Linguagens de Programação

Programação Orientada a Objetos

Andrei Rimsa Álvares
andrei@cefetmg.br

Sumário

- Introdução
- Programação Orientada a Objetos
  - Encapsulamento
  - Herança
  - Composição
  - Polimorfismo
- Resumo

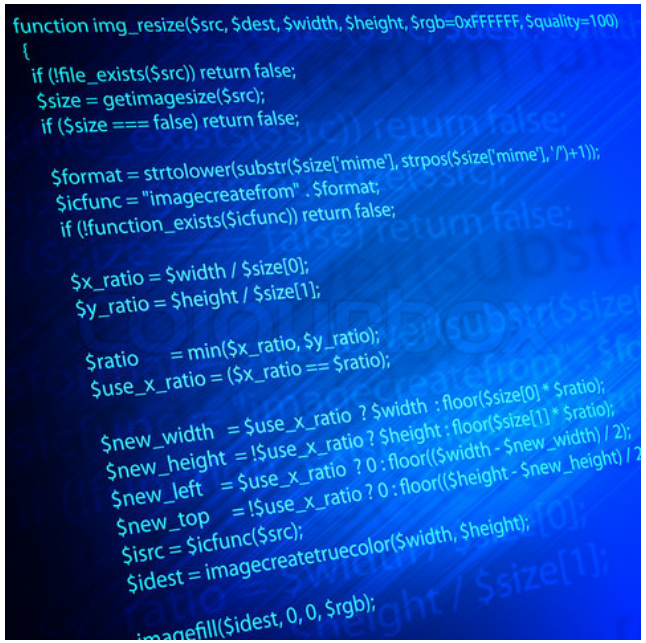
INTRODUÇÃO

Linguagens de Programação

Introdução

- Programas são compostos de duas partes

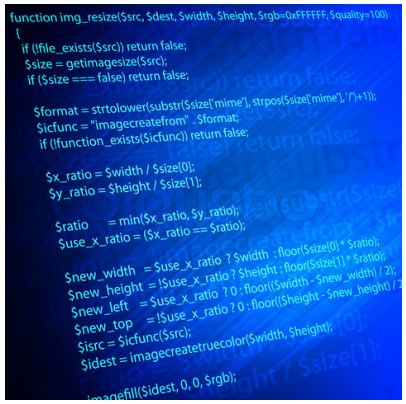
Dados



Código

Introdução

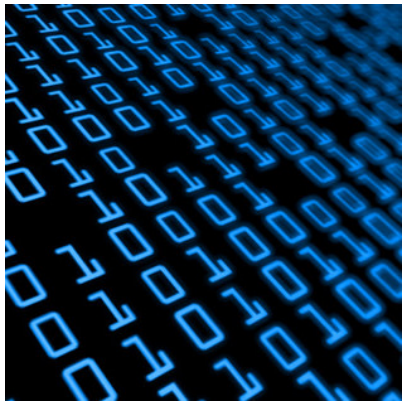
- Conceitualmente, programas podem ser organizados em volta de seus dados ou de seu código



Organizados em volta de seu **código**

"O que está acontecendo?"

Modelo orientado a **processos**



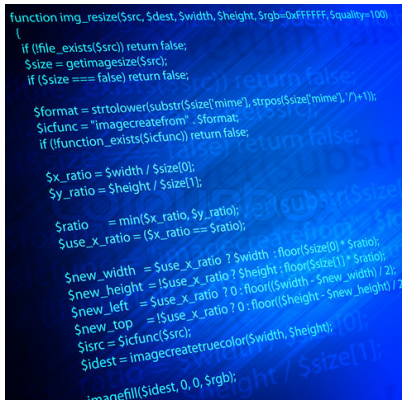
Organizados em volta de seus **dados**

"Quem está sendo afetado?"

Modelo orientado a **objetos**

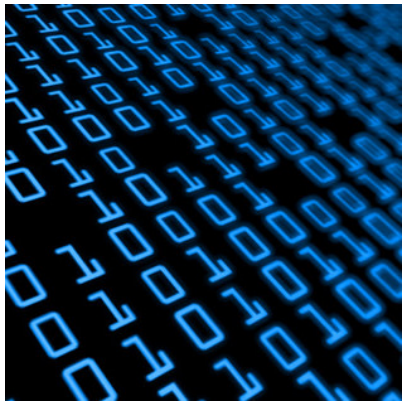
Introdução

- Conceitualmente, programas podem ser organizados em volta de seus dados ou de seu código



Modelo orientado a processos

- Caracterizado por um conjunto de passos lineares (código)
- Visto como código agindo sobre os dados

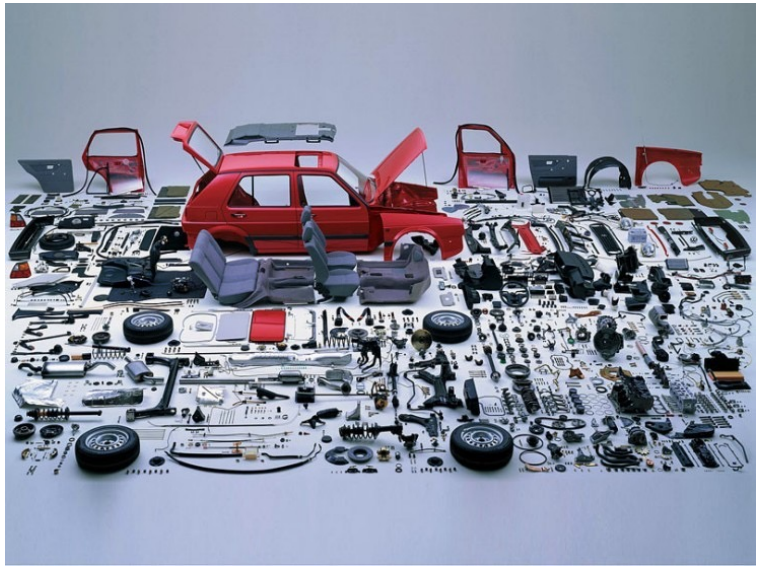


Modelo orientado a objetos

- Caracterizado por objetos (dados) e interfaces bem definidas de acesso a eles
- Visto como dados controlando acesso ao código

Abstração

- Humanos gerenciam a complexidade através de abstração



Complexidade

Abstração

Abstração

- Uma forma de gerenciar a complexidade é através de classificações hierárquicas

- Pode-se criar camadas semânticas de sistemas complexos subdividindo-os em unidades menores (mais gerenciáveis)

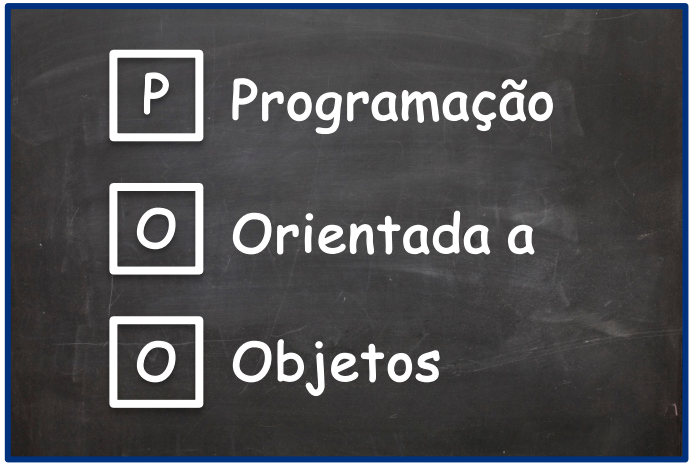
Subsistemas

- Som
- Direção
- Freios
- Segurança
- Resfriamento
- ...

Abstração

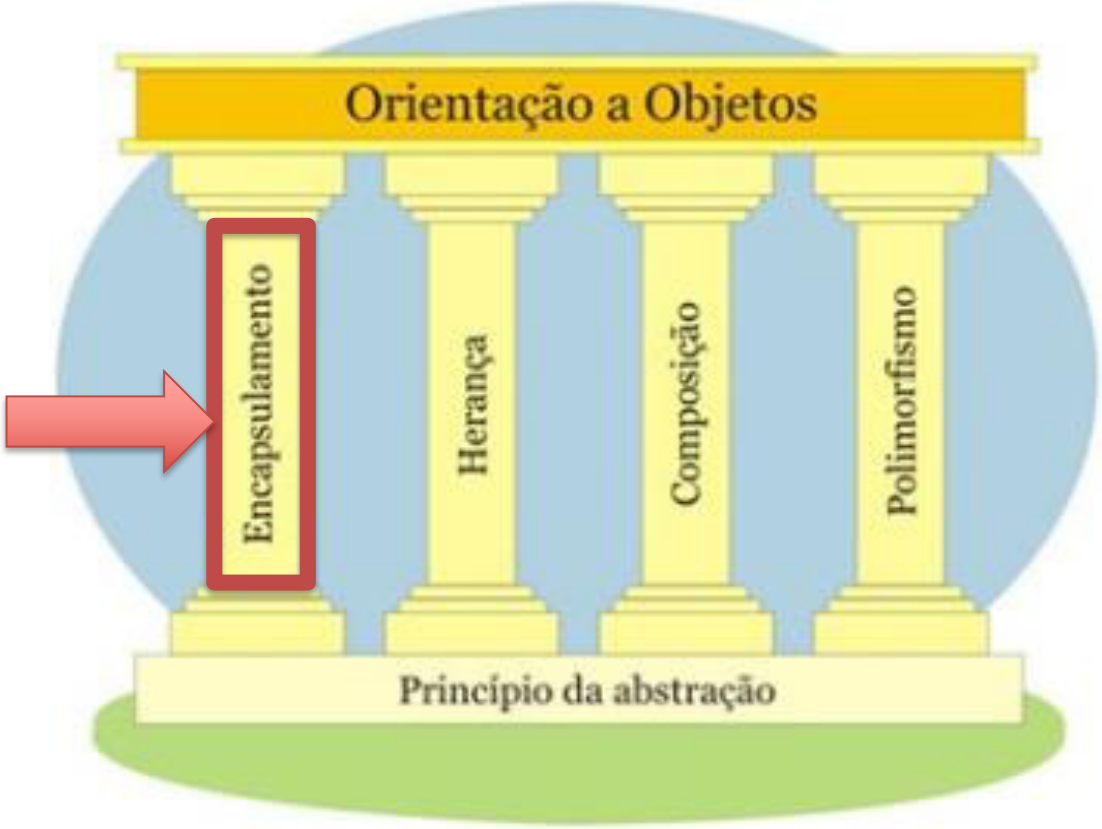
- O mesmo princípio pode ser aplicado à programação de computadores
  - Os dados podem ser abstraídos para seus componentes de objetos
  - Um conjunto de passos pode ser a comunicação entre esses objetos
    - Cada objeto possui seu próprio comportamento
    - Objetos seriam entidades concretas capazes de responder a comandos de “faça alguma coisa” (métodos)

