



Polimorfismo

Prof^a. Rachel Reis
rachel@inf.ufpr.br

Motivação

e-commerce



Objeto da classe
Computador



Objeto da classe
Celular



Objeto da
classe Televisao



Estrutura de dados:
Array



Motivação

```
public class Loja {  
    public void vender(Computador comp) {...}  
    public void vender(Celular cel) {...}  
    public void vender(Televisao tv) {...}  
    ...  
    public void vender(Outros produtos??) {...}  
}
```

Será que ter um método vender para cada item da loja
é a melhor alternativa?



Solução: Polimorfismo

- Geral x Específico.
- Relaciona-se com herança, interfaces e classes abstratas.
- Sistemas extensíveis, com facilidade na adição de novas classes.



O que é polimorfismo?

- Polimorfismo significa que diferentes tipos de objetos podem responder a uma mesma mensagem de diferentes maneiras.
- Polimorfismo é um termo que se originou do grego e significa “várias formas”, ou seja, que uma determinada “coisa” pode ser feita de várias maneiras diferentes.



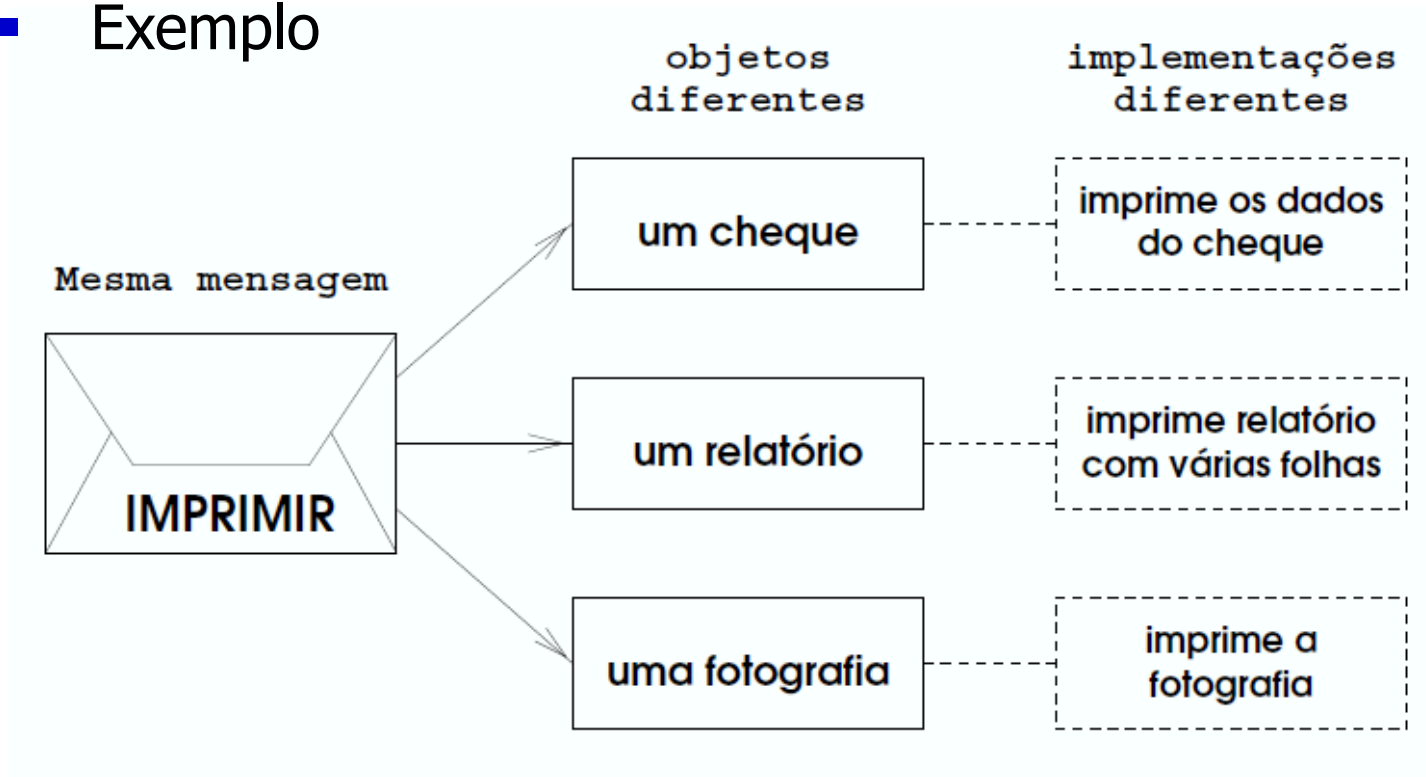
O que é polimorfismo?

- Uma analogia ao polimorfismo é o sinal dos colégios. Embora seja um único sinal, ele significa coisas diferentes para cada estudante:
 - ✓ uns vão para casa
 - ✓ outros para a biblioteca
 - ✓ outros para o barzinho
 - ✓ alguns voltam para a sala de aula

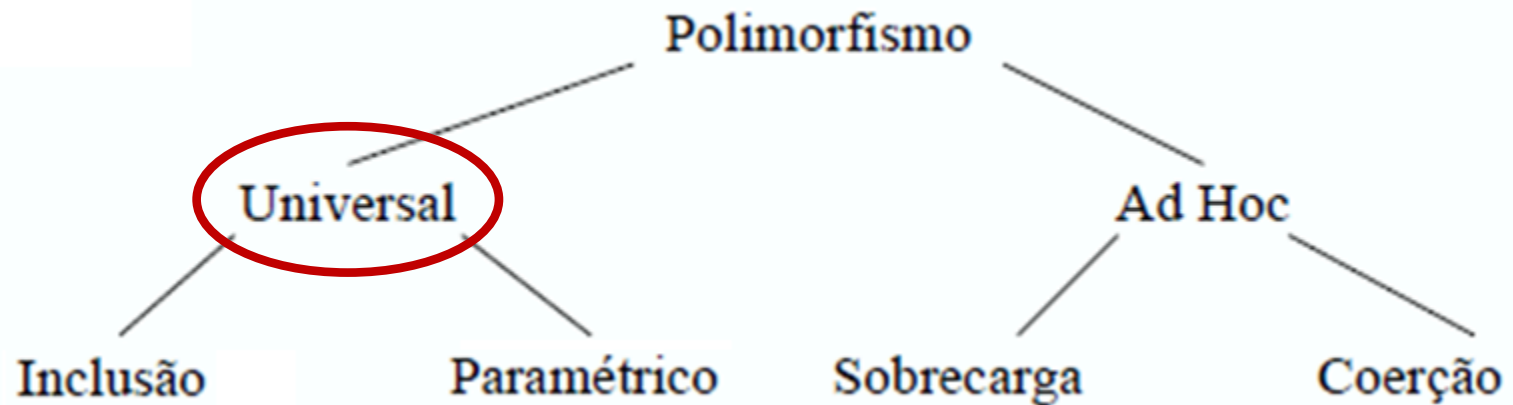
**Todos respondem
ao sinal, mas cada
um do seu jeito**

O que é polimorfismo?

