Departamento de Engenharia de Software
Centro de Educação Superior do Alto Vale do Itajaí - CEAVI
Universidade do Estado de Santa Catarina - UDESC

Programação orientada a objetos
Herança e polimorfismo

Relacionamentos entre classes

- Classes podem se relacionar entre si, definindo um vínculo entre os objetos dessas classes.

Exemplos

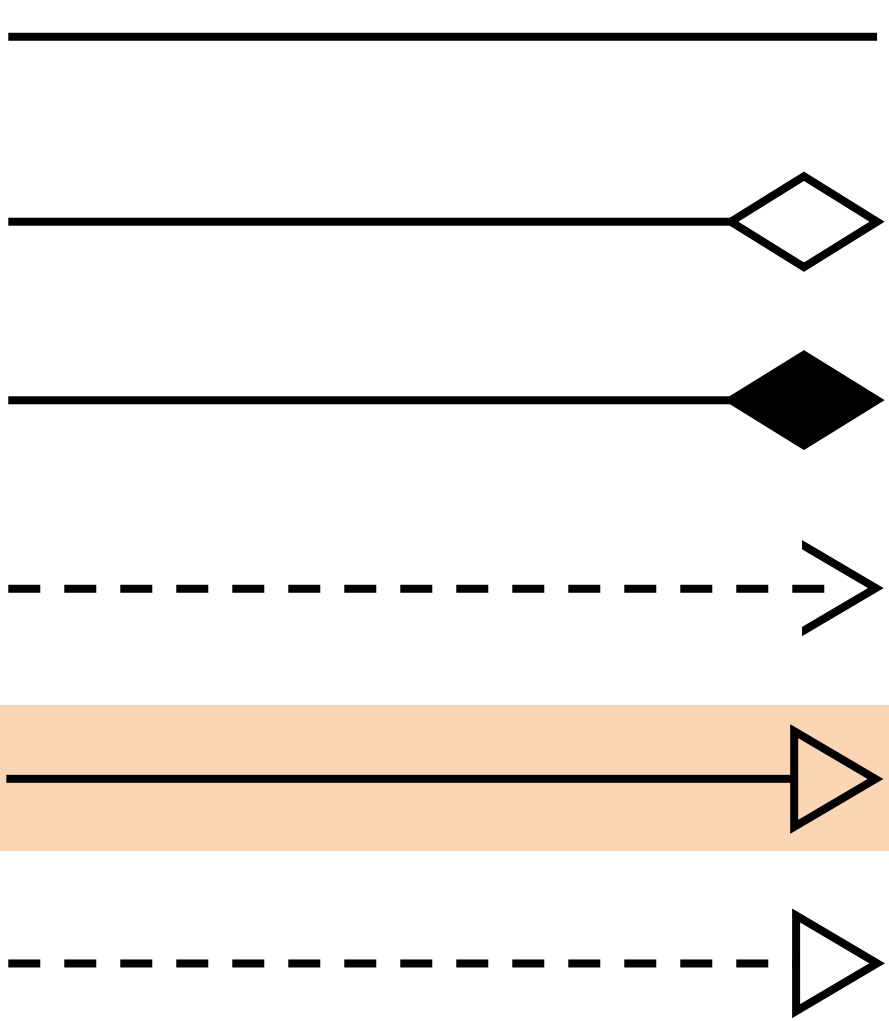
- Um **cliente** possui um **endereço**.
- Uma **empresa** é composta por **funcionários**.
- Uma **moto** é um tipo de **veículo**.
- Um **restaurante** possui **pratos**.
- Uma **correspondência** possui um **remetente** e um **destinatário**.

Relacionamentos entre classes

- **Associação:** conexão entre classes.
- **Agregação e composição:** especialização de uma associação onde um todo é relacionado com suas partes (relacionamento "parte-de").
- **Dependência:** um objeto depende de alguma forma de outro (relacionamento de utilização).
- **Herança (generalização):** um dos princípios da orientação a objetos, permite a reutilização, uma nova classe pode ser definida a partir de outra já existente.
- **Realização:** um contrato que a classe segue (obrigação).