ORIENTAÇÃO A OBJETOS



Abstração

- O modelo de programação orientado a objetos tem como pilares os seguintes princípios de abstração

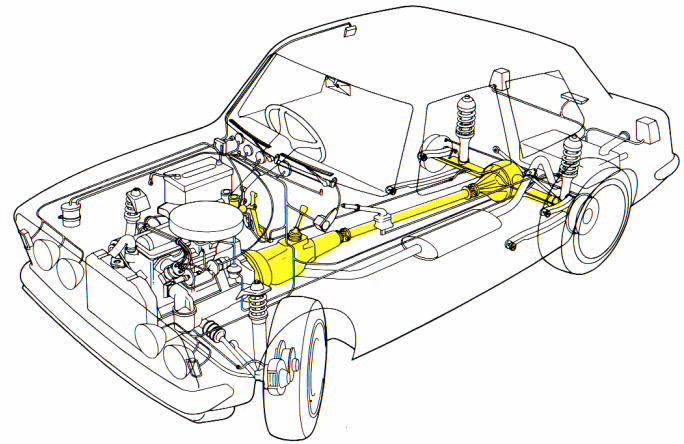


Encapsulamento

- Encapsulamento junta o código com os dados que ele manipula de forma a mantê-los seguros de interferências do lado de fora
- Pode ser visto como um invólucro de proteção que não permite que códigos externos acessem seu conteúdo
- O acesso aos dados é firmemente controlado por interfaces bem definidas

Exemplo

- Considere o sistema de transmissão automática de um carro qualquer

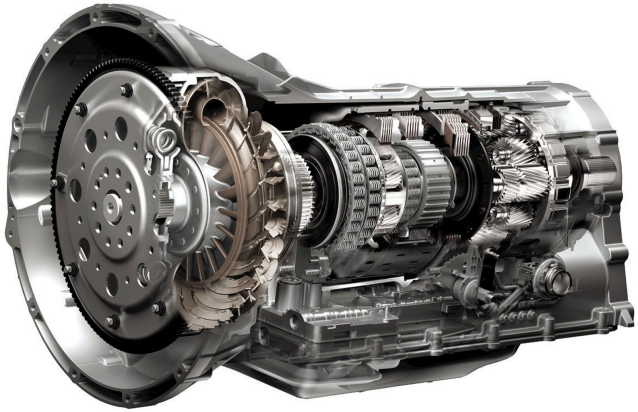


Encapsula

- A aceleração do carro
- Potência do motor
- Superfície de contato com o chão
- A posição do câmbio
- ...

Exemplo

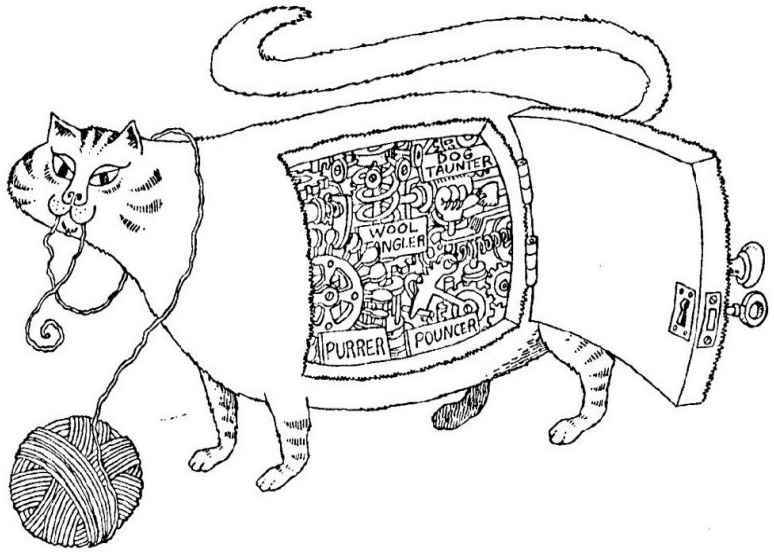
- Considere o sistema de transmissão automática de um carro qualquer



- Sistema bem encapsulado
  - Dar seta ou acionar o para-brisa não tem influência no sistema
- Cada fabricante pode implementar seu próprio sistema de transmissão
- E você dirige do mesmo jeito

Encapsulamento

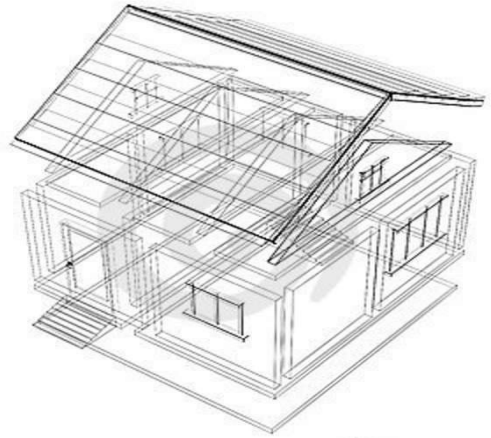
- O mesmo conceito se aplica à programação



- Todo mundo precisa saber como acessar a informação
  - Independente de como foi implementado
  - Sem medo de efeitos colaterais inesperados

Encapsulamento

- Na programação orientada a objetos, o encapsulamento é feito em uma classe

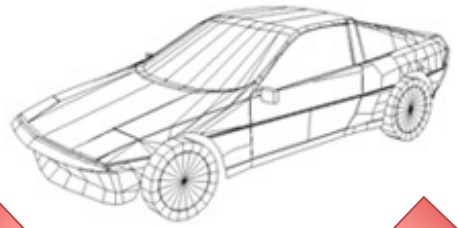


- A classe define a **estrutura** (dados) e o **comportamento** (código) que serão compartilhados entre os diversos objetos do sistema

- Objetos contém a estrutura e o comportamento da classe, como se fossem moldados por ela

Classe e Objeto

- Uma classe é uma construção lógica, enquanto um objeto é uma realidade física



Marca: ?
Cor: ?
Portas: ?

CLASSE

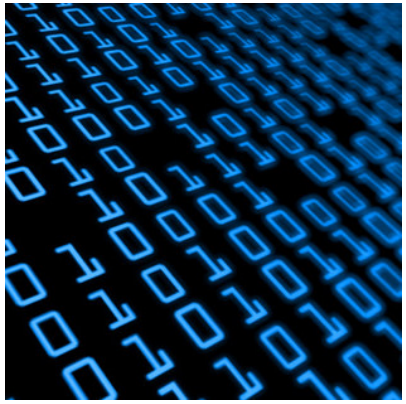
Marca: Lamborghini
Cor: Amarela
Portas: 2

Marca: Mercedes
Cor: Azul
Portas: 4

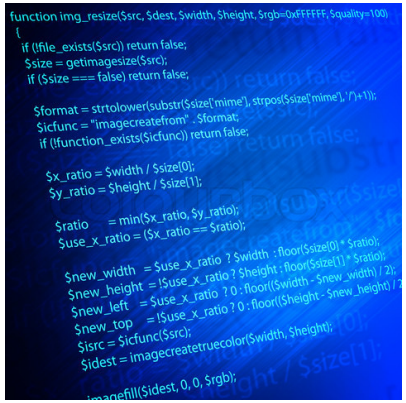
OBJETOS

Classe e Objeto

- Ao criar uma classe, pode-se definir os códigos e os dados (chamados coletivamente de membros)