- Exemplo



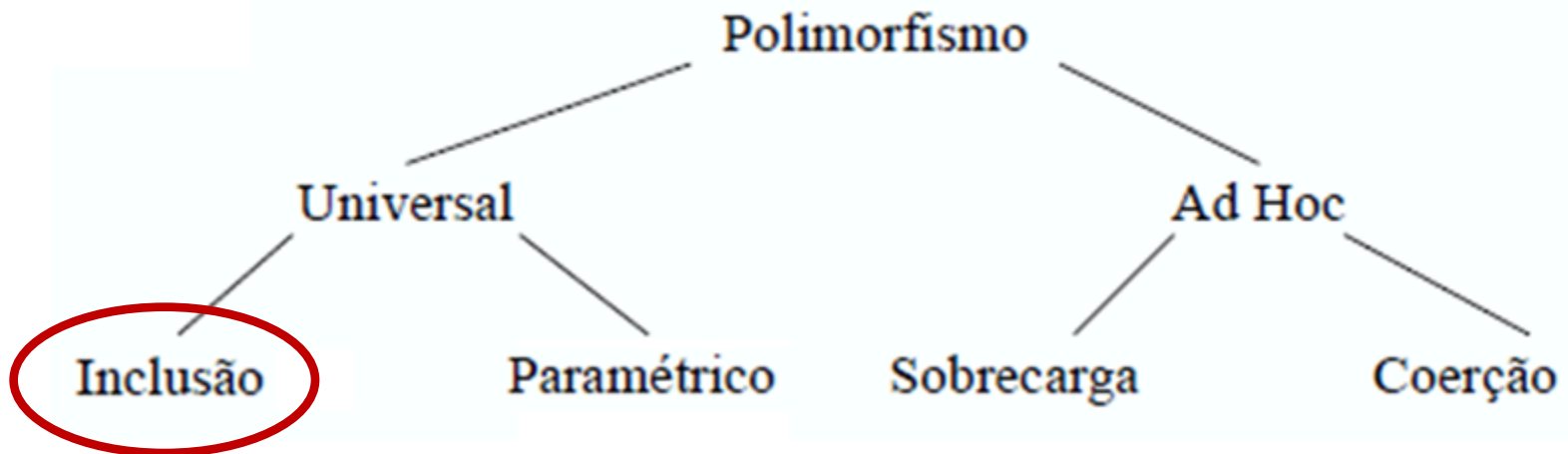
Classificação Polimorfismo



- Universal: ocorre em tempo de execução e trabalha em um conjunto infinito de tipos.

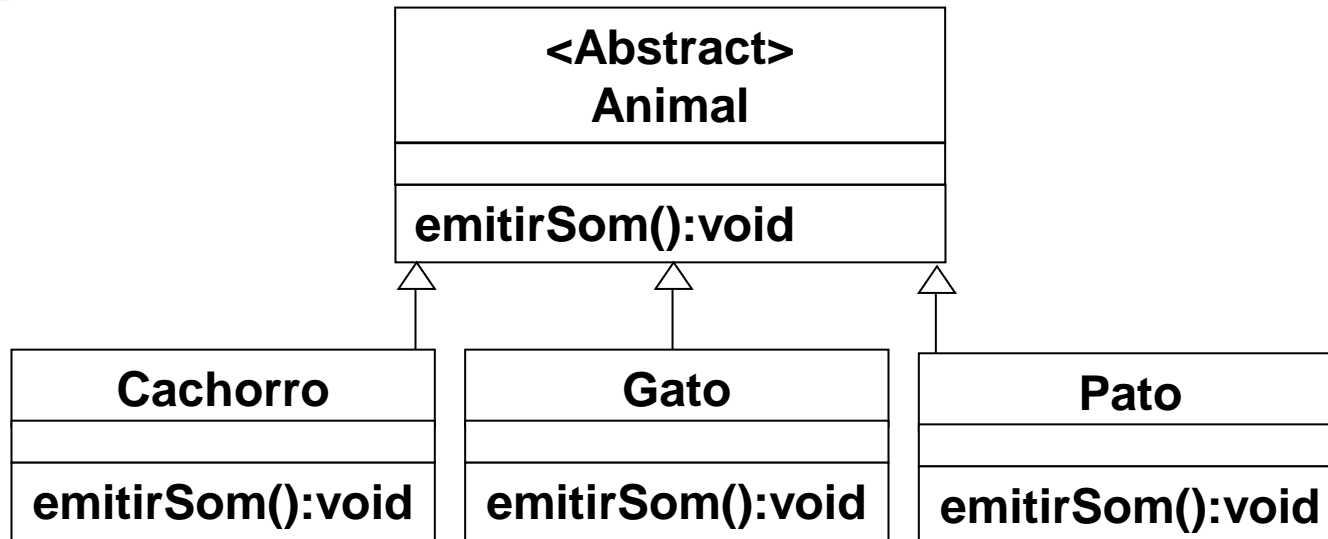


Classificação Polimorfismo



- Inclusão: por meio de uma referência para interfaces ou superclasses, é possível abordar um conjunto amplo de tipos de objetos.

Inclusão - exemplo



- Se um Cachorro é um Animal, o polimorfismo de inclusão diz que uma variável do tipo Animal pode guardar um objeto do tipo Cachorro. O mesmo vale para Gato e Pato.

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

Animal.java

```
public class Cachorro extends Animal{  
    public Cachorro(){}  
    public void emitirSom(){  
        System.out.println("Au, Au!!!");  
    }  
}
```

Cachorro.java

```
public class Gato extends Animal{  
    public Gato(){}  
    public void emitirSom(){  
        System.out.println("Miau!!!");  
    }  
}
```

Gato.java

Pato.java

```
public class Pato extends Animal{  
    public Pato(){}  
    public void emitirSom(){  
        System.out.println("Quack!!!");  
    }  
}
```

Principal.java

```
public class Principal{  
    public static void main(String[] args){  
        Animal x, y;  
        x = new Cachorro();  
        x.emitirSom();  
  
        y = new Gato();  
        y.emitirSom();  
    }  
}
```

- x e y são referências da classe Animal, x apontando para um objeto do tipo Cachorro e y para um objeto do tipo Gato.

```
import java.util.Random;

public class Principal1{
    public static void main(String[] args){
        Animal[] gaiolao = new Animal[3];
        gaiolao[0] = new Cachorro();
        gaiolao[1] = new Gato();
        gaiolao[2] = new Pato();

        Random aleatorio = new Random();

        ...
    }
}
```

• • •

// Vamos fazer uma brincadeira agora. A gente enfia a mão
// na gaiola (não vale olhar), pega um bicho e aperta o
// pescoço dele pra ver que som sairá.

