rex);              //"rex 15 au au"
System.out.println(fifi);             //"fifi 5 miau"
```


Exercício

- Aumente a hierarquia Animal com nova classe de sua escolha
- Criar um objeto da nova classe
- Exibir o objeto

Polimorfismo de emitirSom()

Teste

```
//instanciar os objetos
Gato fifi =       new Gato("fifi", 5, 3);
Cachorro rex =    new Cachorro("rex", 15);

//exibir nome, peso e som dos objetos com toString()
System.out.println(rex);              //"rex 15 au au"
System.out.println(fifi);             //"fifi 5 miau"
```

Observe que os prints usam o mesmo toString() herdado de Animal por Gato e Cachorro

classe Animal

```
public String toString() {
    return  " nome= " + nome +
            ", peso= " + peso +
            ", som= "+  emitirSom();
}
```

Qual versão executar? Existem 3 versões de emitirSom()

Como é o polimorfismo de emitirSom()?

```
Cachorro rex = new Cachorro("rex", 15);
System.out.println(rex);                    //"rex 15 au au"
```

classe Animal

```
public String toString() {
    return  " nome= " + nome +
            ", peso= " + peso +
            ", som= "+ emitirSom();
}
```

O tipo do objeto envolvido é Cachorro
Logo, chama emitirSom() de Cachorro

```
public class Cachorro extends Animal {
   public String emitirSom()  {
      return "au au";
   }
}
```


Como é o polimorfismo de emitirSom()?

```
Gato fifi = new Gato("fifi", 5, 3);
System.out.println(fifi);                   //"fifi 5 miau"
```

```
public String toString() {
    return  " nome= " + nome +
            ", peso= " + peso +
            ", som= "+ emitirSom();
}
```

O tipo do objeto envolvido é Gato
Logo, chama emitirSom() de Gato

```
public class Gato extends Animal {
   public String emitirSom()  {
      return "miauu";
   }
}
```

Atributos protegidos

Sobrescrita de toString() em Gato (1)

- Sobrescrita de toString() na classe Gato com acesso aos atributos **privativos** de Animal

```
@Override
public String toString() {
    return "nome=" + getNome() +
           ", peso=" + getPeso() +
           ", salto=" + salto +
           ", emitirSom()=" + emitirSom() ;
}
```

Observe que nome e peso são ***private*** na classe Animal

```
public class Animal {
    private String nome;
    private double peso;
```

Sobrescrita de toString() em Gato (2)

- Sobrescrita de toString() na classe Gato com acesso aos atributos **protegidos** de Animal

```
@Override
public String toString() {
    return "nome=" + nome +
           ", peso=" + peso +
           ", salto=" + salto +
           ", emitirSom()=" + emitirSom() ;
}
```

Considere agora que nome e peso são ***protected*** na classe Animal

```
public class Animal {
    protected String nome;
    protected double peso;
```


Sobrescrita de toString() em Gato (3)

- Sobrescrita de toString() na classe Gato com chamada **super**.toString()

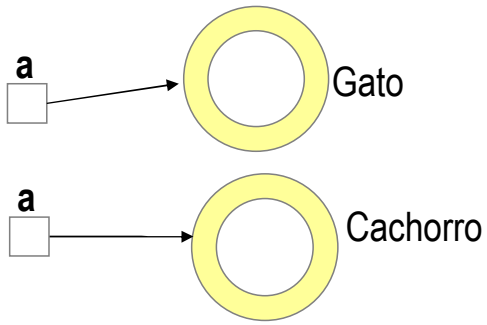
```
@Override
public String toString() {
    return  super.toString() + ",  salto=" + salto ;
}
```

Cuidado: se tirar a variável **super** haverá uma recursão

POLIMORFISMO DE VARIÁVEL (VARIÁVEL POLIMÓRFICA)

Variável polimórfica

- Uma variável do **tipo** T pode referenciar um objeto de um **subtipo** de T




```
Animal a;  //variavel polimorfica
a = new Gato(...);
System.out.println(a.emitirSom());   //miauu
a = new Cachorro(...);
System.out.println(a.emitirSom());   //auau
```

Exemplo

- Um veterinário aplica injeção em qualquer tipo de animal

Variável polimórfica no parâmetro
Pode receber qualquer tipo de animal

