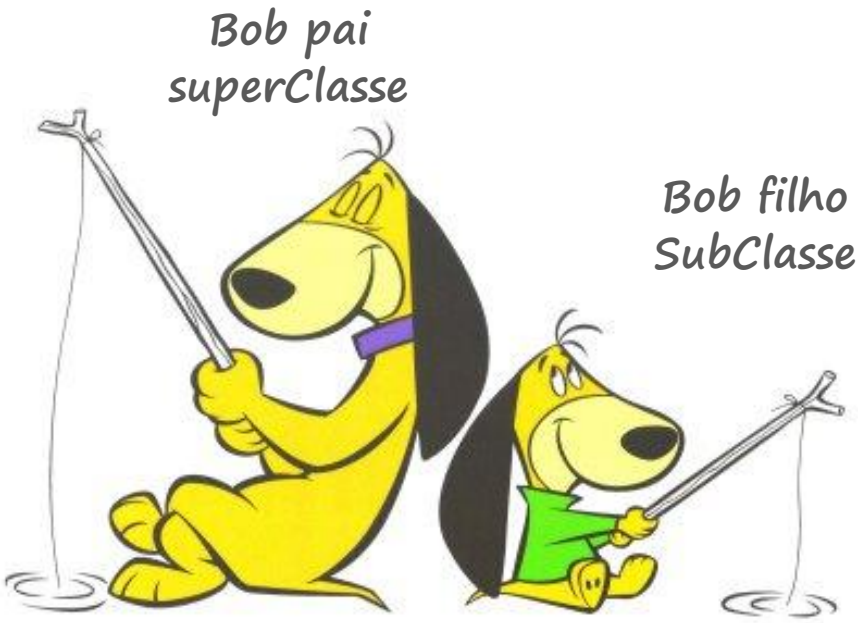
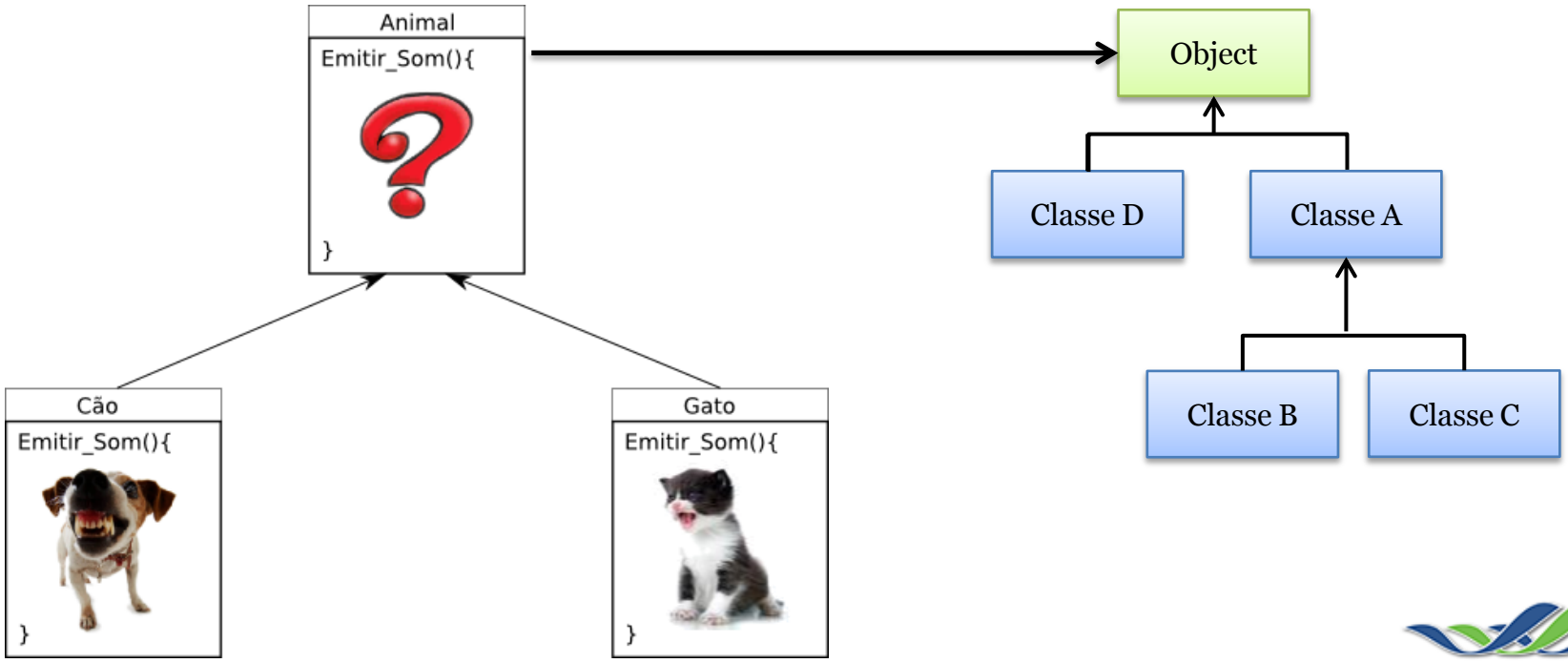
Java Orientado a Objetos
Herança e Polimorfismo