Relacionamentos entre classes

- **Associação:**
- **Agregação**
- **Composição:**
- **Dependência:**
- **Herança (generalização):**
- **Realização:**



Herança

- A herança permite definir elementos **específicos**, que incorporam a estrutura (atributos) e o comportamento (operações) de elementos mais **gerais**. Neste sentido, a classe específica herda a estrutura e o comportamento da classe geral, definindo uma hierarquia entre elas.
  - Por isso, também é chamada de especialização ou generalização.
- A herança permite reduzir a reescrita (redundância) de código e traz flexibilidade e manutenibilidade ao projeto.

Herança

- Imagine uma empresa que possui funcionários e gerentes. Estas entidades são modeladas pelas classes **Funcionario** e **Gerente**. Todo o funcionário da empresa possui uma matrícula e um salário, inclusive os gerentes. Porém, cada gerente possui um número de subordinados e uma senha para acesso ao sistema.

```
public class Funcionario {
    private String matricula;
    private double salario;

    //...
}
```

```
public class Gerente {
    private String matricula;
    private double salario;
    private int subordinados;
    private int senha;

    //...
}
```

Herança

- Imagine uma empresa que possui funcionários e gerentes. Estas entidades são modeladas pelas classes **Funcionario** e **Gerente**. Todo o funcionário da empresa possui uma matrícula e um salário, inclusive os gerentes. Porém, cada gerente possui um número de subordinados e uma senha para acesso ao sistema.

```
public class Funcionario {
    private String matricula;
    private double salario;

    //...
}
```

```
public class Gerente {
    private String matricula;
    private double salario;
    private int subordinados;
    private int senha;

    //...
}
```

Reescrita (redundância) de código!

Herança

- Imagine uma empresa que possui funcionários e gerentes. Estas entidades são modeladas pelas classes **Funcionario** e **Gerente**. Todo o funcionário da empresa possui uma matrícula e um salário, inclusive os gerentes. Porém, cada gerente possui um número de subordinados e uma senha para acesso ao sistema.

```
public class Funcionario {
    private String matricula;
    private double salario;

    //...
}
```

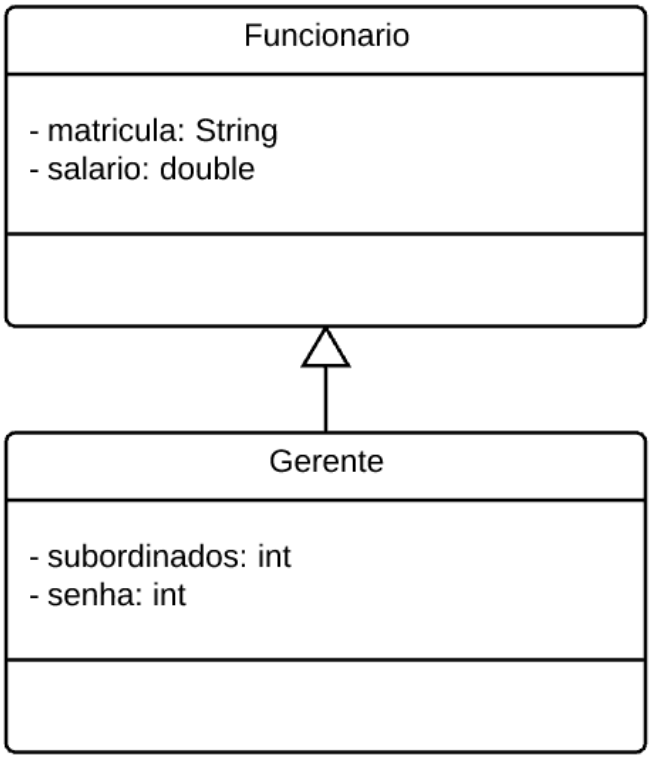
```
public class Gerente {
    private String matricula;
    private double salario;
    private int subordinados;
    private int senha;

    //...
}
```

Se forem incluídos outros tipos de funcionários (secretária, diretor, presidente)?
O código deverá ser replicado para cada uma das classes.

Se, após criados muitos funcionários, seus atributos tiverem que ser alterados?
Cada classe deverá ser alterada.