```
public class Veterinario {
   private String nome;

   public String aplicarInjecao(Animal  a) {
      return
            " Dr." + nome +
            " aplicou injeção em " + a.getNome() +
            " que fez " + a.emitirSom() ;
   }
}
```


Teste

- Aplicar injeção nos animais

```
Gato fifi = new Gato("fifi", 5, 3);
Cachorro rex = new Cachorro("rex", 15);
Veterinario bob = new Veterinario("Bob");

System.out.println( bob.aplicarInjecao(fifi) ) ;
System.out.println( bob.aplicarInjecao(rex) ) ;
```

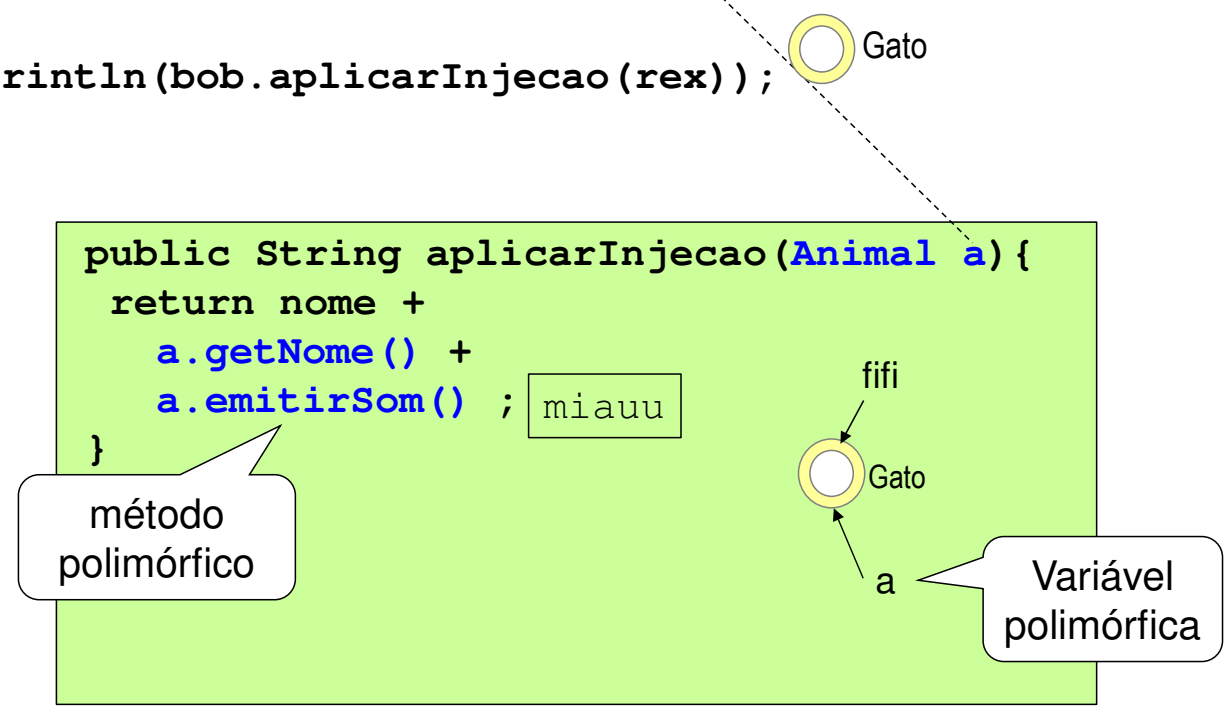
Saida:

```
Dr. Bob aplicou injeção em fifi que fez miau
Dr. Bob aplicou injeção em rex que fez au au
```

Como é o polimorfismo de variável

```
...
println(bob.aplicarInjecao(fifi));