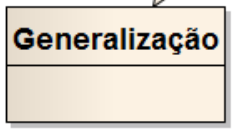
Herança

*Em Java, todas as classes, incluindo as que formam a API Java, são **subclasses** da classe **Object***



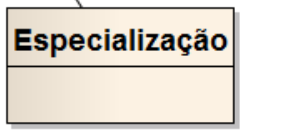
SuperClasse e SubClasse

Qualquer classe acima de uma classe específica na hierarquia de classes



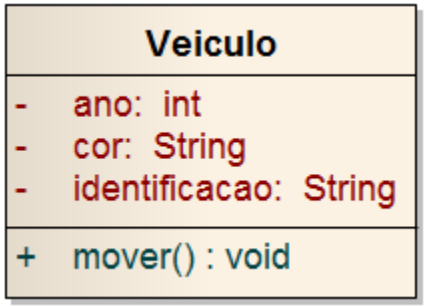
Permite codificar um método apenas uma única vez e este pode ser usado por todas as subclasses

Qualquer classe abaixo de uma classe específica na hierarquia de classes



Uma subclasse necessita apenas implementar as diferenças entre ela própria e sua classe pai.

Herança – classe Veiculo

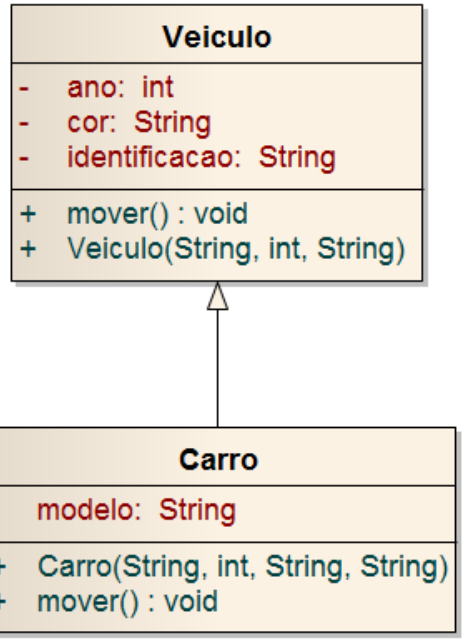



```java
public class Veiculo {// nome da classe

    // (1)Atributos - Variáveis
    private String cor;
    private int ano;
    private String identificacao;

    // (2)Construtor
    public Veiculo( String cor, int ano, String identificacao ) {

        this.cor = cor;
        this.ano = ano;
        this.identificacao = identificacao;
        System.out.println("Criando objeto Veiculo");
    }

    // (3)Métodos
    public void mover() {
        System.out.println("Veiculo se movendo");
    }

}
```