Herança

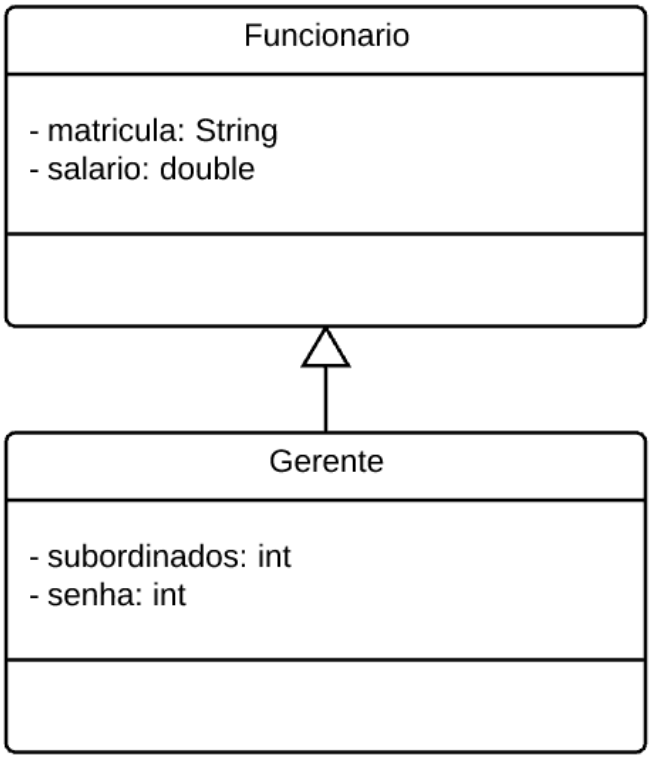
- A solução para este problema é utilizar herança, de modo que uma classe geral define a estrutura e o comportamento básicos de todos os funcionários, enquanto classes específicas herdam estas características e adicionam o necessário para cada tipo de funcionário.



A classe **Gerente** herda os atributos **matricula** e **salario** de **Funcionario,** definindo atributos adicionais (**subordinados** e **senha**).

Herança

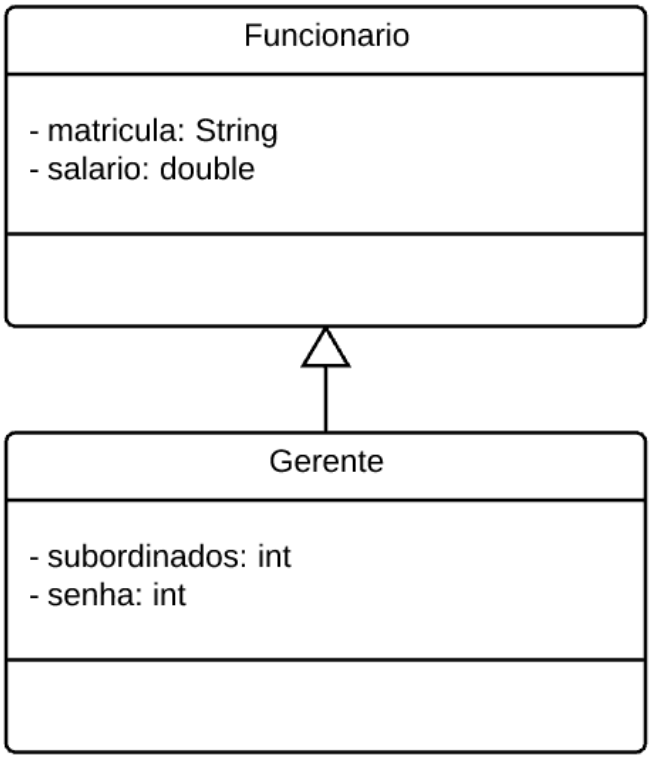
- A solução para este problema é utilizar herança, de modo que uma classe geral define a estrutura e o comportamento básicos de todos os funcionários, enquanto classes específicas herdam estas características e adicionam o necessário para cada tipo de funcionário.



A herança define um relacionamento do tipo **É UM**. Neste caso, um gerente **É UM** funcionário (ou gerente é **UM TIPO DE** funcionário).