// Pra fazer isso em Java, a coisa é um pouquinho mais
// chata, a tem que usar um mecanismo pra escolher um
// desses bichos de forma aleatória.

// Vamos então usar um esquema randômico de seleção – a
// classe do Java chamada Random. Esta parte do código para
// seleção é de pouca importância para o assunto de hoje...

• • •

...

```
Animal animalEscolhido;
```

```
// Enfiando a mão na gaiola...
```

```
for(int i = 0; i < 5; i++)
```

```
{
```

```
    // Pega aleatoriamente um bicho de cada vez da gaiola.
```

```
    animalEscolhido=gaiolao[aleatorio.nextInt(3)];
```

```
    // Aqui o animal escolhido emitirá o seu som...
```

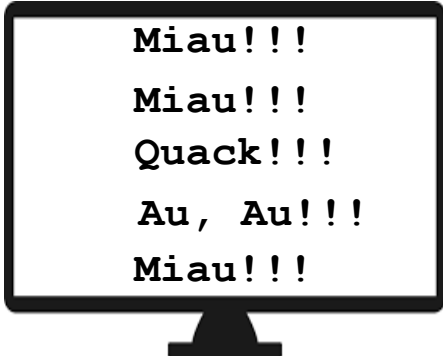
```
    // Estamos apertando um pescoço agora...
```

```
    animalEscolhido.emitirSom();
```

```
}
```

```
}
```

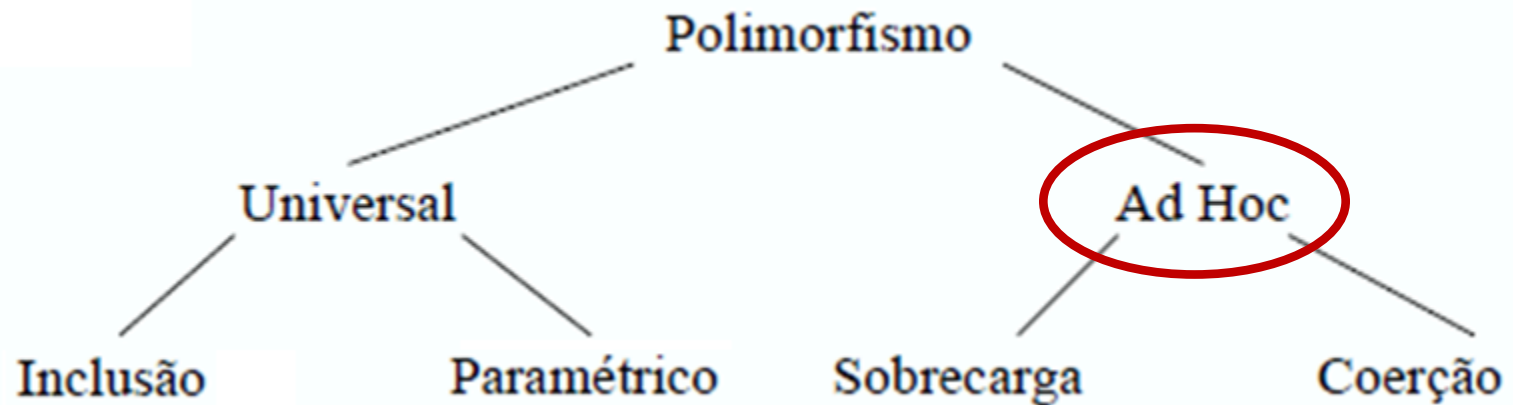
```
}
```



Miau!!!
Miau!!!
Quack!!!
Au, Au!!!
Miau!!!

- Polimorfismo de inclusão: uma variável do tipo da superclasse pode armazenar objetos da subclasse.

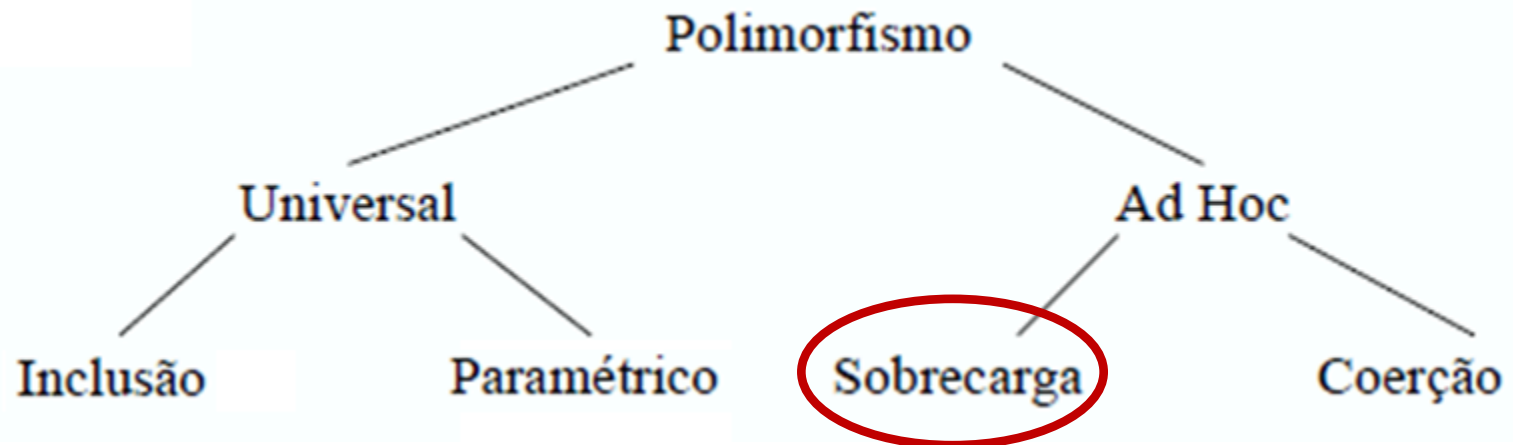
Classificação Polimorfismo



- Ad Hoc: ocorre em tempo de compilação e trabalha em um conjunto finito de tipos.



Classificação Polimorfismo



- Sobrecarga: métodos com mesmo nome e assinaturas diferentes, podem ter comportamentos totalmente distintos.