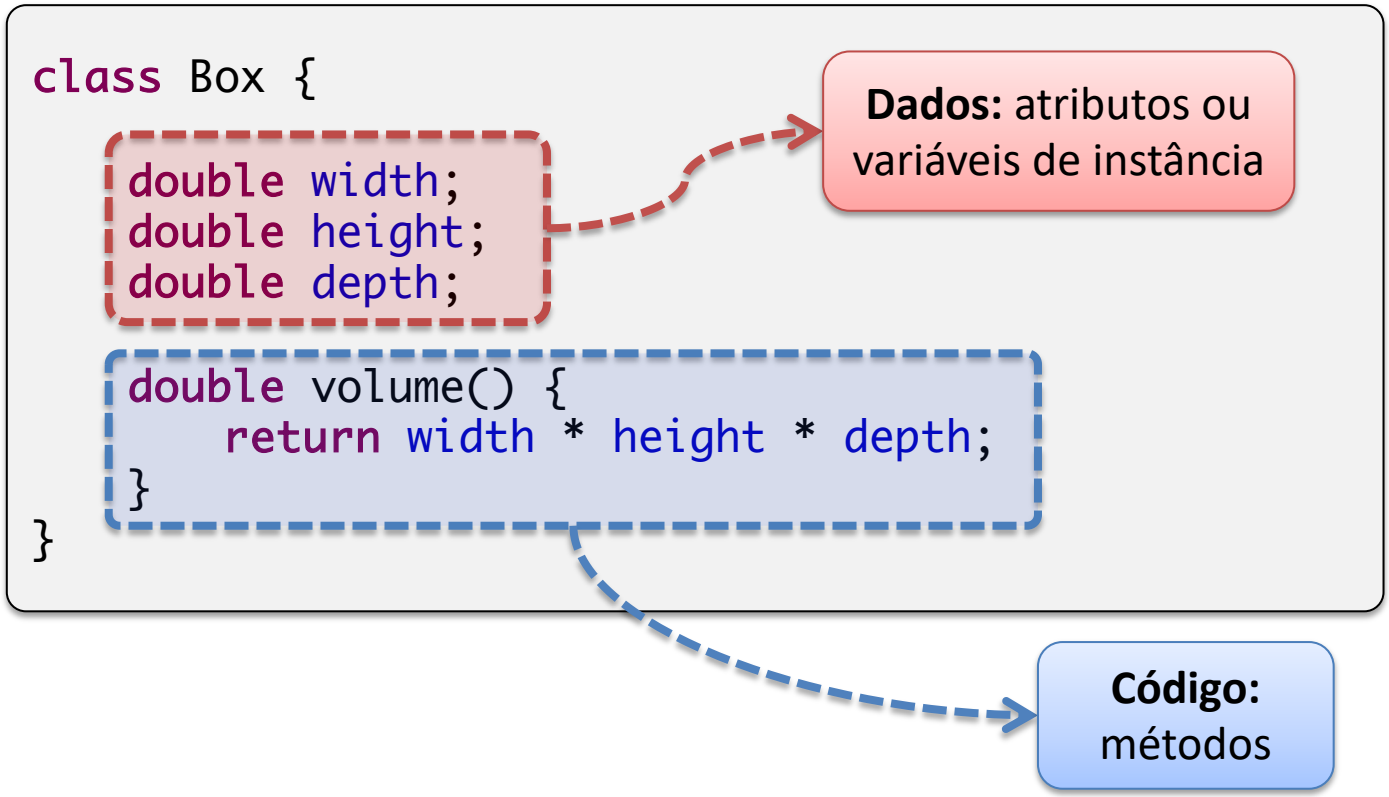
- Os dados são chamados de **variáveis membros**, **variáveis de instância** ou **atributos**



- Os códigos são chamados de **métodos membros** ou somente **métodos**

Classe

- Declaração de uma classe em Java que representa uma caixa que tem largura, altura e profundidade (**dados**) e é capaz de calcular seu volume (**código**) com base nesses dados

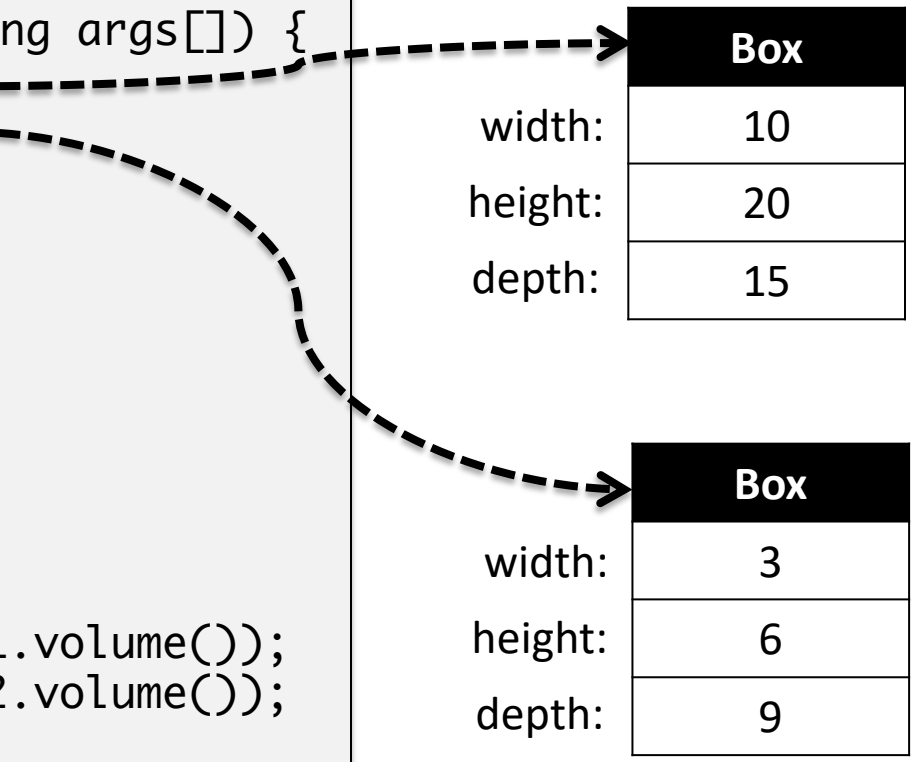


Objeto

- Criando dois objetos de caixa e exibindo seu volume

Onde fica a implementação de `volume`?

```
class BoxDemo {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        System.out.println(mybox1.volume());
        System.out.println(mybox2.volume());
    }
}
```



Controle de Acesso

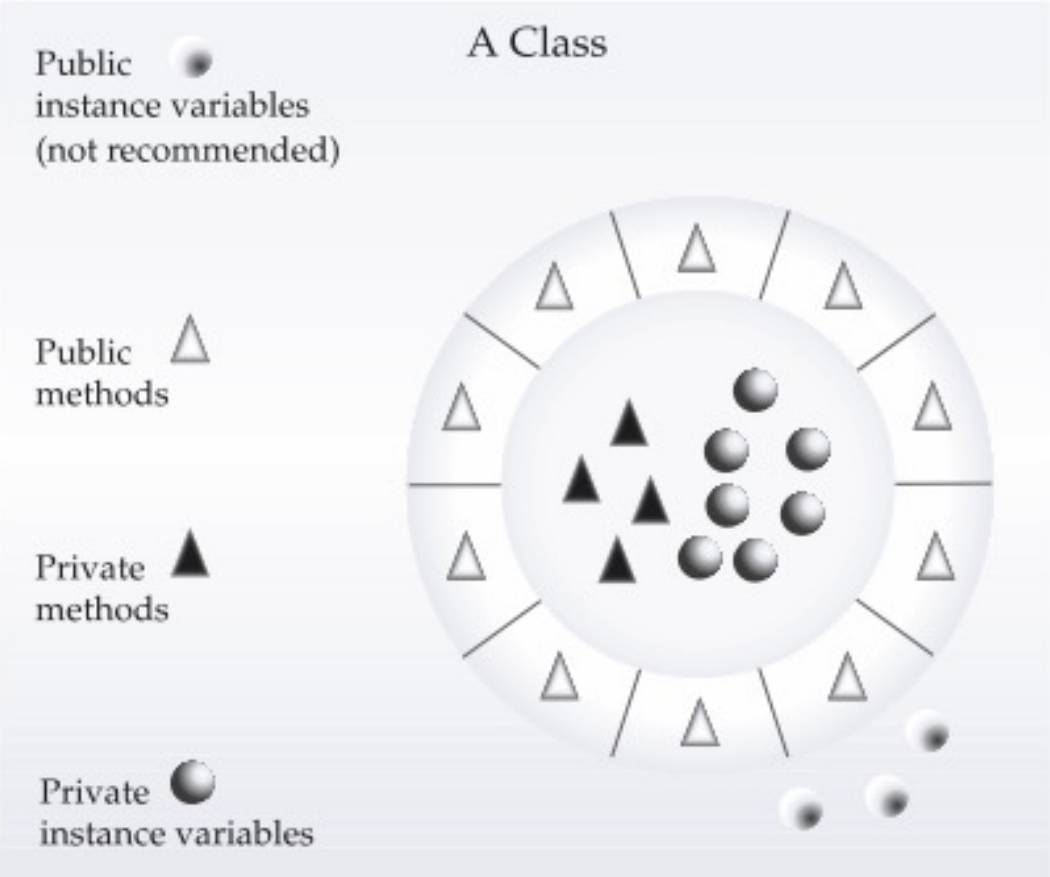
- As classes possuem um mecanismo para esconder a complexidade de sua implementação interna
  - Cada variável ou método pode ser marcado com um modificador: público ou privado

Existem outros modificadores?

- Membros privados
  - Não podem ser manipulados externamente, apenas pela própria classe
- Membros públicos
  - Tudo que classes externas precisam ou podem saber

Encapsulamento

- A interface pública deve ser bem projetada para não expor o funcionamento interno da classe



Encapsulamento

- Em Java: protegendo acesso aos atributos da classe

```
class Box {

    private double width;
    private double height;
    private double depth;

    public Box(double w, double h, double d) {
        this.width = w;
        this.height = h;
        this.depth = d;
    }

    public double getWidth() { return width; }
    public double getHeight() { return height; }
    public double getDepth() { return depth; }

    public double volume() {
        return width * height * depth;
    }
}
```

É possível alterar o valor da largura, altura e/ou profundidade a partir de outra classe?