Herança – classe Carro

```java
public class Carro extends Veiculo { // nome da classe

    // (1)Atributos - Variáveis
    private String modelo;

    // (2)Construtor
    public Carro( String cor, int ano, String placaIdentificacao, String modelo ) {

        super(cor, ano, placaIdentificacao);
        this.modelo = modelo;

        System.out.println("Criando objeto Carro");
    }

    @Override
    public void mover() {
        System.out.println("Correr");
    }

}
```

Uma chamada a um construtor super() no construtor de uma subclasse resultará na execução do construtor referente da superclasse, baseado nos argumentos passados.



Modificador de Classe *final*

Classes que não podem ter subclasses

<modificador>* final class <nomeClasse> { ... }

Muitas classes na API Java são declaradas **final** *para certificar que seu comportamento não seja herdado e, possivelmente modificado. Exemplos, são as classes* **Integer**, **Double**, **Math** *e* **String**

Polimorfismo

Sobreposição de Métodos @Override

```java
public class Veiculo {// nome da classe
    // ... //
    public void mover() {
        System.out.println("Veiculo se movendo");
    }
    // ... //
}
```

```java
public class Barco extends Veiculo {
    // ... //
    @Override
    public void mover() {
        System.out.println("Navegar");
    }
    // ... //
}
```

```java
public class Aviao extends Veiculo {
    // ... //
    @Override
    public void mover() {
        System.out.println("Voar");
    }
    // ... //
}
```

```java
public class Carro extends Veiculo {
    // ... //
    @Override
    public void mover() {
        System.out.println("Correr");
    }
    // ... //
}
```

*O tipo de **retorno** do método na **subclasse** deve ser **idêntico** ao do **método sobreposto** na **superclasse**.*

Referências polimórficas

Referência

Instância

```
public static void main(String[] args) {

    Veiculo veiculo = new Carro("Cinza", 2012, "NWP-2552", "Gol");
    veiculo.mover();
    veiculo = new Aviao("Prata", 2013, "NWP-2552");
    veiculo.mover();
    veiculo = new Barco("Branco", 2008, "NWP-2552");
    veiculo.mover();
}
```

Tem que correr

Tem que voar

Tem que navegar

Uma variável de referência de uma classe mais genérica (superclasse) pode receber referência de objetos de classes mais especializadas (as subclasses).

Coleções Heterogêneas de Objetos

```
Carro [] carros = new Carro [3];
// coleção de carros
carros[0] = new Carro("Cinza", 2012, "NWP-2552", "Gol");
carros[1] = new Carro("Preto", 1995, "PAX-4589", "Pálio");
carros[2] = new Carro("Vermelho", 2000, "XPY-3895", "Celta");

Barco [] barcos = new Barco [2];
// coleção de Barco
barcos[0] = new Barco("Verde", 1999, "Naúfrago");
barcos[1] = new Barco("Preto", 1312, "Pérola Negra");

// criar coleção
Veiculo [] veiculo = new Veiculo [4];
// atribui referência a coleção
veiculo[0] = new Carro("Cinza", 2012, "NWP-2552", "Gol");
veiculo[1] = new Barco("Preto", 1312, "Pérola Negra");
veiculo[2] = new Carro("Preto", 1995, "PAX-4589", "Pálio");
veiculo[3] = new Aviao("Branco", 2010, "Boing 737");
```

O recurso do polimorfismo nos permite criar um único array para nossa coleção de animais.