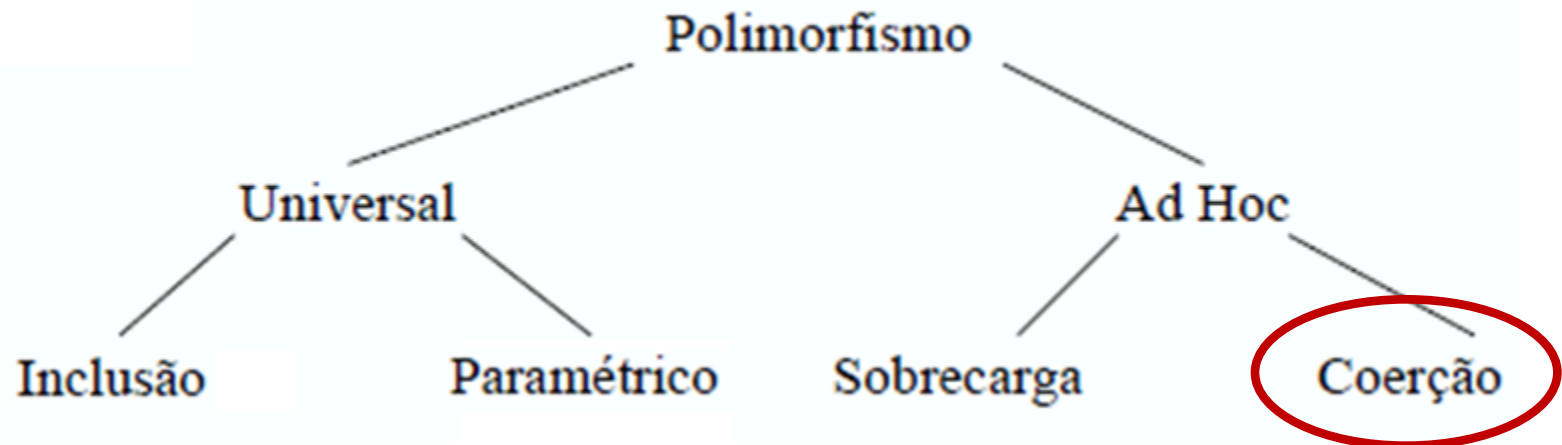
| |
|---------------------------------------|
| Exemplo 1 Sobrecarga de construtor |
|---------------------------------------|

```
public class ContaBancaria{  
    private String nomeT;  
    private double saldo;  
  
    public ContaBancaria(String nomeT) {  
        this(nomeT, 0.0);  
    }  
  
    public ContaBancaria(double saldo) {  
        this(" ", saldo);  
    }  
  
    public ContaBancaria(String nomeT, double saldo) {  
        this.setNomeT(nomeT);  
        this.setSaldo(saldo);  
    }  
}
```

```
public class Operacao{  
  
    ...  
  
    public int soma(int x, int y) {  
        return x + y;  
    }  
  
    public double soma(double x, double y) {  
        return x + y;  
    }  
  
    public String soma(String x, String y) {  
        return x + y;  
    }  
}
```



Classificação Polimorfismo

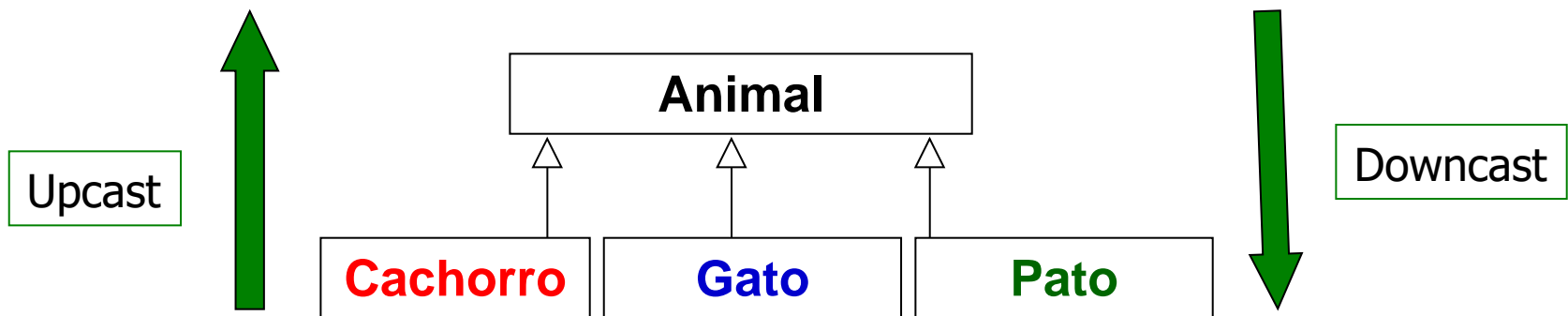


- Coerção: também conhecida como *casting*, consiste em forçar um objeto a assumir um tipo específico.



Coerção

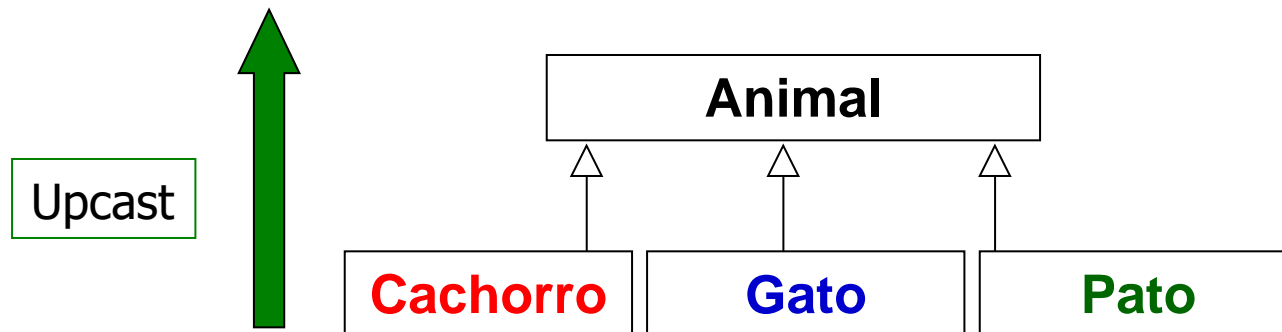
- Útil quando se deseja definir um modelo genérico que será redefinido ou especializado para outras classes de objetos. Para isso, o modelo deve ser construído a partir de tipos abstratos como interfaces e classes abstratas
- Tipos de coerção





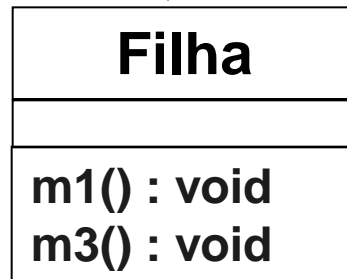
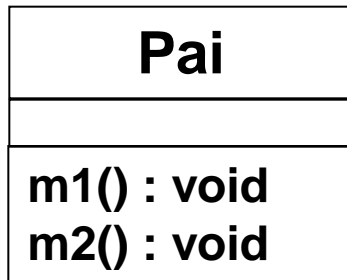
Coerção - Upcast

- Conversão de um tipo específico para um tipo mais genérico.
- Sempre seguro e automático.