Encapsulamento

- Em C++: protegendo acesso aos atributos da classe

```
class Box {

private:
     double width;
     double height;
     double depth;

public:
     Box(double w, double h, double d);
     double getWidth() const;
     double getHeight() const;
     double getDepth() const;

     double volume() const;

};
```

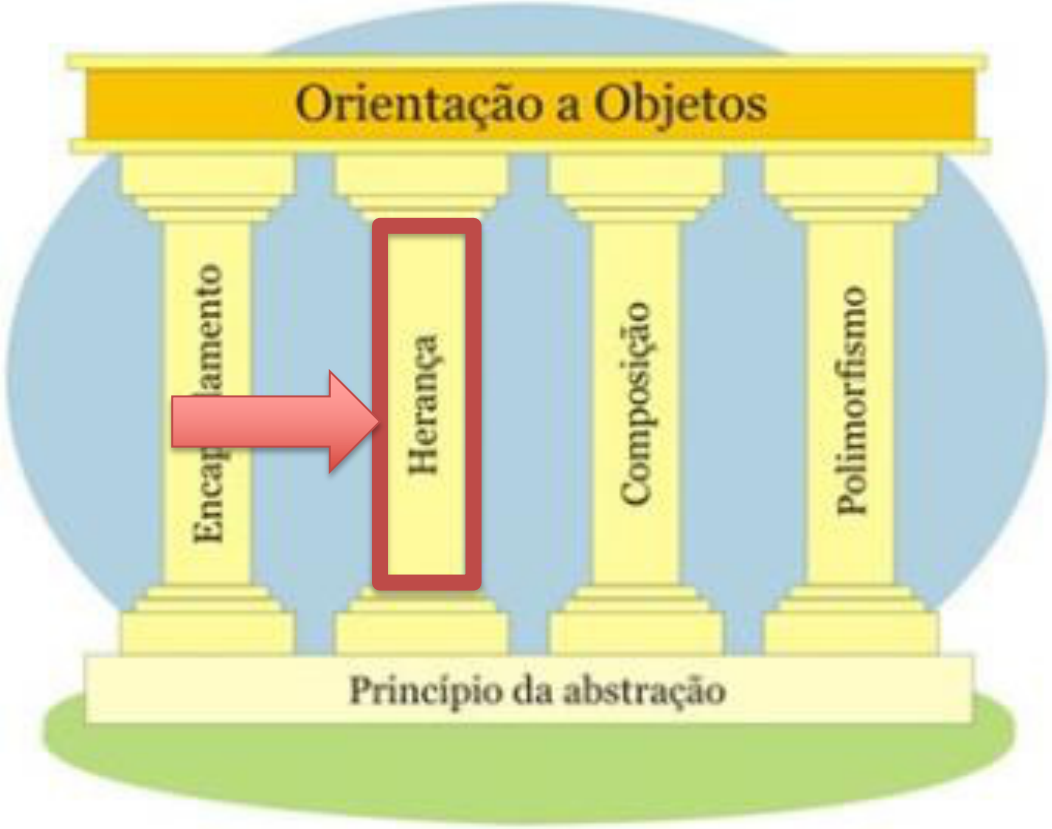
Box.h

```
Box::Box(double w, double h, double d) :
     width(w), height(h), depth(d) {
}

double Box::getWidth() const {
     return width;
}

double Box::getHeight() const {
     return height;
}

double Box::getDepth() const {
     return depth;
}

double Box::volume() const {
     return width * height * depth;
}
```

Box.cpp

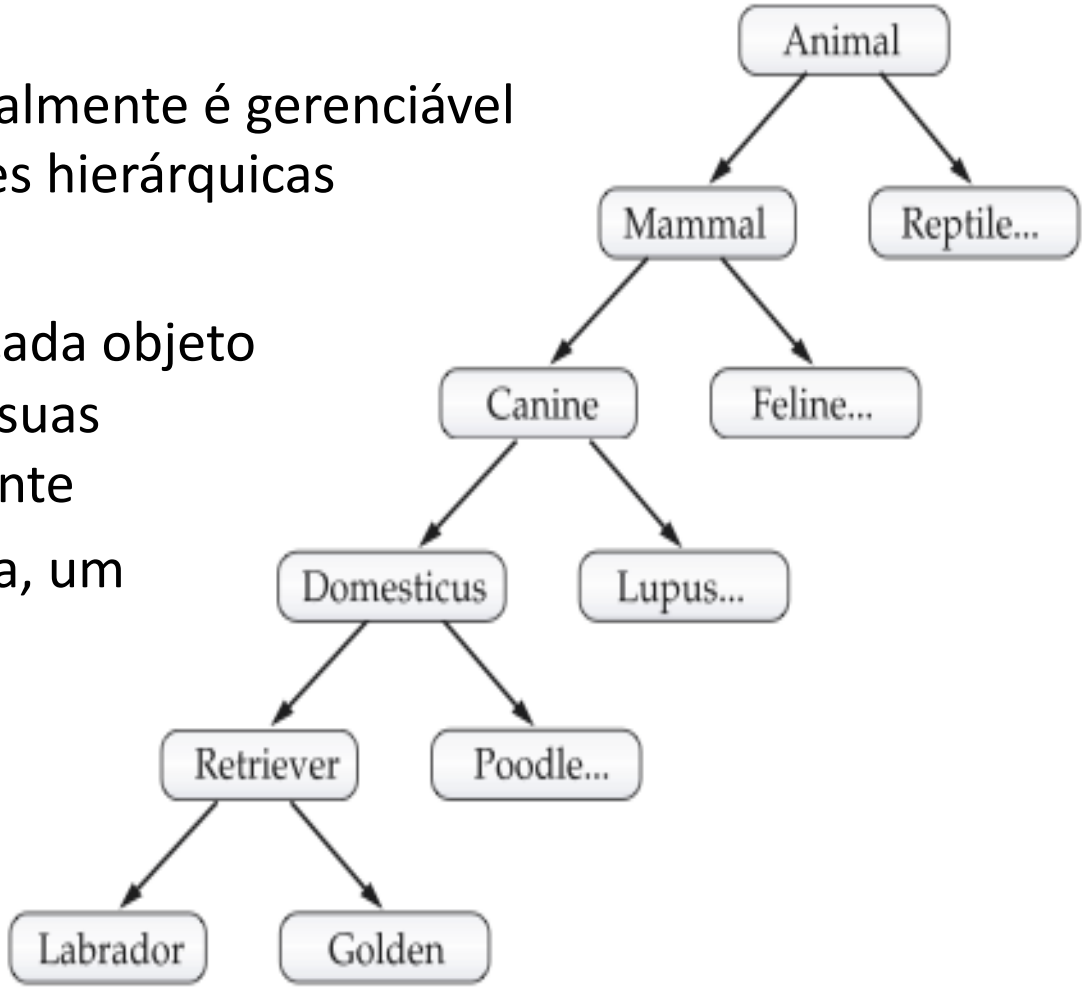
Abstração

- O modelo de programação orientado a objetos tem como pilares os seguintes princípios de abstração



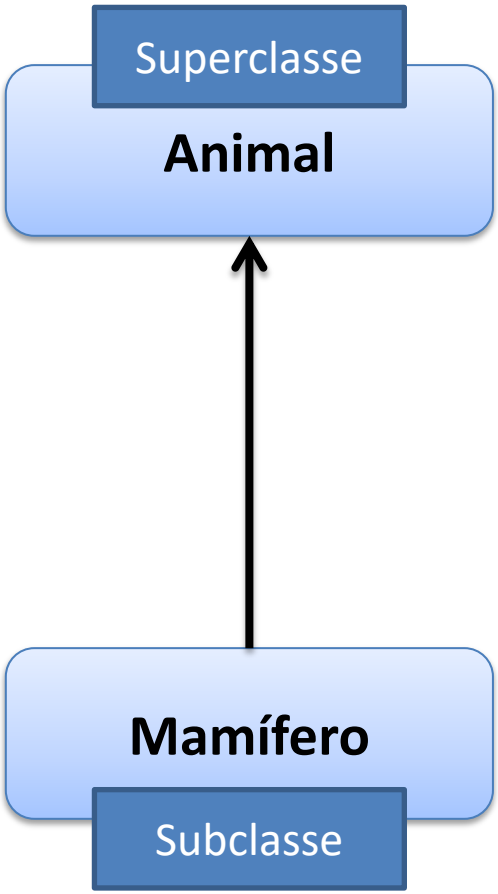
Herança

- Herança é o processo no qual um objeto obtém propriedades de um outro objeto
  - O conhecimento normalmente é gerenciável através de classificações hierárquicas



- Sem o uso de hierarquia, cada objeto precisaria definir todas as suas características explicitamente
  - Porém, usando herança, um objeto precisa definir apenas o que o torna único para a sua classe

Herança



- A classe **Animal** pode ser definida pelos seus atributos e comportamentos
  - **Atributos:** tamanho, inteligência, tipo de esqueleto
  - **Comportamentos:** comer, respirar, dormir

- A classe **Mamífero** é uma versão mais específica de Animal, mas que herda todos os atributos e comportamentos da classe mais geral
  - **Atributos específicos:** tipos de dentição, glândulas mamárias
  - **Comportamento específico:** amamentar

Herança e Encapsulamento

- Herança interage diretamente com encapsulamento
  - Se uma superclasse encapsula alguns atributos, então uma subclasse irá herdar esses atributos além daqueles definidos especificamente por ela

Herança

Encapsulamento

Herança

- Em Java: estendendo a classe *Box* adicionando o atributo *weight*

```
class BoxWeight extends Box {

  private double weight;

  public BoxWeight(double w, double h,double d, double e) {
    super(w, h, d);
    this.weight = e;
  }

  public double getWeight() {
    return weight;
  }
}
```

```
class DemoBoxWeight {
  public static void main(String[] args) {
    BoxWeight mybox =
        new BoxWeight(10, 20, 15, 34.3);

    System.out.println("Volume mybox: " +
                       mybox.volume());
    System.out.println("Peso mybox: " +
                       mybox.getWeight());
  }
}
```

Herança

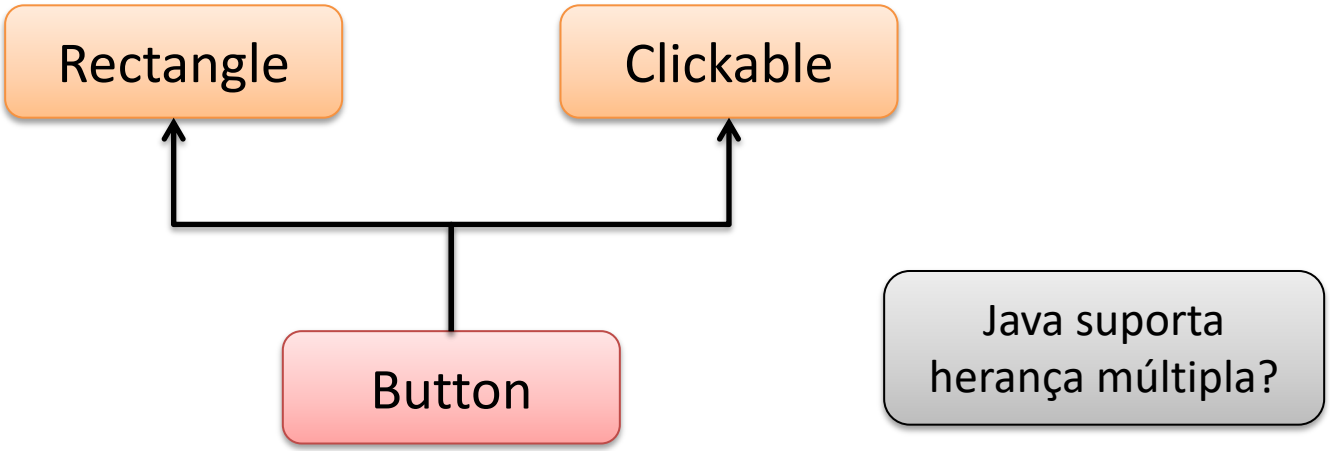
- Em C++: estendendo a classe *Box* adicionando o atributo *color*

```
class BoxColor : public Box {

private:
    int color;

public:
    BoxColor(double w, double h, double d, double c)
        : Box(w, h, d), color(c) {
    }

    ~BoxColor() {
    }

    int getColor() const {
        return color;
    }

};
```