println(bob.aplicarInjecao(rex));
```

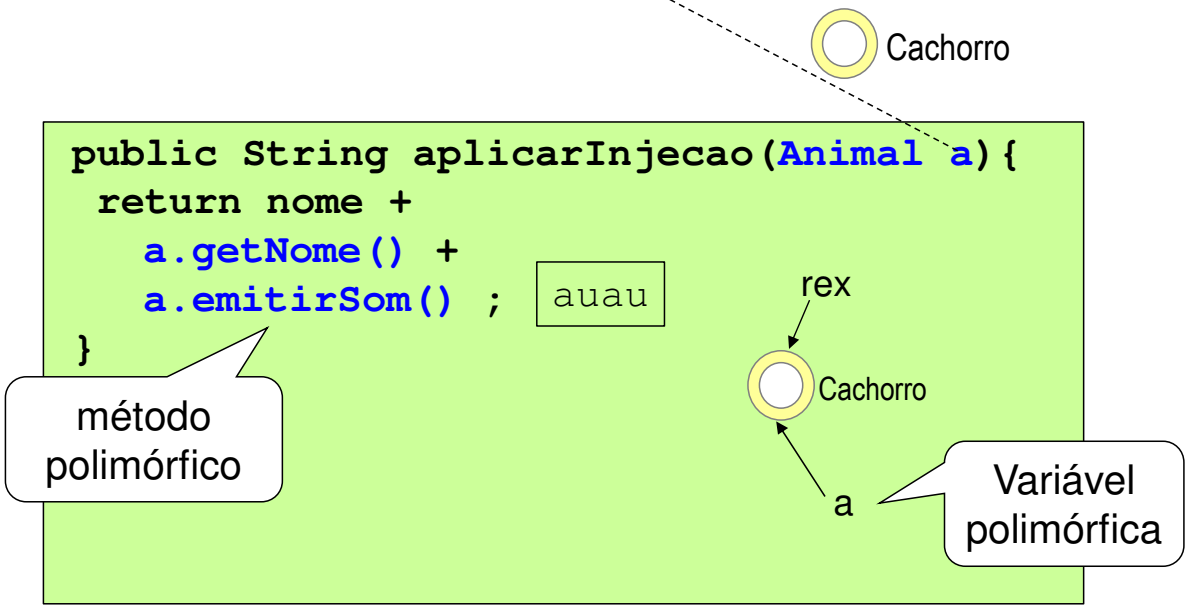
Chamada 1



Como é o polimorfismo de variável

```
...
println(bob.aplicarInjecao(g1)) ;

println(bob.aplicarInjecao(rex)) ;
```

Chamada 2



CLASSE E MÉTODO ABSTRATO

O que é classe Abstrata ?

- É uma classe que não pode ser instanciada, ou seja, não pode instanciar objetos

```
public abstract class Animal {
    private String nome;
    private double peso;

    public Animal(String nome, double peso) {
        super();
        this.nome = nome;
        this.peso = peso;
    }
```

O construtor é necessário, pois será chamado pelos construtores das subclasses

```
Animal a = new Animal(...);    //erro
```

Classe Abstrata como um Tipo

- O nome da classe abstrata pode ser usada como **Tipo** na declaração de variáveis, parâmetros, métodos, etc:

```
public String aplicarInjecao(Animal a) {
```

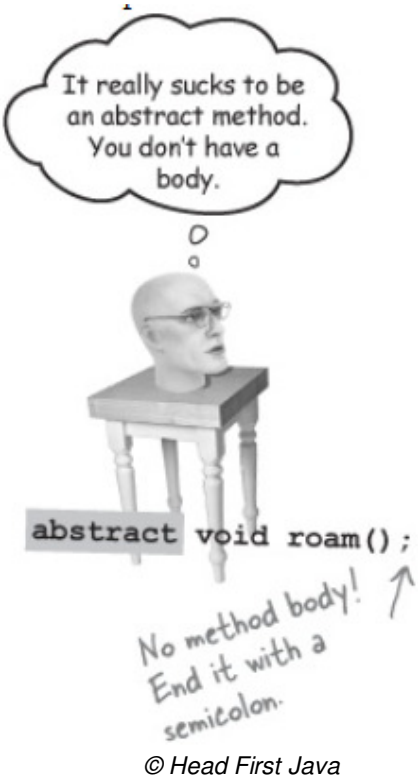

Método Abstrato

- É um método ***sem corpo***, apenas com assinatura
- Ele obriga a classe a ser abstrata

Ex

```
public abstract String emitirSom();
```

Este método não tem corpo



© Head First Java

Chamada a método Abstrato

- A chamada a um método abstrato é aceita pelo compilador, mas nunca será executada.