Coerção - Upcast



```
Pai p = new Filha();
```

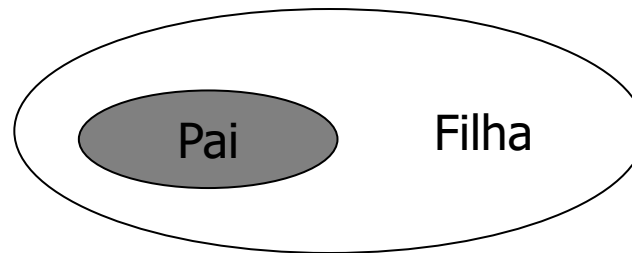
➤ **p** é uma referência da superclasse **Pai** apontando para um objeto da subclasse **Filha**

```
p.m1(); // chama m1() da classe Filha  
p.m2(); // chama m2() da classe Pai, herdado por Filha  
p.m3(); // ERRO de compilação
```



Coerção - Upcast

- A parte escura da figura mostra o que o Pai sabe.



- A Filha herda do Pai todo o seu conhecimento (atributos não privados) e comportamentos (métodos), mas pode acrescentar conhecimento e comportamentos novos exclusivamente seus, aos quais o Pai não tem acesso.



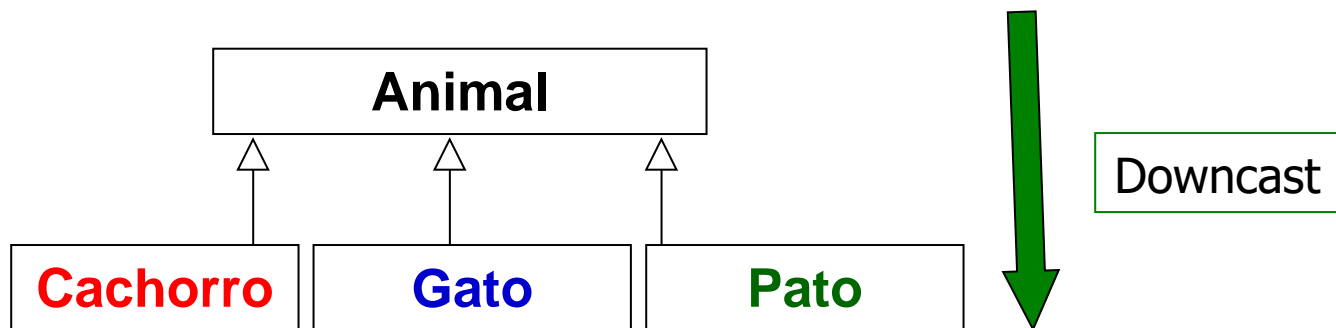
Coerção - Upcast

- Resumo:
 - É a conversão de um objeto do tipo mais específico para um tipo mais genérico.
 - Essa conversão é feita de forma implícita e automática no momento da atribuição.
 - A partir dessa conversão só os membros do tipo mais genérico podem ser acessados.



Coerção - Downcast

- Conversão de um tipo genérico para um tipo mais específico.
- Sempre verificada e explícita.





Coerção - Downcast

- O que acontece ao executar as instruções abaixo?

```
Pai p = new Filha();  
Filha f = p; // Erro
```

- Problema: O relacionamento É UM é sempre da subclasse para a superclasse. Logo, o compilador não entende que o objeto p, apesar de ser do tipo Pai, foi criado usando um construtor da classe Filha.
- Solução: Pode-se “forçar a barra” aplicando a coerção *downcast*.



Coerção - Downcast

```
1. Pai p = new Filha();  
2. Filha f = (Filha) p;  
3. f.m3();
```

- O *downcast* é realizado utilizando o operador *casting* de classes.
- O código será verificado em tempo de compilação.
- Se o objeto for da subclasse, a coerção será válida, mas se não for, ocorrerá um erro.
- Para proteger o código dessa incerteza, deve-se usar o operador especial **instanceof**



Coerção - Downcast

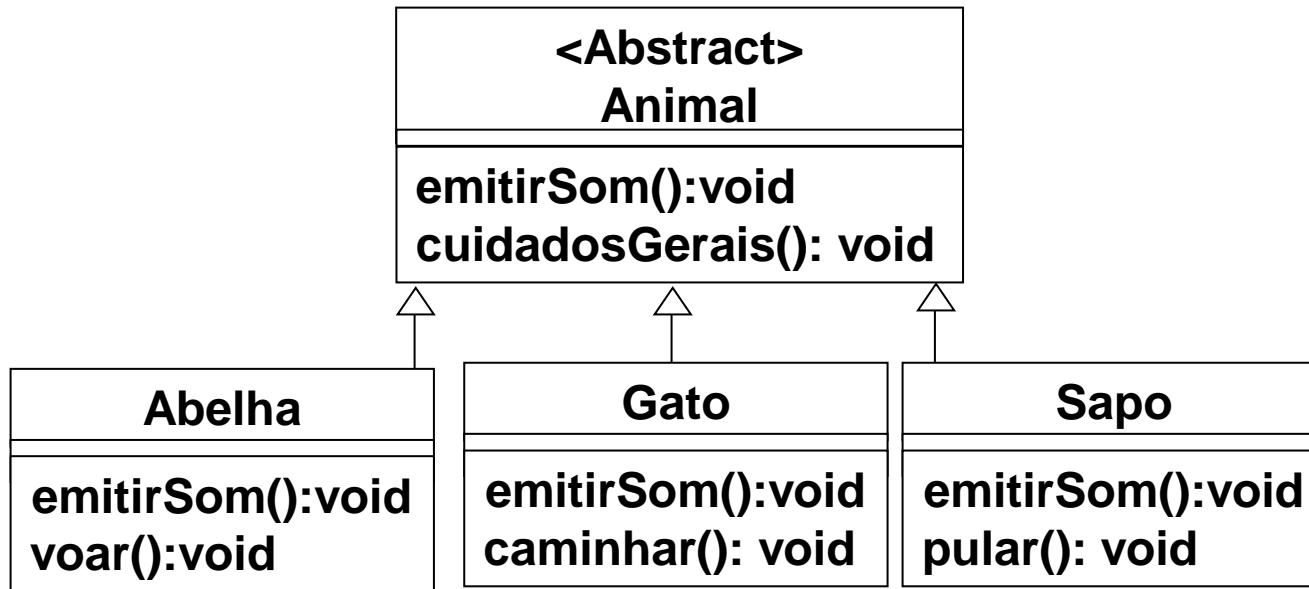
```
1. Pai p = new Filha();  
2. if (p instanceof Filha) {  
3.     Filha f = (Filha) p;  
4.     f.m3();  
5. }
```

- O operador **instanceof** deve ser usado como segue:
 <variavelObjeto> instanceof <NomeClasse>



Exemplo

- Vamos implementar este exemplo para praticar o conceito de inclusão e coerção.



```
public abstract class Animal
{
    public Animal() {}
    public void emitirSom() {
        System.out.println("sem som");
    }
    public void cuidadosGerais() {
        System.out.println("Cuidados importantes");
    }
}
```

| |
|--|
| <Abstract> Animal |
| emitirSom():void cuidadosGerais(): void |

```
public class Abelha extends Animal
```