Herança Múltipla

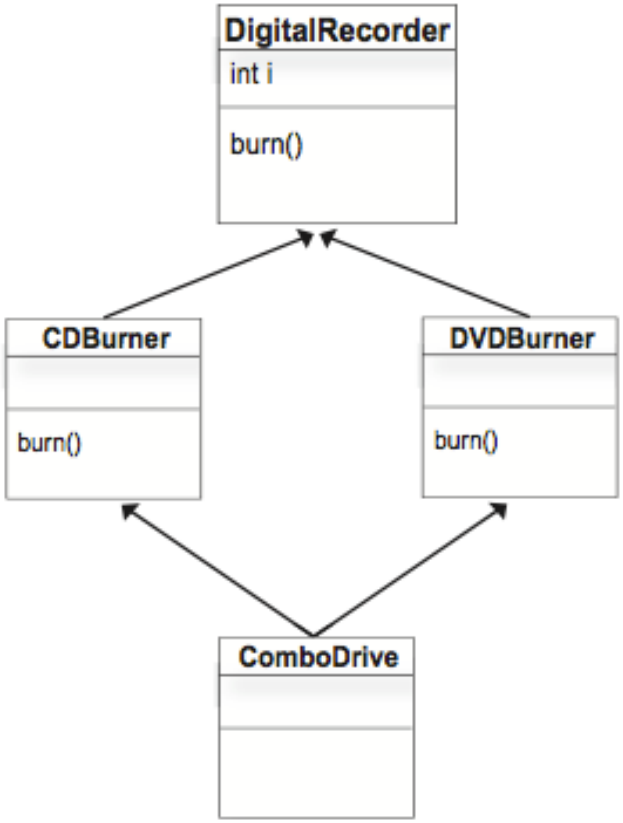
- Algumas linguagens, como C++, permitem que se herde características de mais de uma classe

- Por exemplo: a classe **Button** herda de **Rectangle** (para desenho) e **Clickable** (para funcionalidade)



Herança Múltipla

- Herança múltipla é também conhecida como **diamante da morte mortal** (*deadly diamond of death*)

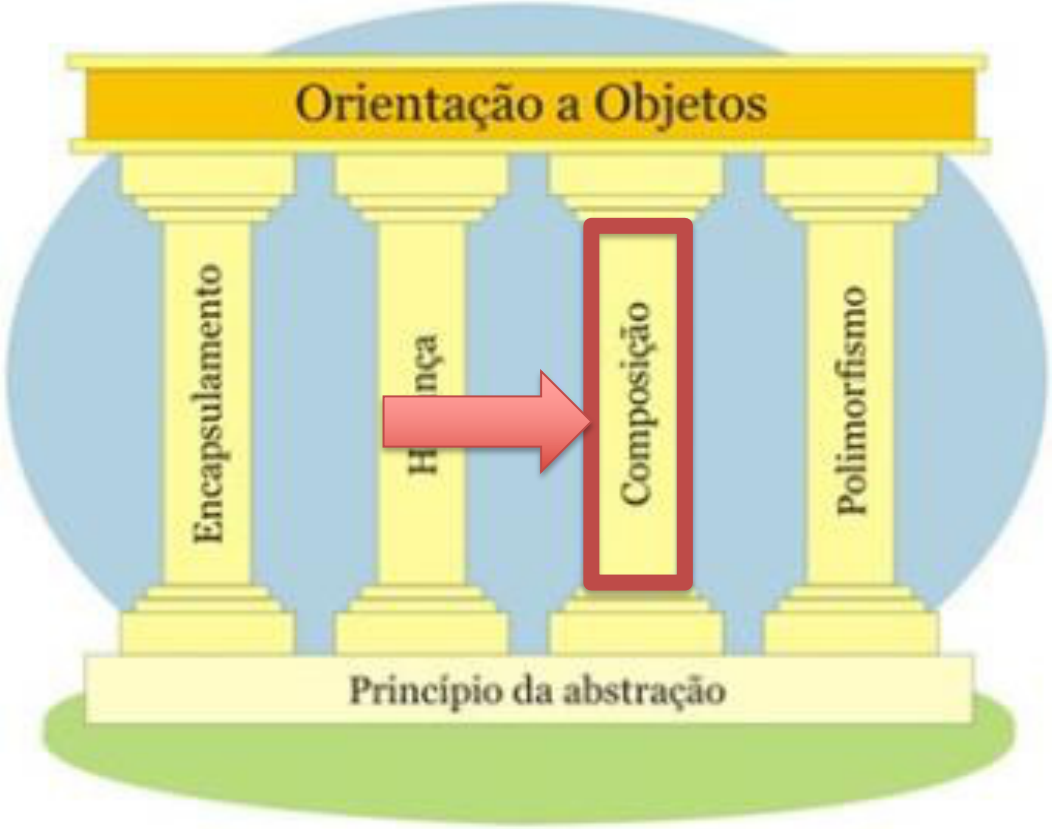


- O que acontece em **ComboDrive** se **CDBurner** e **DVDBurner** utilizam diferentes valores para o atributo **i**?
- E **ComboDrive** irá usar qual método **burn()**, de **CDBurner** ou **DVDBurner**?

Como Java lida com esse problema?

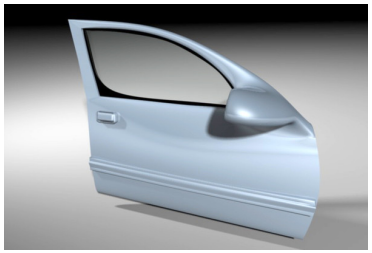
Abstração

- O modelo de programação orientado a objetos tem como pilares os seguintes princípios de abstração



Composição

- Composição é uma forma de combinar objetos simples com o objetivo de formar objetos mais complexos
  - Implica em uma relação de **ter** (ou é composto), ao invés de uma relação de **ser** (obtida via herança)



Um carro possui…

Composição

- Novos objetos podem ser criados **combinando** objetos já existentes (princípio da reutilização)

- Um objeto (A) pode ter uma **relação** com outro objeto (B) de diversas formas (multiplicidade)
  - **0..1:** A tem **no máximo um** B
  - **1..1:** A tem **um e somente um** B
  - **0..*:** A **tem muitos** B's
  - **1..*:** A **tem um ou mais** B's
  - **3..5:** A tem **de três a cinco** B's (valores específicos)

Composição

- A multiplicidade é implementada usando objetos como atributos
- Um objeto pode ter um objeto como atributo
  - Ex.: Um relógio possui data e hora
- Um objeto pode ter uma coleção de objetos como atributo
  - Ex.: Uma turma possui vários alunos (implementado como *arrays*, coleções ou outros)

Exemplo em C++

- Exemplo de objetos como atributos

```
class Date {
public:
    int day, month, year;

    Date(int d, int m, int y) :
        day(d), month(m), year(y) {
    }
};
```

```
class Time {
public:
    int hour, minute, second;

    Time(int h, int m, int s) :
        hour(h), minute(m), second(s) {
    }
};
```

```
class Clock {
    Date* date; Time* time;

public:
    Clock(Date* d, Time* t) : date(d), time(t) { }
    ~Clock() { delete date; delete time; }

    void show() {
        printf("%02d:%02d:%04d %02d:%02d:%02d", date->day, date->month,
            date->year, time->hour, time->minute, time->second);
    }
};
```

Exemplo em Java

- Exemplo de coleções de objetos como atributos

```
class Turma {

    Aluno[] alunos;

    Turma(int n) {
        alunos = new Aluno [n];
    }

    boolean matricular(Aluno a) {
        for (int i = 0;
                i < alunos.length; i++) {
            if (alunos[i] == null) {
                alunos[i] = a;
                return true;
            }
        }
        return false;
    }
}
```

ou

```
import java.util.*;
class Turma {

    List<Aluno> alunos = new
        ArrayList<Aluno>();

    void matricular(Aluno a) {
        alunos.add(a);
    }
}
```

Herança vs. Composição

- **Herança** e **composição** são dois mecanismos para reutilização de funcionalidades
  - **Herança**: estende atributos e métodos de uma classe
  - **Composição**: estende através de delegação
- **Herança** sempre foi considerada como a ferramenta básica para extensão e reuso, mas **composição** é muito superior na maioria dos casos
  - Herança define um relacionamento estático, enquanto composição um relacionamento dinâmico
- **Composição** e **herança** não são mutuamente exclusivas; as técnicas podem ser combinadas para obter melhores resultados