```
public abstract class Animal {
   private String nome;
   private double peso;
   ...
   public abstract String emitirSom();

   public String toString() {
      return nome + peso + emitirSom();
   }
}
```

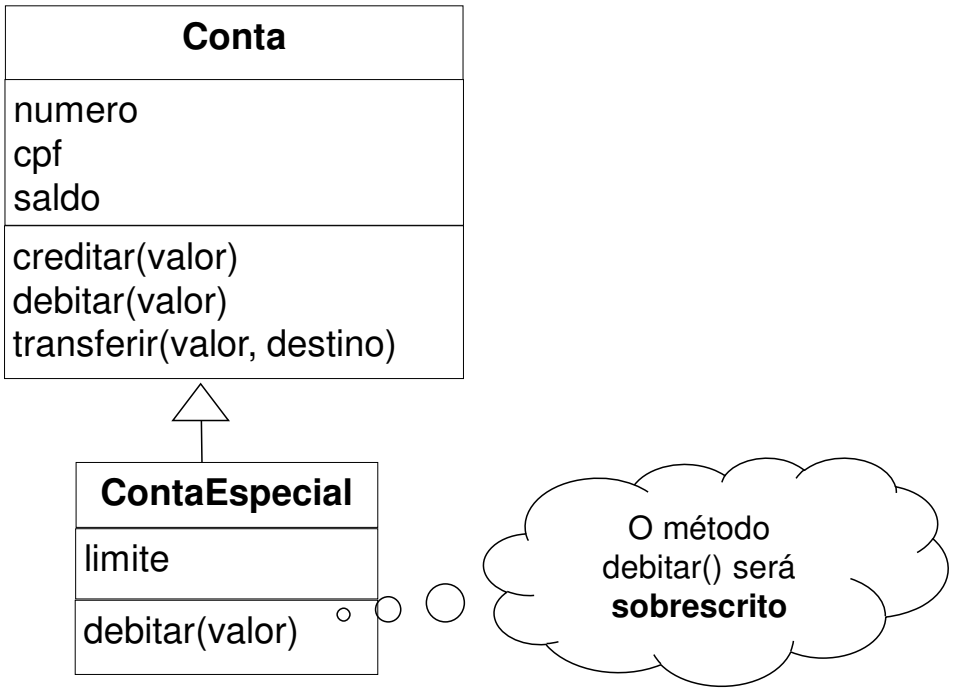
Compila normalmente

O polimorfismo de emitirSom() buscará executar uma de suas sobrescritas

EXERCÍCIOS

Exercício 1

- Implementar a classe ContaEspecial



Teste

- Transferência entre contas simples e especial

```
Conta conta1 = new Conta(1,"111");       //numero, cpf
ContaEspecial conta2 = new ContaEspecial(2, "222", 500.0);

try{
       conta1.creditar(500.0);
       conta1.transferir(500.0, conta2);        //saldo 0
       conta1.transferir(1.0, conta2);          //exceção
}catch(Exception e) {System.out.println(e.getMessage()); }

try{
       conta2.transferir(1000.0, conta1);       //saldo -500
       conta2.transferir(2.0, conta1);          //exceção
}catch(Exception e) {System.out.println(e.getMessage()); }

System.out.println(conta1);        //saldo 1000.0
System.out.println(conta2);        //saldo -500.0
```

Exercício 2

Considere a hierarquia de classes abaixo:

```
public class Pai {
   public void quemsoueu () {System.out.println( "sou pai" );}
}
public class Filho extends Pai {
   public void quemsoueu () {System.out.println( "sou filho" ); }
}
public class Neto extends Filho {
   public void quemsoueu () {System.out.println( "sou neto" ); }
}
```

Mostre os dois resultados exibidos:

```
Pai p = new Filho();
Filho f = new Neto();
p.quemsoueu();
f.quemsoueu();
```


Exercício 3

- Execute o teste abaixo:

  A a = new A();

  a.escreva();

  a.escreva(5);

```
class A {
   public void escreva(int n)     { System.out.println(n+2); }
   public void escreva()           { escreva(3); }
}
class B extends A {
   public void escreva(int n)     { super.escreva(n+1); }
   public void escreva()           { super.escreva(); }
}
```

Cont

- Execute o teste abaixo:

  B b = new B();
  b.escreva();
  b.escreva(5);

```
class A {
   public void escreva(int n)     { System.out.println(n+2); }
   public void escreva()          { escreva(3); }
}
class B extends A {
   public void escreva(int n)     { super.escreva(n+1); }
   public void escreva()          { super.escreva(); }
}
```