Abelha.java

```
{
```

```
    public Abelha() {}
```

```
    public void emitirSom() {
```

```
        System.out.println("Bzzzzz, Bzz, Bzzz!");
```

```
    }
```

```
    public void voar() {
```

```
        System.out.println("Abelha voando...");
```

```
    }
```

```
}
```

| Abelha |
|---------------------------------|
| emitirSom():void voar():void |


```
public class Gato extends Animal
```

Gato.java

```
{
```

```
    public Gato() {}
```

```
    public void emitirSom() {
```

```
        System.out.println("Miau, miau, miau!");
```

```
    }
```

```
    public void caminhar() {
```

```
        System.out.println("Gato caminhando...");
```

```
    }
```

```
}
```

| Gato |
|-------------------------------------|
| emitirSom():void caminhar():void |

```
public class Sapo extends Animal
```

Sapo.java

```
{
```

```
    public Sapo() {}
```

```
    public void emitirSom() {
```

```
        System.out.println("Rabit, rabbit, rabbit!");
```

```
    }
```

```
    public void pular() {
```

```
        System.out.println("Sapo pulando...");
```

```
    }
```

```
}
```

| Sapo |
|----------------------------------|
| emitirSom():void pular():void |

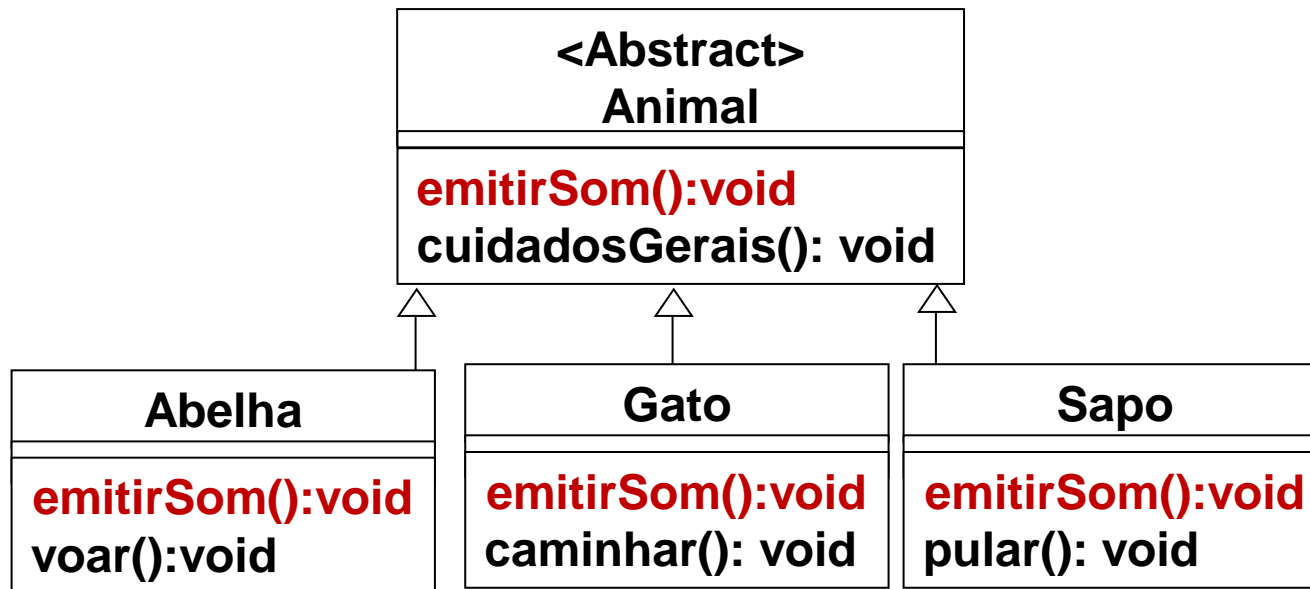
```
public class Principal{  
    public static void main(String[] args)  
    {  
        Animal a1 = new Abelha();  
        Animal a2 = new Gato();  
        Animal a3 = new Sapo();  
    }  
}
```

Está sendo utilizado o conceito de
polimorfismo?

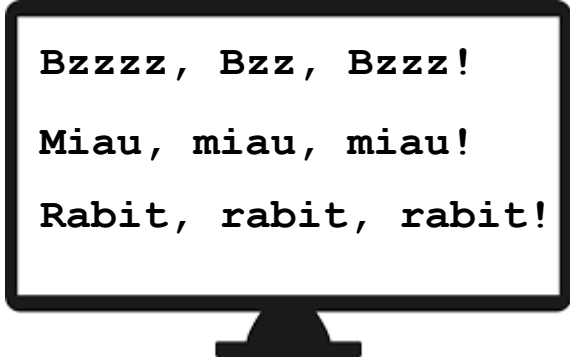
Sim. Inclusão e coerção *upcast*.



Exemplo - situação 1



```
public class Principal{  
    public static void main(String[] args)  
    {  
        Animal a1 = new Abelha();  
        Animal a2 = new Gato();  
        Animal a3 = new Sapo();  
  
        a1.emitirSom();  
        a2.emitirSom();  
        a3.emitirSom();  
    }  
}
```



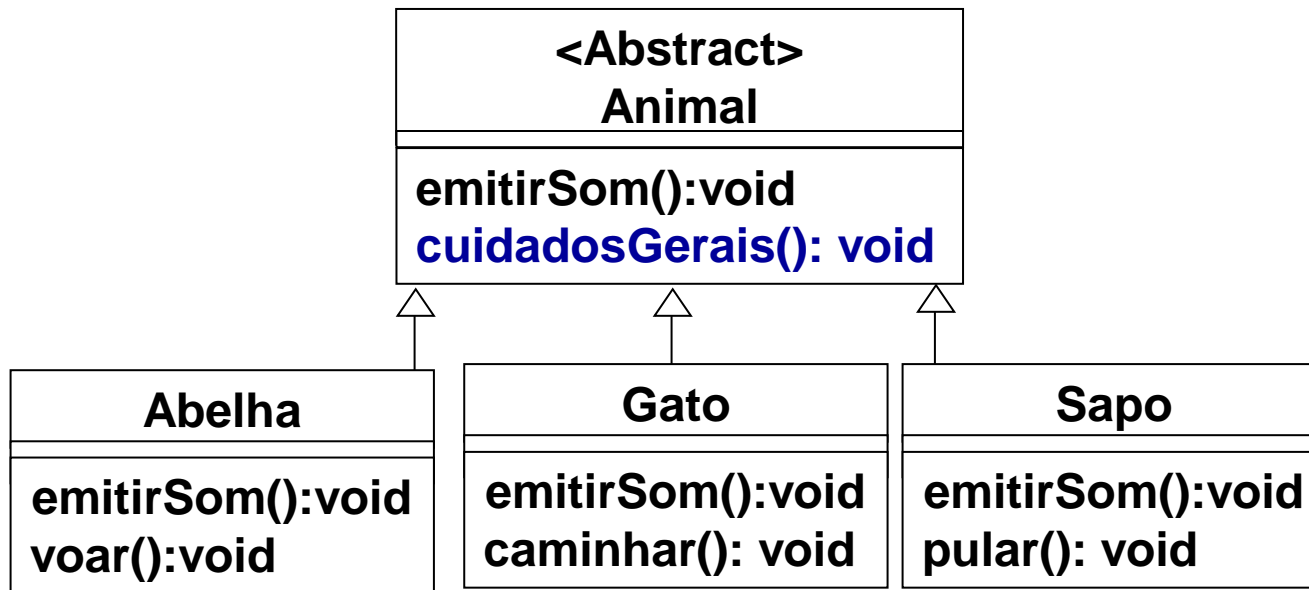
Bzzzz, Bzz, Bzzz!
Miau, miau, miau!
Rabit, rabbit, rabbit!