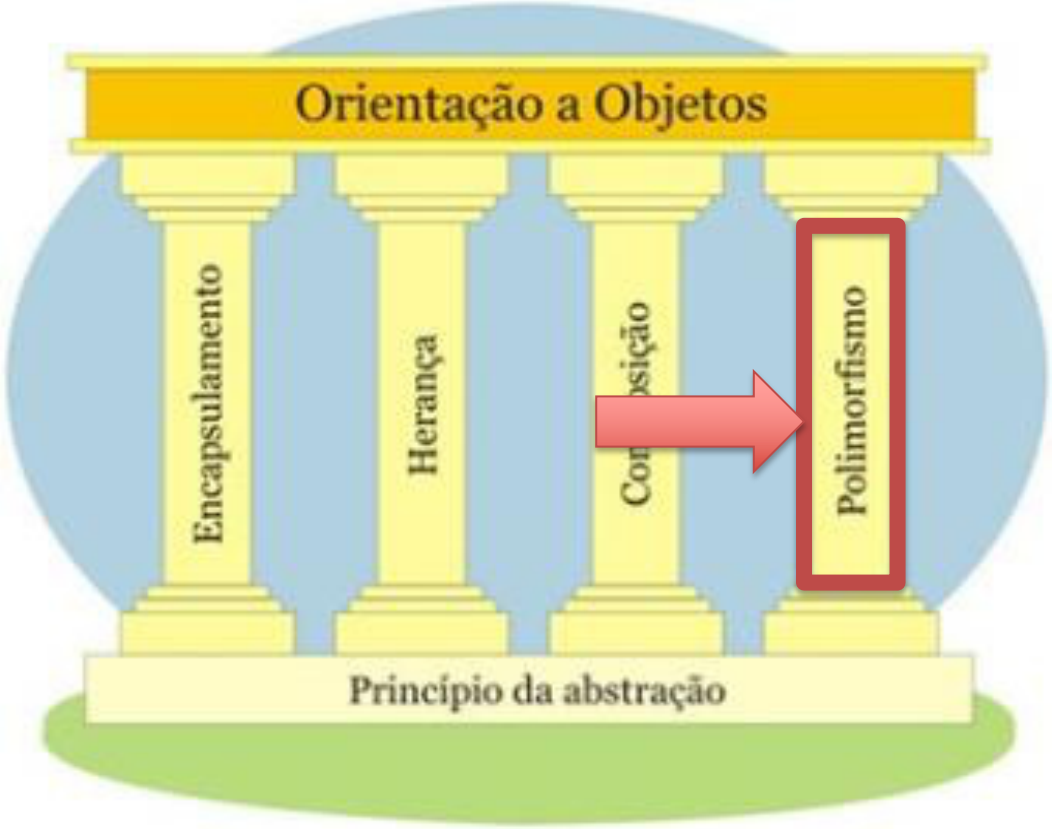
Herança vs. Composição

- Quando usar **Herança?**
  - Somente deve ser usada quando estiver construindo uma família de tipos (relacionados entre si)
  - Somente se puder comparar um objeto A com outro objeto B dizendo que, A **"É UM tipo de..."** B
- Quando usar **Composição?**
  - Em todos os outros casos!

Dica: sempre favoreça composição sobre herança

Abstração

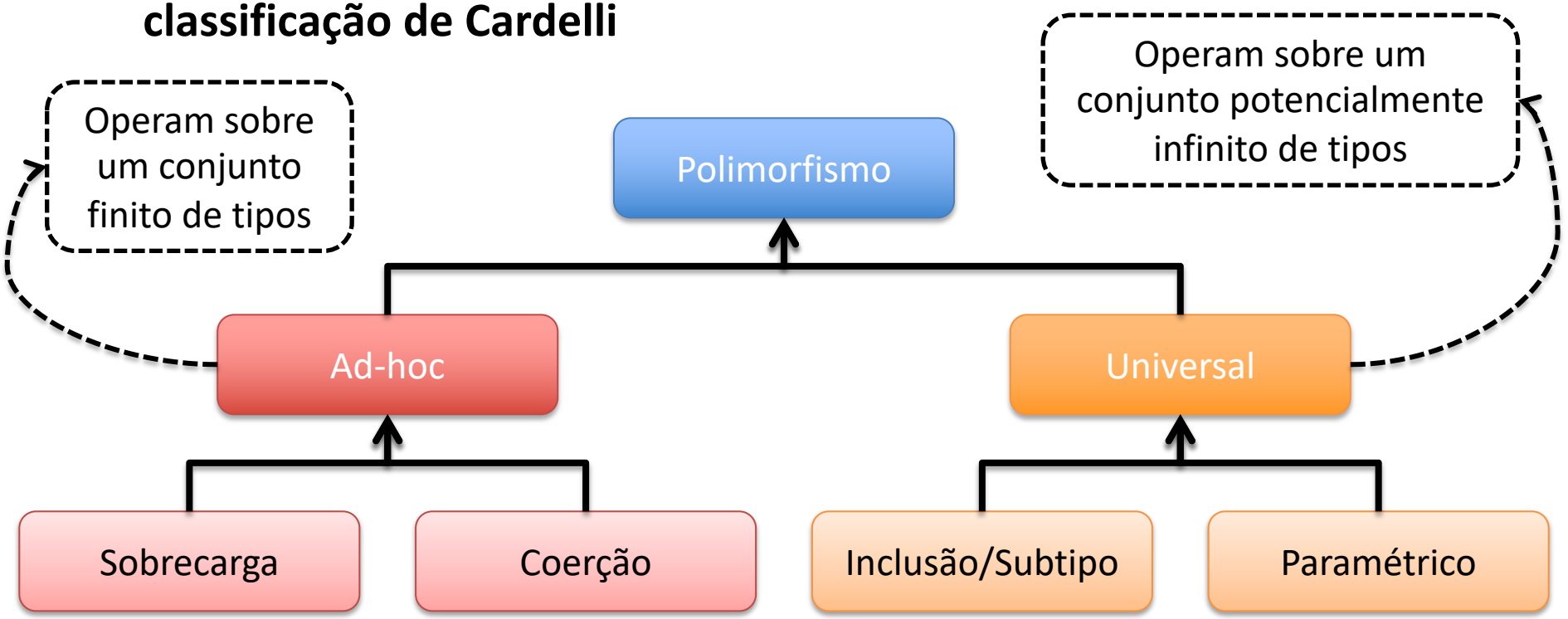
- O modelo de programação orientado a objetos tem como pilares os seguintes princípios de abstração



Polimorfismo

- **Definição:** abstrações que operam uniformemente sobre valores de tipos diferentes

- Os tipos de polimorfismos podem ser organizados de acordo com a **classificação de Cardelli**



Polimorfismo Ad-hoc: Sobrecarga

- Permite que um “nome de função” seja usado mais de uma vez com diferentes tipos de parâmetros
- O compilador automaticamente chama a função “correta” que deve ser utilizada
- Exemplo: sobrecarga de métodos

Polimorfismo: métodos relacionados sob um nome comum

```
class Math {

    public static int abs(int a);
    public static long abs(long a);
    public static float abs(float a);
    public static double abs(double a);

}
```

Em Java
(com sobrecarga de métodos)

```
int abs(int a);
long labs(long a);
long long llabs(long long a);
float fabsf(float a);
double fabs(double a);
```

Em C
(sem sobrecarga de métodos)

Polimorfismo Ad-hoc: Sobrecarga

- Algumas linguagens permitem a sobrecarga de operadores
- Exemplo em C++

Quais suas vantagens?
E desvantagens?

```
struct Point {

    int x, y;

    Point(int x, int y) : x(x), y(y) {
    }

    Point operator+(Point p) {
        return Point(x + p.x, y + p.y);
    }
};
```

```
Point p1(10, 5);
Point p2(2, 4);

Point p3 = p1 + p2;
```

Polimorfismo Ad-hoc: Coerção

- Em linguagens de programação é comum atribuir um valor de um determinado tipo para uma variável de um tipo diferente (coerção)

- Podem ser divididas da seguinte maneira
  - **Conversões implícitas:** feitas automaticamente pelo compilador
    - Ex. em C#

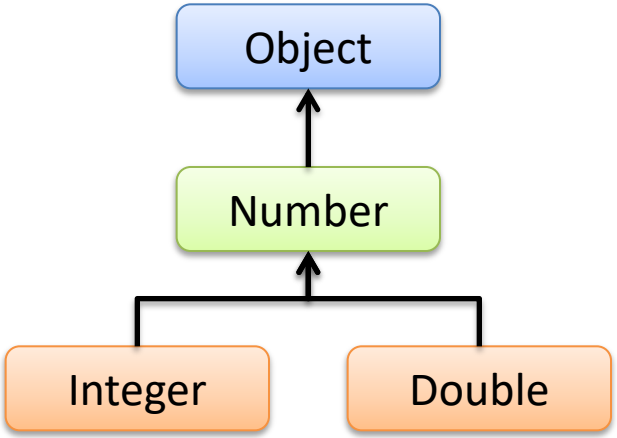
```
short num = 2;
int numInteiro = num;
string res = "Numero: " + num;
```

  - **Conversões explícitas:** especificadas manualmente pelo programador (*type casting*)
    - Ex. em C#

```
int num = 0;
byte x = (byte) num;
byte y = byte.Parse(num);
```

Polimorfismo Ad-hoc: Coerção

- Java permite conversão **implícita** ou **explícita** dependendo da hierarquia de classes




```
Integer i = Integer.valueOf(10);
Number n = i;
```

Conversão Implícita

```
Number n = Integer.valueOf(10);
Integer i = (Integer) n;
```

Conversão Explícita

- Essas conversões podem gerar erros em tempo de **compilação** e em tempo de **execução**

```
Integer i = Integer.valueOf(10);
Double d = (Double) i;
```

Erro de **compilação**

```
Integer i = Integer.valueOf(10);
Number n = i;
Double d = (Double) n;
```

Erro de **execução**

Polimorfismo Ad-hoc: Sobrecarga vs. Coerção

- Sobrecarga com coerção pode gerar conflito de operações

- Exemplo: para as seguintes expressões

  a) 3 + 4 b) 3.0 + 4 c) 3 + 4.0 d) 3.0 + 4.0

  – Operador + sobrecarregado quatro vezes

int x int → int **real x int → real** **int x real → real** **real x real → real**

