UFMS DIGITAL

Programação Orientada a Objetos

Prof. Dr. Anderson V. de Araujo

UFMS DIGITAL

Módulo III - Pilares da programação orientada a objetos

Unidade I - Herança e Polimorfismo

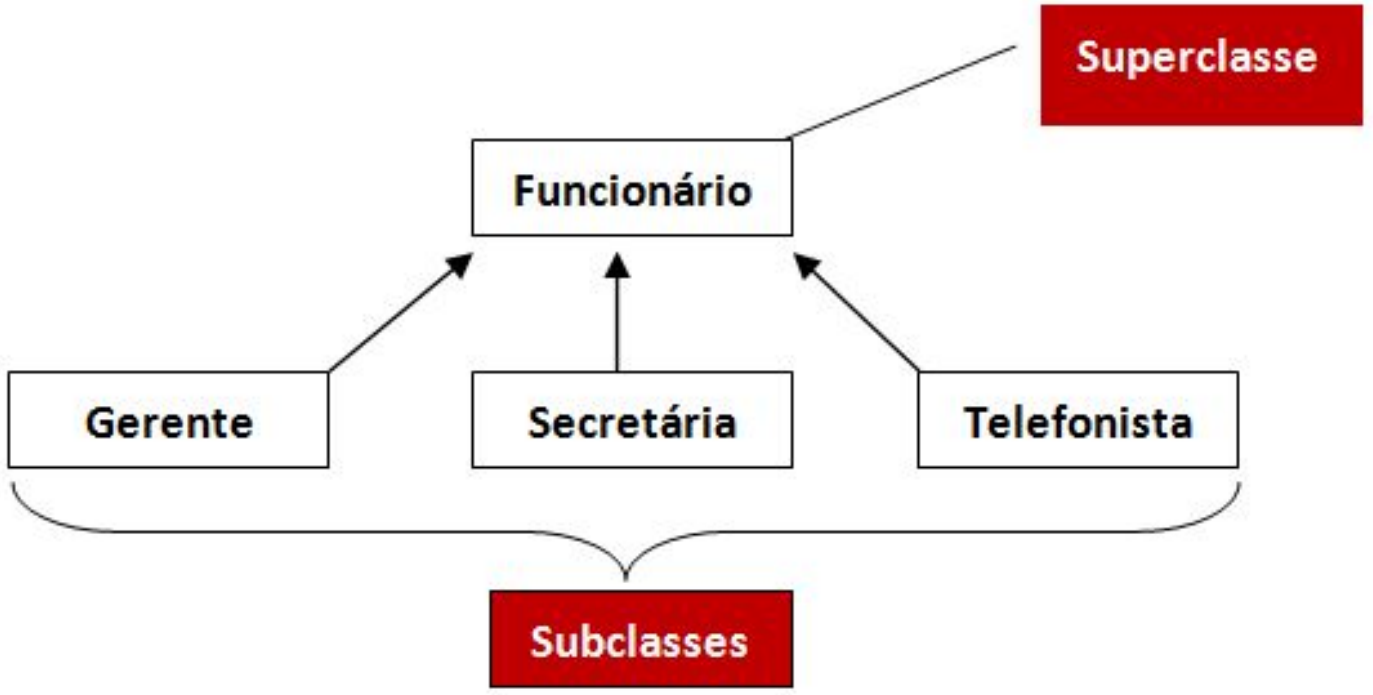
Herança

- A habilidade de criar uma nova classe a partir de uma classe existente;
- É uma forma de reutilização de código na qual uma nova classe é criada, absorvendo membros de uma classe existente:
  - Podendo ser aprimorada com novos membros ou modificar os existentes;
- Economiza tempo no desenvolvimento;
- Aumenta a qualidade do código.