Situação 1

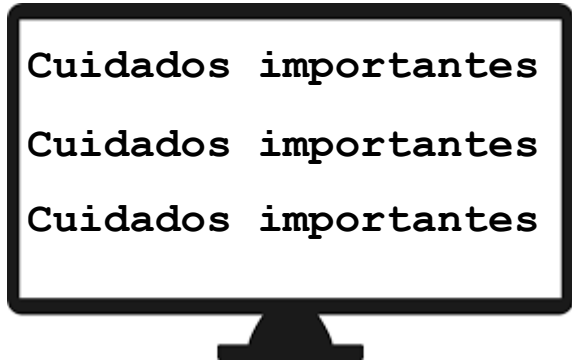
O conceito de polimorfismo **não** está sendo usado nas linhas de código que chamam o método emitirSom().



Exemplo - situação 2



```
public class Principal{  
    public static void main(String[] args)  
    {  
        Animal a1 = new Abelha();  
        Animal a2 = new Gato();  
        Animal a3 = new Sapo();  
  
        a1.cuidadosGerais();  
        a2.cuidadosGerais();  
        a3.cuidadosGerais();  
    }  
}
```



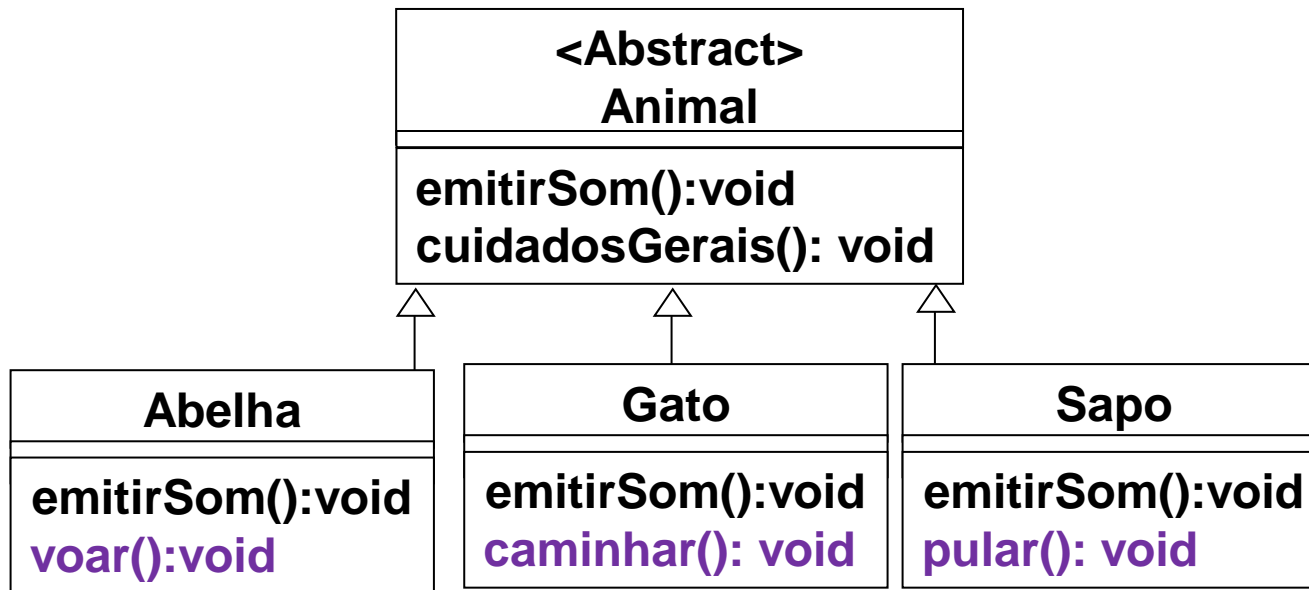
Cuidados importantes
Cuidados importantes
Cuidados importantes

Situação 2

O conceito de polimorfismo **não** está sendo usado nas linhas de código que chamam o método cuidadosGerais().



Exemplo - situação 3




```
public class Principal{  
    public static void main(String[] args)  
    {  
        Animal a1 = new Abelha();  
        Animal a2 = new Gato();  
        Animal a3 = new Sapo();  
  
        a1.voar(); ✗  
        a2.caminhar(); ✗  
        a3.pular(); ✗  
    }  
}
```

Situação 3

Que conceito do polimorfismo pode ser usado para corrigir o problema acima?

```
public class Principal{  
    public static void main(String[] args)  
    {  
        Animal a1 = new Abelha();  
        Animal a2 = new Gato();  
        Animal a3 = new Sapo();  
  
        if(a3 instanceof Sapo){  
            Sapo s = (Sapo)a3;  
            s.emitirSom();  
            s.pular();  
        }  
    }  
}
```

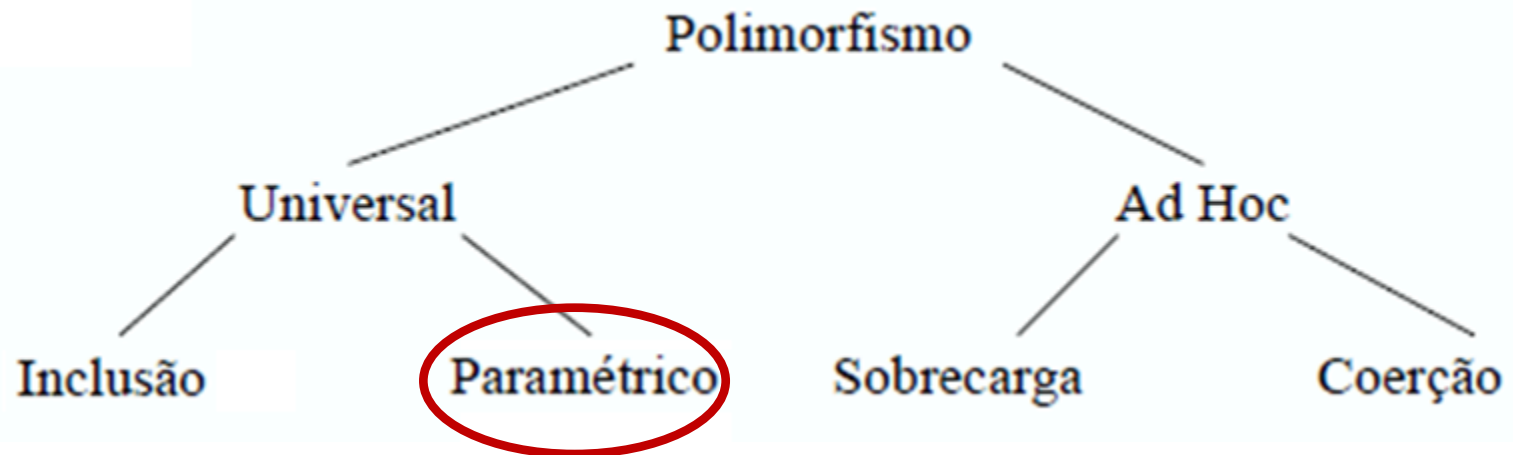
Resposta: coerção *downcast*

```
public class Principal{  
    public static void main(String[] args)  
    {  
        Animal a1 = new Abelha();  
        Animal a2 = new Gato();  
        Animal a3 = new Sapo();  
  
        if(a1 instanceof Abelha){  
            Abelha a = (Abelha)a1;  
            a.emitirSom();  
            a.voar();  
        }  
    }  
}
```