  – Operador + sobrecarregado duas vezes

int x int → int **real x real → real**

  – Operador + sobrecarregado uma vez

real x real → real

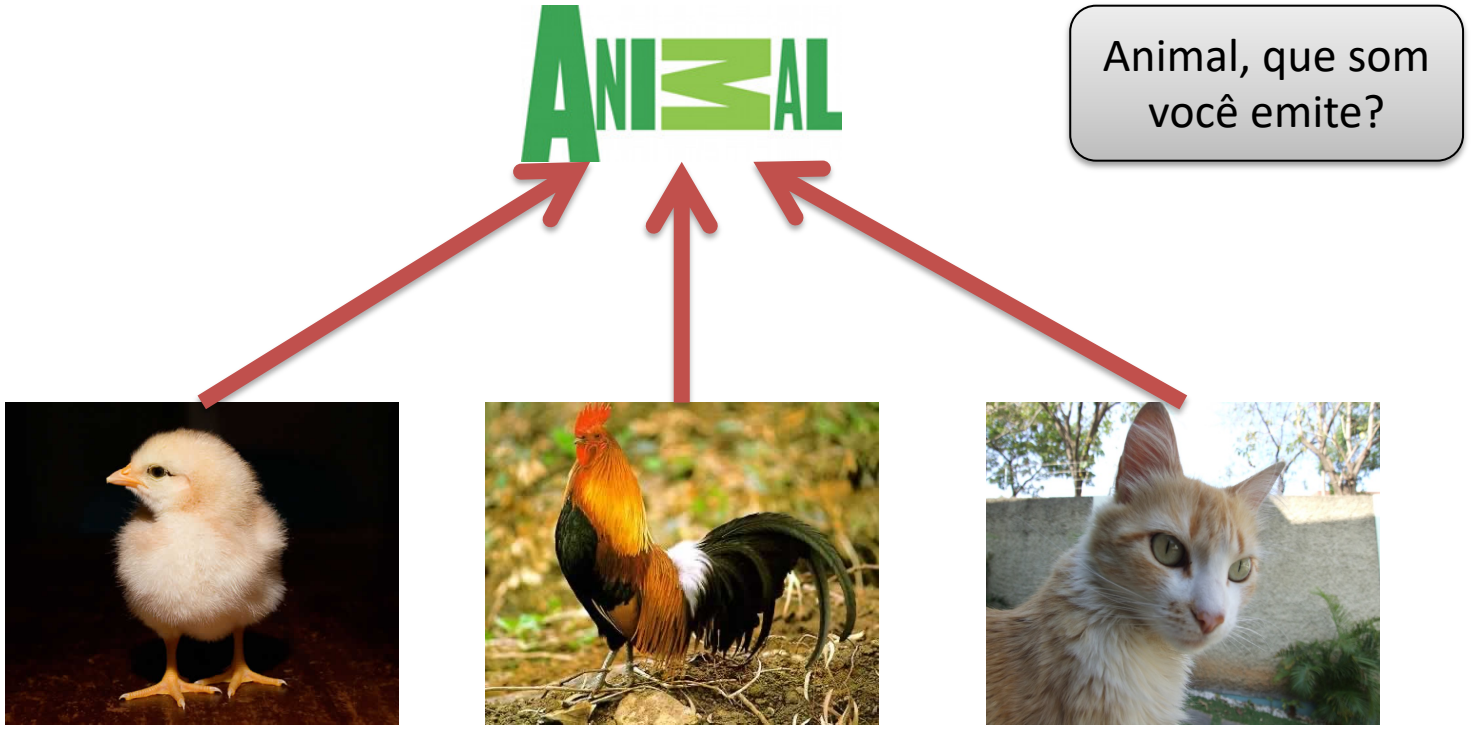
Polimorfismo Universal: Inclusão/Subtipo

- Polimorfismo universal de inclusão pode ser expressado por: **“Uma interface, múltiplos métodos”**

- É possível projetar uma interface genérica para um grupo de atividades relacionadas
  - Reduz a complexidade, pois permite que a mesma interface seja utilizada para especificar uma classe de ações

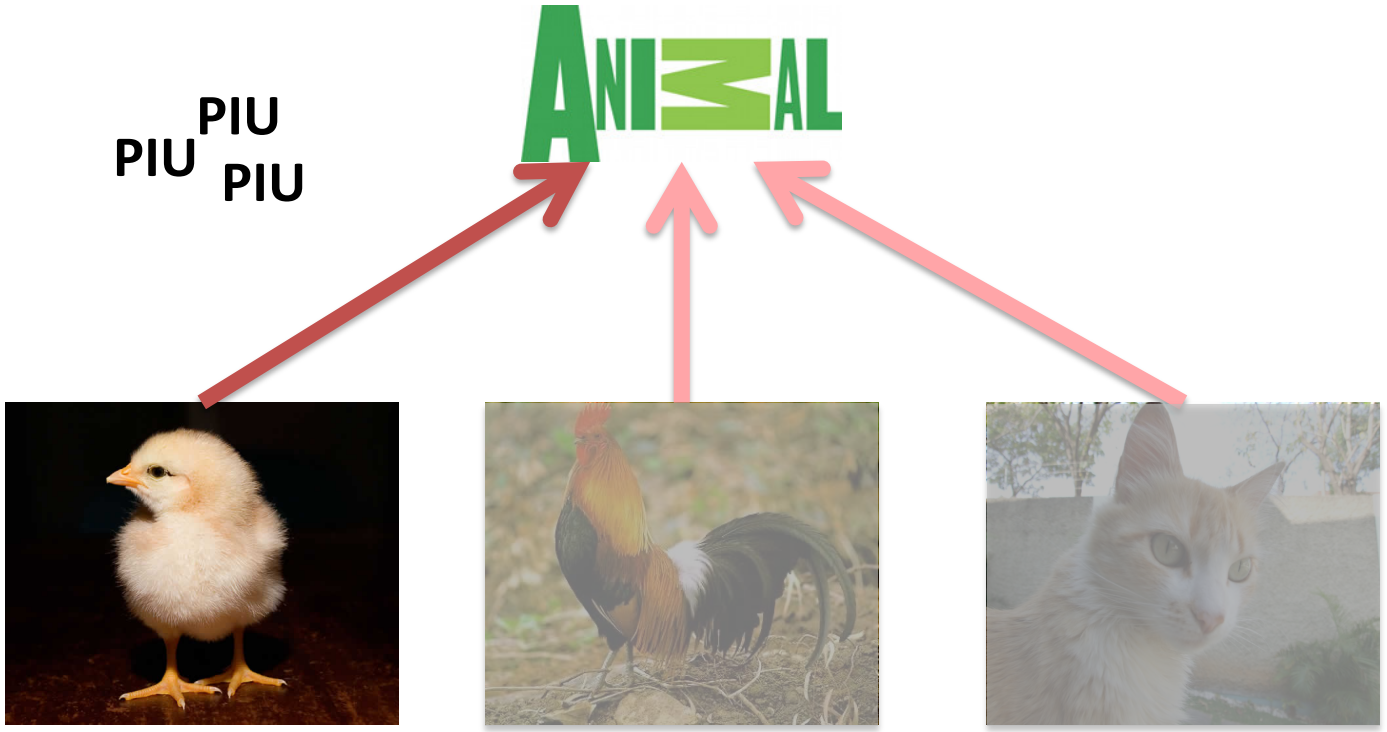
Polimorfismo Universal: Inclusão/Subtipo

- Exemplo de uma interface (som emitido) e múltiplos métodos (um para cada animal)



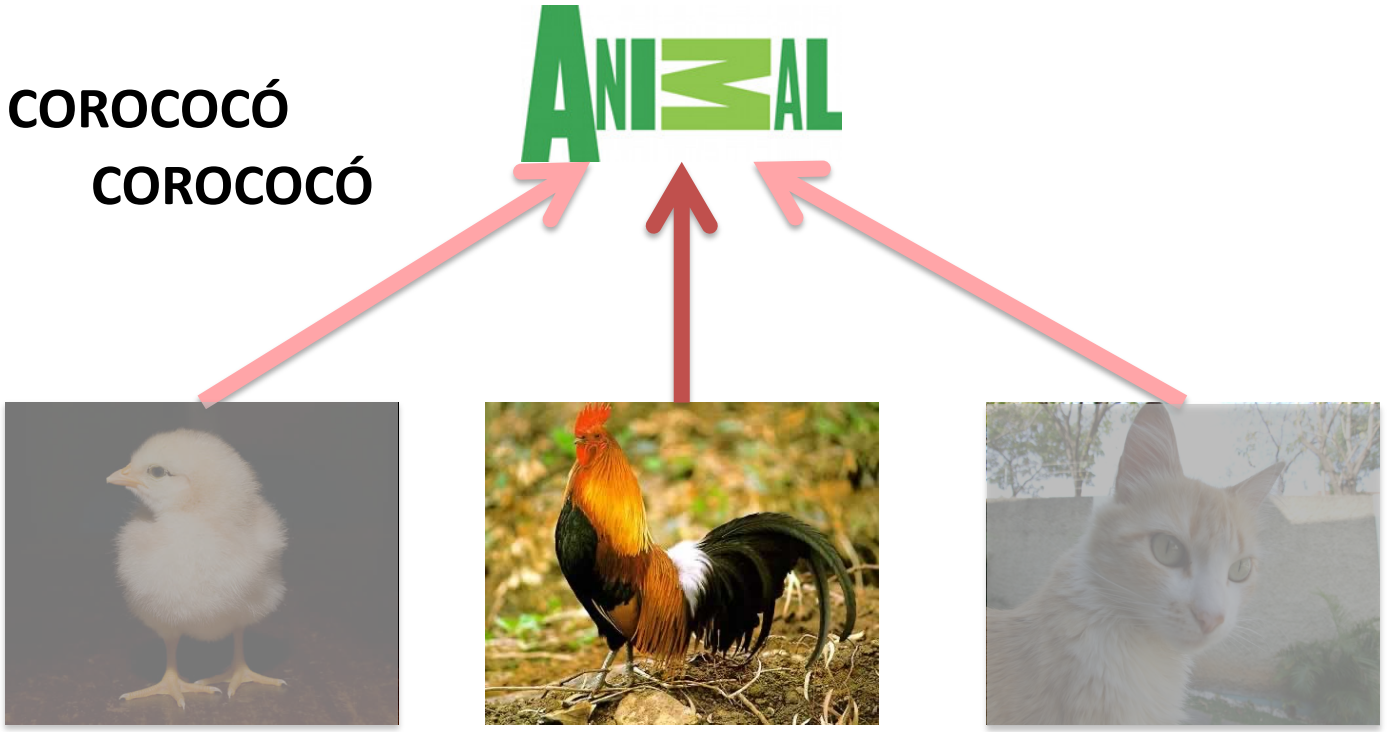
Polimorfismo Universal: Inclusão/Subtipo