Herança (2)

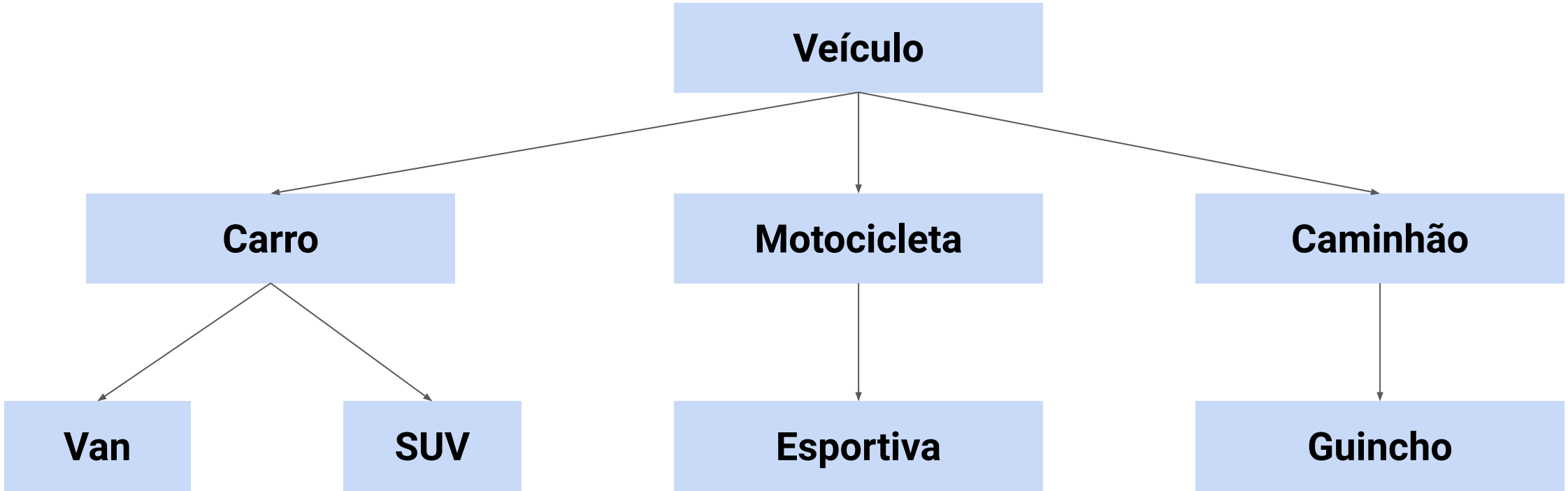
- Ao declarar uma classe, em Java é possível definir UMA classe para herdar seus membros internos;
- A classe existente é chamada de **superclasse** (ou classe pai):
  - Superclasse direta: Primeira na hierarquia;
  - Superclasse indireta: Qualquer outra classe na hierarquia (classe pai da classe pai, ...).
- A nova classe sendo criada é chamada de **subclasse** (ou classe filha):
  - A subclasse é mais específica que a superclasse e representa um grupo mais especializado de objetos

Herança – Exemplo



Fonte: O próprio autor.

Herança - Exemplo (2)



Fonte: O próprio autor.

Herança - Sintaxe

- **Superclasse**

```
class Esporte{
        String nome;
}
```

- **Subclasse**
  - A cláusula opcional extends indica a superclasse de onde os membros vão ser herdados

```
class EsporteAquatico extends Esporte {}
```

```
public class CellPhone {
    public void print() {
        System.out.println("I'm a cellphone");
    }
}

public class TouchPhone extends CellPhone {
    public void printTouch() {
        super.print();
        System.out.println("I'm a touch screen cellphone");
    }

    public static void main (String[] args) {
        TouchPhone p = new TouchPhone();
        p.printTouch();
    }
}
```