Aplicando o conceito de coerção *downcast* no objeto **a1** da classe abelha.



Classificação Polimorfismo



- Paramétrico: implementado em Java por meio do Generics.



Origem do Generics

- Introduzido no JDK 1.5
- Similar ao template de C++
- Permite abstrair os tipos



Origem do Generics

- Casting (conversão de tipos)

```
LinkedList lista = new LinkedList();
```



Problemas com casting

```
Integer x = Integer.valueOf(123);  
LinkedList lista = new LinkedList();  
lista.add(x);
```

✓ `Integer y = (Integer) lista.iterator().next();`

```
Vaca v = new Vaca();  
lista.add(v)
```

✗ `Integer z = (Integer) lista.iterator().next();`

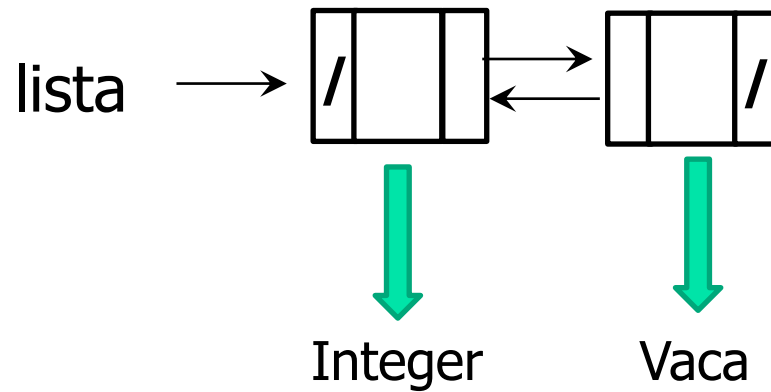
- Não é possível “forçar a conversão” de uma vaca para inteiro



número
inteiro = ?



Problemas com casting



```
Integer z = (Integer)lista.iterator().next();
```




Problemas com casting

- Evitar o uso de *casting* sempre que for possível.
- Destrói benefícios de linguagens com tipos fortemente definidos.



Utilizando Generics

Como trabalhar com listas em Java sem
que seja precisar usar o casting?

Utilizar Generics

- Generics permite que uma classe trabalhe com uma grande variedade de tipo (um de cada vez).

```
LinkedList<T> lista = new LinkedList<T>();
```



Utilizando Generics

- Exemplo 1

```
Integer x = Integer.valueOf(123) ;  
LinkedList<Integer> lista = new  
LinkedList<Integer>() ;  
lista.add(x) ;  
Integer num = (Integer)lista.iterator().next() ;
```

- Ao usar o generics não é preciso fazer o casting para converter um elemento da lista em Integer.



Utilizando Generics

- Exemplo 2

```
Vaca v = new Vaca();
```

```
LinkedList<Vaca> listaDeVaca = new  
LinkedList<Vaca>();
```

```
listaDeVaca.add(v);
```

```
Vaca z = (Vaca)listaDeVaca.iterator().next();
```

- Ao usar o generics não é preciso fazer o casting para converter um elemento da lista em Vaca.



Utilizando Generics

- É uma forma natural de eliminar a necessidade de se fazer casting.

```
LinkedList<T> lista = new LinkedList<T>();
```



Tipo do Objeto



Tipo do Objeto

- Benefício: código mais confiável.



Utilizando Generics


- Exemplo 3

```
Produto tv = new Televisao();  
Produto cel = new Celular();  
  
LinkedList<Produto> carrinho = new  
LinkedList<Produto>();  
  
carrinho.add(tv);  
carrinho.add(cel);  
  
Produto p = carrinho.iterator().next();
```

- Qualquer subclasse de Produto poderá ser armazenado no objeto carrinho.

Como criar classe com Generics

```
public class BasicGeneric <A> {  
    private A dado;  
  
    public A getDado() {  
        return this.dado;  
    }  
    public void setDado(A dado) {  
        this.dado = dado;  
    }  
}
```



- A é o parâmetro da classe (não pode ser do tipo primitivo).



Utilizando a classe BasicGenerics

- Instâncias da classe BasicGeneric “presa” ao tipo **String**.

```
String nome = "Laura";  
  
BasicGeneric<String> bG = new  
BasicGeneric<String>();  
  
bG.setDado(nome);  
  
String nome1 = bG.getDado(); // sem casting
```




Utilizando a classe BasicGenerics

- Instâncias da classe BasicGeneric “presa” ao tipo **Integer**.

```
Integer ano = new Integer(2022) ;
```

```
BasicGeneric<Integer> bG = new  
BasicGeneric<Integer>() ;
```

```
bG.setDado(ano) ;
```

```
Integer ano1 = bG.getDado() ; // sem casting
```



Atenção!!!

- Tipo Generics do Java são restritos a tipos de referência (objetos) e **não** funcionarão com tipos primitivos.

```
BasicGeneric<int> bG = new BasicGeneric<int>() ;
```





Cuidado!!!

- Evite “prender” instâncias de classe Generics ao tipo Object.

```
LinkedList<Object> lista = new  
LinkedList<Object>();
```

- Permite incluir qualquer tipo de objeto na lista.



Referências

- Deitel, P. J.; Deitel, H. M. (2017). Java como programar. 10a edição. São Paulo: Pearson Prentice Hall.
- Barnes, D. J. (2009). Programação orientada a objetos com Java: uma introdução prática usando o BlueJ (4. ed.). São Paulo, SP: Prentice Hall.
- Boratti, I. C. (2007). Programação orientada a objetos em Java. Florianópolis, SC: Visual Books.