- Exemplo de uma interface (som emitido) e múltiplos métodos (um para cada animal)



Polimorfismo Universal: Inclusão/Subtipo

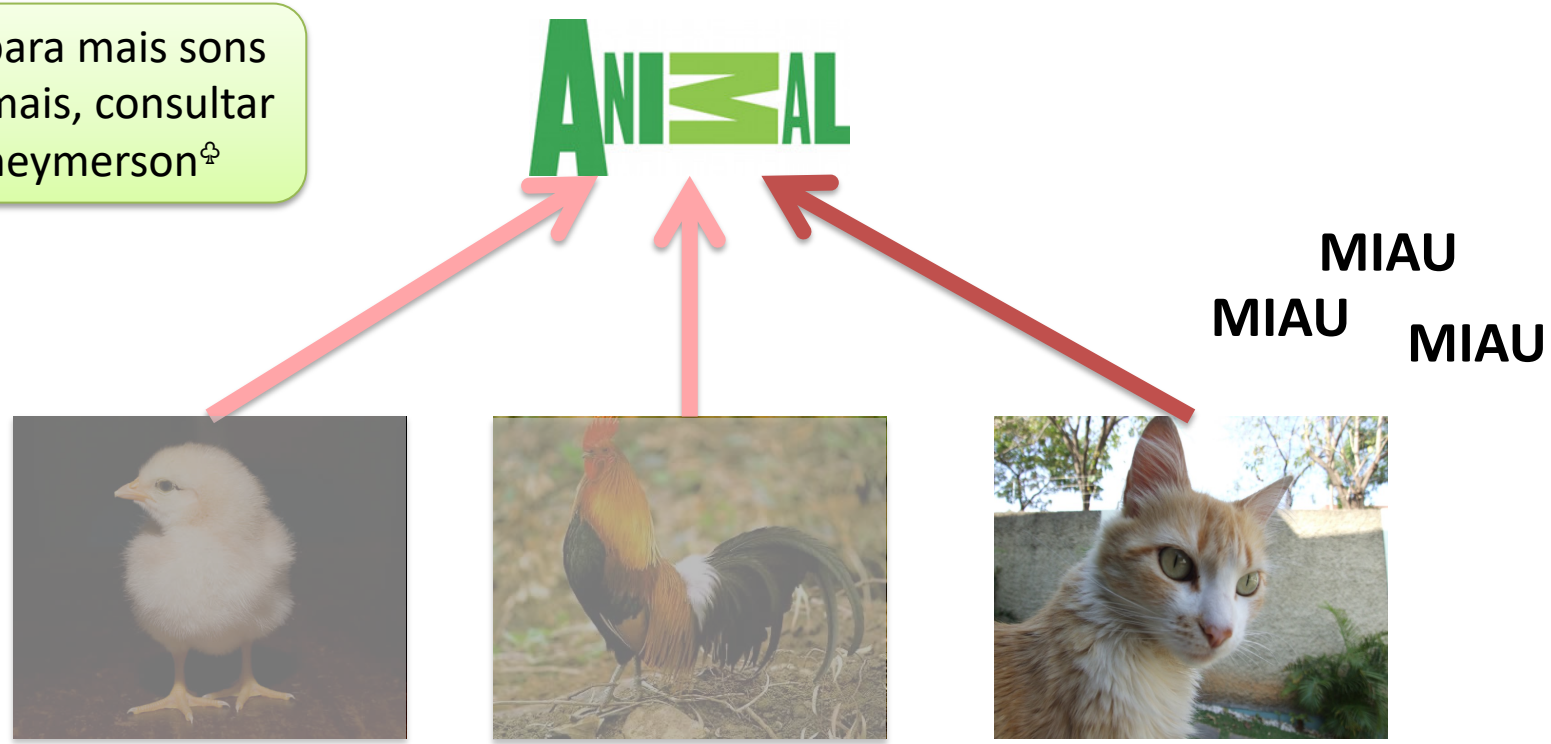
- Exemplo de uma interface (som emitido) e múltiplos métodos (um para cada animal)



Polimorfismo Universal: Inclusão/Subtipo

- Exemplo de uma interface (som emitido) e múltiplos métodos (um para cada animal)

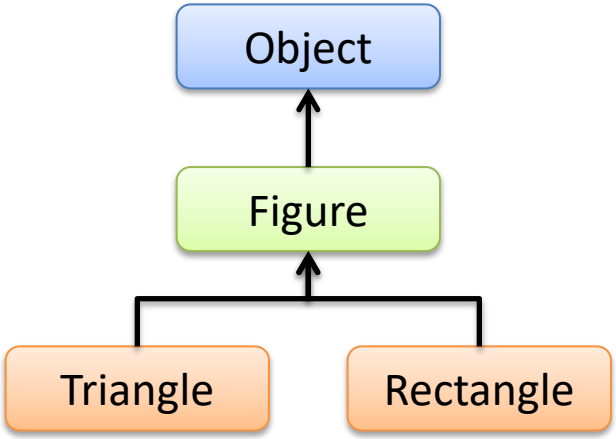
Dica: para mais sons de animais, consultar o Dheymerson♧



♧: E o pintinho piu: https://youtu.be/3pIvIvx5nSg

Polimorfismo Universal: Inclusão/Subtipo

- Exemplo em Java: implementação para o cálculo de área de figuras geométricas



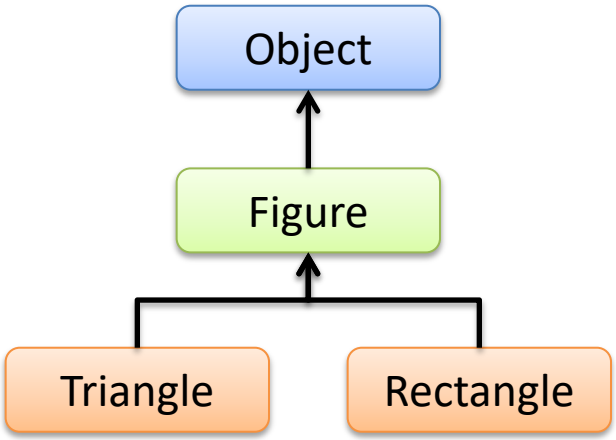

```
class Figure {
    double dim1, dim2;
    Figure(double a, double b) {
        dim1 = a; dim2 = b;
    }
    double area() {
        return 0;
    }
}
```

```
class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }
    double area() {
        return dim1 * dim2 / 2;
    }
}
```

```
class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }
    double area() {
        return dim1 * dim2;
    }
}
```

Polimorfismo Universal: Inclusão/Subtipo

- Exemplo em Java: implementação para o cálculo de área de figuras geométricas




```
class FindAreas {
    public static void main(String args[]) {
        Triangle t = new Triangle(10, 8);
        Rectangle r = new Rectangle(9, 5);
        Figure fig;

        fig = t;
        System.out.println("Area: " + fig.area());

        fig = r;
        System.out.println("Area: " + fig.area());
    }
}
```

Polimorfismo Universal: Paramétrico

- Funções que operam da mesma forma sobre objetos de tipos diferentes (Funções genéricas)

- Exemplo: o algoritmo que implementa a estrutura de dados **pilha** (o último que entra é o primeiro que sai) é o mesmo independente do tipo armazenado

Polimorfismo Universal: Paramétrico

- Exemplo de implementação de uma classe pilha genérica em C++ para armazenar valores de qualquer tipo

```
template<class T>
class stack {
private:
     T* s;
     int top;

public:
     stack(int size) {
          s = new T [size];
          top = -1;
     }
     ~stack() {
          delete [] s;
     }
     void push(T c) {
          s[++top] = c;
     }
     T pop() {
          return s[top--];
     }
};
```

```
stack<int> fibo(100);
fibo.push(1);
fibo.push(2);
fibo.push(3);
fibo.push(5);
fibo.push(8);
```

```
stack<string> msg(100);
msg.push("i");
msg.push("love");
msg.push("programming");
msg.push("languages");
```

Polimorfismo Universal: Paramétrico

- Exemplo de um método genérico em C++ para trocar valores de qualquer tipo

```
template <typename T>
void swap(T& t1, T& t2) {
    T tmp = t1;
    t1 = t2;
    t2 = tmp;
}
```

```
int n[] = { 1, 2, 3 };
swap<int>(n[1], n[2]);
swap<int>(n[0], n[1]);
swap<int>(n[1], n[2]);

std::cout << n[0] << " "
          << n[1] << " "
          << n[2] << std::endl;
```

RESUMO

Resumo

- **Herança/Composição**
  - Permitir a reutilização de código

- **Encapsulamento**
  - Migrar a implementação sem quebrar o código que depende da interface pública

- **Polimorfismo**
  - Criar códigos limpos, sensíveis, legíveis e resistentes

Analogia

- O que programação orientada a objetos tem a ver com um carro?

  Herança/Composição: motoristas são capazes de dirigir qualquer tipo de veículo (subclasses), como um sedan, micro-ônibus, minivan ou um esportivo

  - Operam o acelerador, marcha e freio do mesmo jeito

  Encapsulamento: o freio e o acelerador possuem uma complexidade incrível, ainda assim são operados com o pé, independente do tipo de motor, estilo de freio e tamanho das rodas

  Polimorfismo: fabricantes podem oferecer diversos opcionais, como freio ABS, direção hidráulica, motor de 16V

ISSO É TUDO, PESSOAL!