Determinando a Classe de um Objeto

```
// criar coleção
Veiculo [] veiculo = new Veiculo [4];
// atribui referência a coleção
veiculo[0] = new Carro("Cinza", 2012, "NWP-2552", "Gol");
veiculo[1] = new Barco("Preto", 1312, "Pérola Negra");
veiculo[2] = new Carro("Preto", 1995, "PAX-4589", "Pálio");
veiculo[3] = new Aviao("Branco", 2010, "Boing 737");
```

Carro

```
System.out.println(veiculo[0].getClass().getSimpleName());
System.out.println(veiculo[1].getClass().getSimpleName());
```

Barco

```
System.out.println(veiculo[0] instanceof Carro);
System.out.println(veiculo[1] instanceof Barco);
```

Retorna true se retornar objeto do tipo Especificado

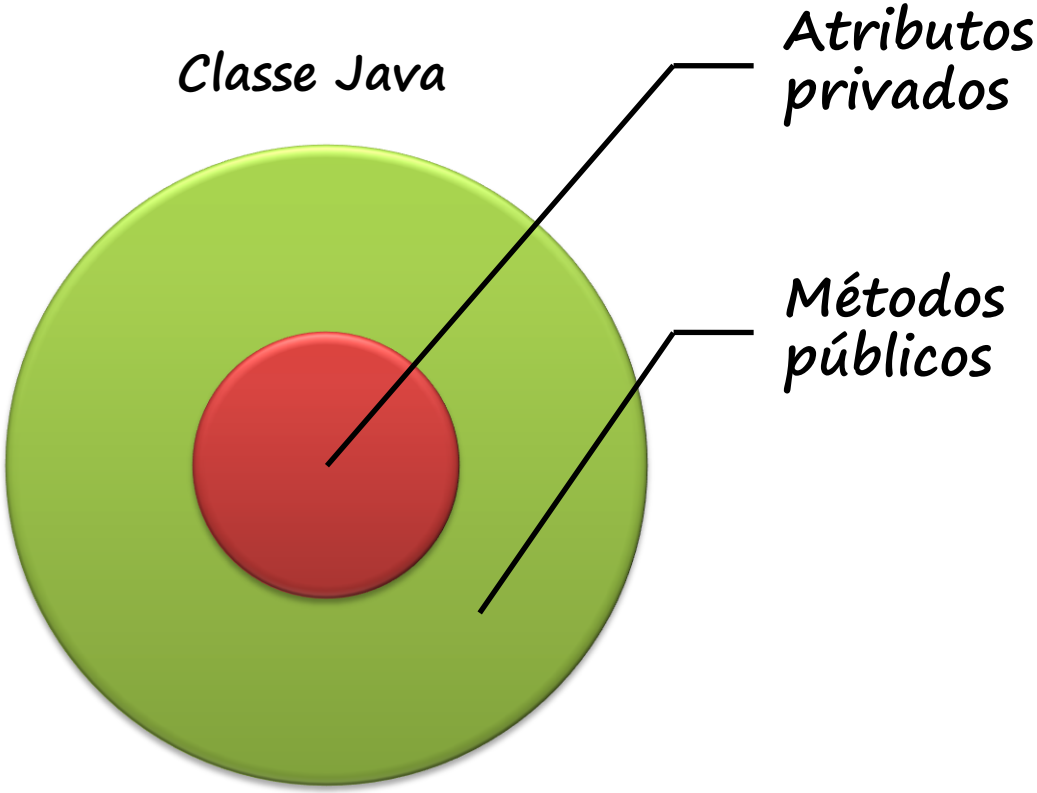
*Um dos problemas que enfrentaremos ao lidar com **referências genéricas** para **objetos** de **subclasses**, é que não sabemos mais a qual classe **pertence a referência** armazenada.*

Modificador de método *final*

Impede o polimorfismo das subclasses por override

<modificador> final <tipo retorno> <nome>(<parâmetros>)<clausula throws> { ... }

Encapsulamento



Para implementar o encapsulamento, temos os **modificadores de acesso**

Métodos de Configuração e Captura

```
private String cor;
```

```
public void setCor(String cor) {

    this.cor = cor;
}
```

possibilita ***alteração*** *dos valores (variáveis) por outros objetos.*

set<NomeAtributo>(<tipo dado> <parâmetro>)

```
public String getCor() {

    return cor;
}
```

usados para ***ler valores*** *de atributos.*

get<NomeDoAtributo>.