A Classe Object

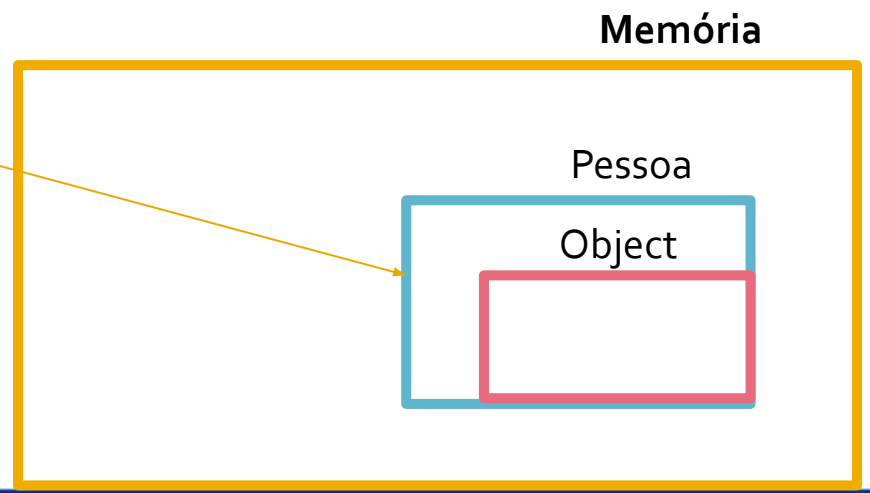
- A classe Object é a classe pai de todos as classes definidas
  - Mesmo que você não tenha definido isso (através da palavra reservada `extends`).
- Ou seja, todas as classes que criar serão filhas de Object
  - Com isso, a sua classe tem acesso aos membros internos visíveis da classe `Object;`
  - Inclusive vetores.

Métodos da Classe Object

- `boolean equals(Object obj)`
- `void finalize() throws Throwable`
- `int hashCode()`
- `String toString()`
- Etc...

```
• Object o1 = new Object();
    //Uma variável de referência do tipo Object pode

    //referenciar qualquer outro tipo
• Object o2 = new String();
• Object o3 = new Pessoa();
```

Neste exemplo a variável de referência `o3` só irá conseguir *enxergar* a parte referente a classe *`Object`* da instância criada



Polimorfismo

- Poli = Muitas; Morfo = Formas
- É a capacidade de um objeto poder ter muitas formas
- Está fortemente associada à herança
- A subclasse pode definir seu próprio comportamento único e ainda compartilhar as mesmas funcionalidades de seu pai
- Mas uma classe pai não pode ter o comportamento da sua subclasse

Polimorfismo - Tipos

- Funciona somente sobre métodos
  - Não serve para atributos
- Tipos:
  - Sobrecarga
  - Sobrescrita

Polimorfismo - Sobrecarga

- Ter vários métodos dentro de uma classe com o mesmo nome, mas com **assinaturas diferentes**
- Também chamado de *compile time polymorphism* ou *static binding*
- É possível:
  - Ter um número diferente de parâmetros
  - Ter tipos diferentes para os parâmetros
- Não é possível:
  - Ter somente o tipo de retorno diferente*
    - Claro, o tipo de retorno não faz parte da assinatura do método
  - Mudar apenas os nomes dos parâmetros
    - Se for de pai pra filho, vira sobrescrita

Polimorfismo – Sobrecarga - Exemplos

```
//compiler error - can't overload based on the type returned

//(one method returns int, the other returns a float)

int changeDate(int year) {...};

float changeDate(int year) {...};


//compiler error - can't overload by changing just

//the name of the parameter (from year to month)

int changeDate(int year) {...};

int changeDate(int month) {...};
```

```
//valid case of overloading, since the methods

//have different number of parameters

int changeDate(int year, int month) {...};

int changeDate(int year) {...};

//also a valid case of overloading, since the

//parameters are of different types

int changeDate(float year) {...};

int changeDate(int year) {...};

//also a valid case of overloading, since the

//parameters AND return type are different

int changeDate(float year) {...};

double changeDate(double var) {...};
```

Polimorfismo - Sobrescrita

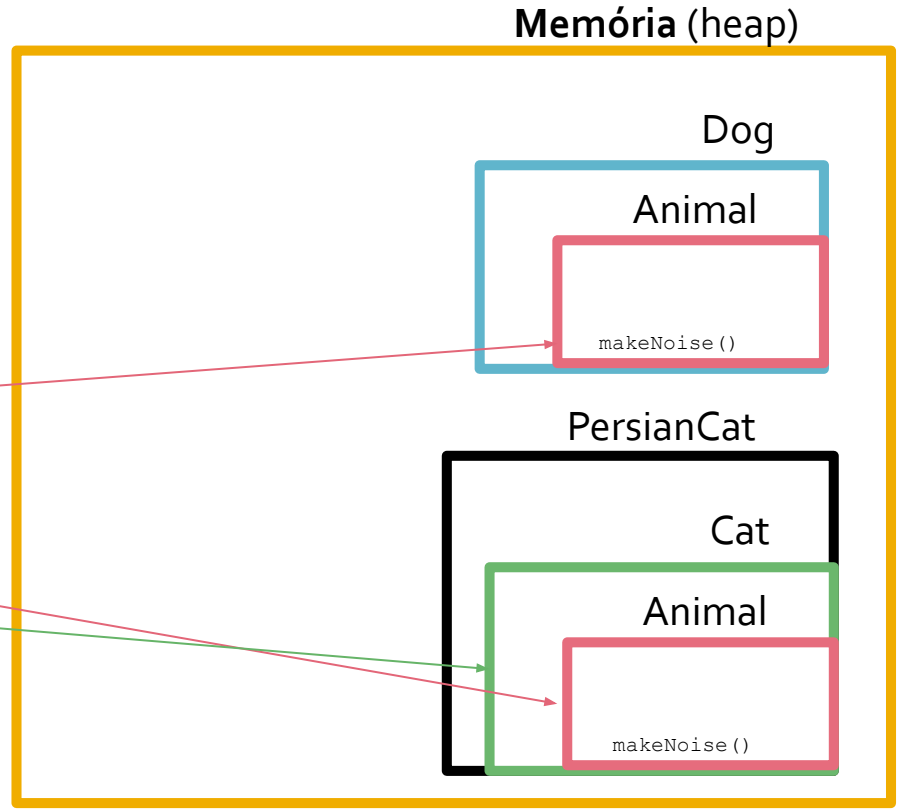
- Provê uma nova implementação para um **método existente** e visível da classe pai
- Depende da Herança
- Também chamado de *Runtime polymorphism* ou *dynamic binding*

```
class Animal {
    public void makeNoise(){
        System.out.println("Some sound");
    }
}

class Dog extends Animal{
    public void makeNoise(){
        System.out.println("Bark");
    }
}

class Cat extends Animal{
    public void makeNoise(){
        System.out.println("Meawoo");
    }
}
```

Polimorfismo – Exemplo Sobrescrita e Super


```
public class CellPhone {
    public void print() {
        System.out.println("I'm a cellphone");
    }
}

public class TouchPhone extends CellPhone {
    public void print() {
        super.print();
        System.out.println("I'm a touch screen cellphone");
    }

    public static void main (String[] args) {
        TouchPhone p = new TouchPhone();
        p.print();
    }
}
```

```
class PersianCat extends Cat{

    public void makeNoise(){

        sysout("Meawooooooo");

    }

}

class Test{

   public static void main(String[] args){
        Animal an1 = new Dog();
        Animal an2 = new PersianCat();
        Cat cat = (Cat)an2;

        an1.makeNoise();
        cat.makeNoise();
   }

}
```



- As variáveis de referência **an1** e **an2** apontam para os campos **visíveis** da classe **Animal** para a instância criada (Dog)
- A variável de referência **cat** aponta para os campos **visíveis** da classe **Cat** para a instância (PersianCat) criada

Licenciamento