Herança

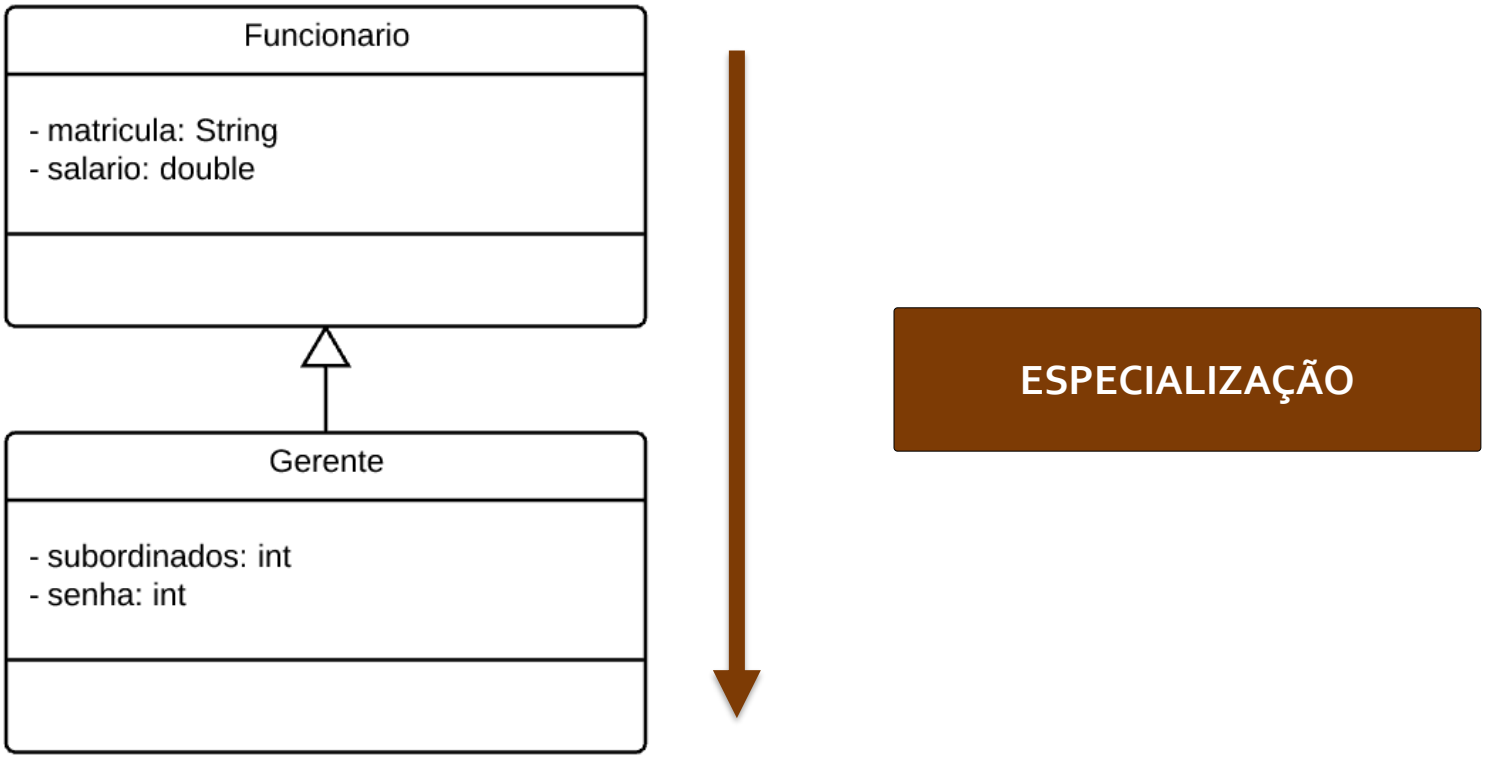
- A solução para este problema é utilizar herança, de modo que uma classe geral define a estrutura e o comportamento básicos de todos os funcionários, enquanto classes específicas herdam estas características e adicionam o necessário para cada tipo de funcionário.



Atenção: um gerente É UM funcionário, mas um funcionário **NÃO É UM** gerente (não necessariamente).

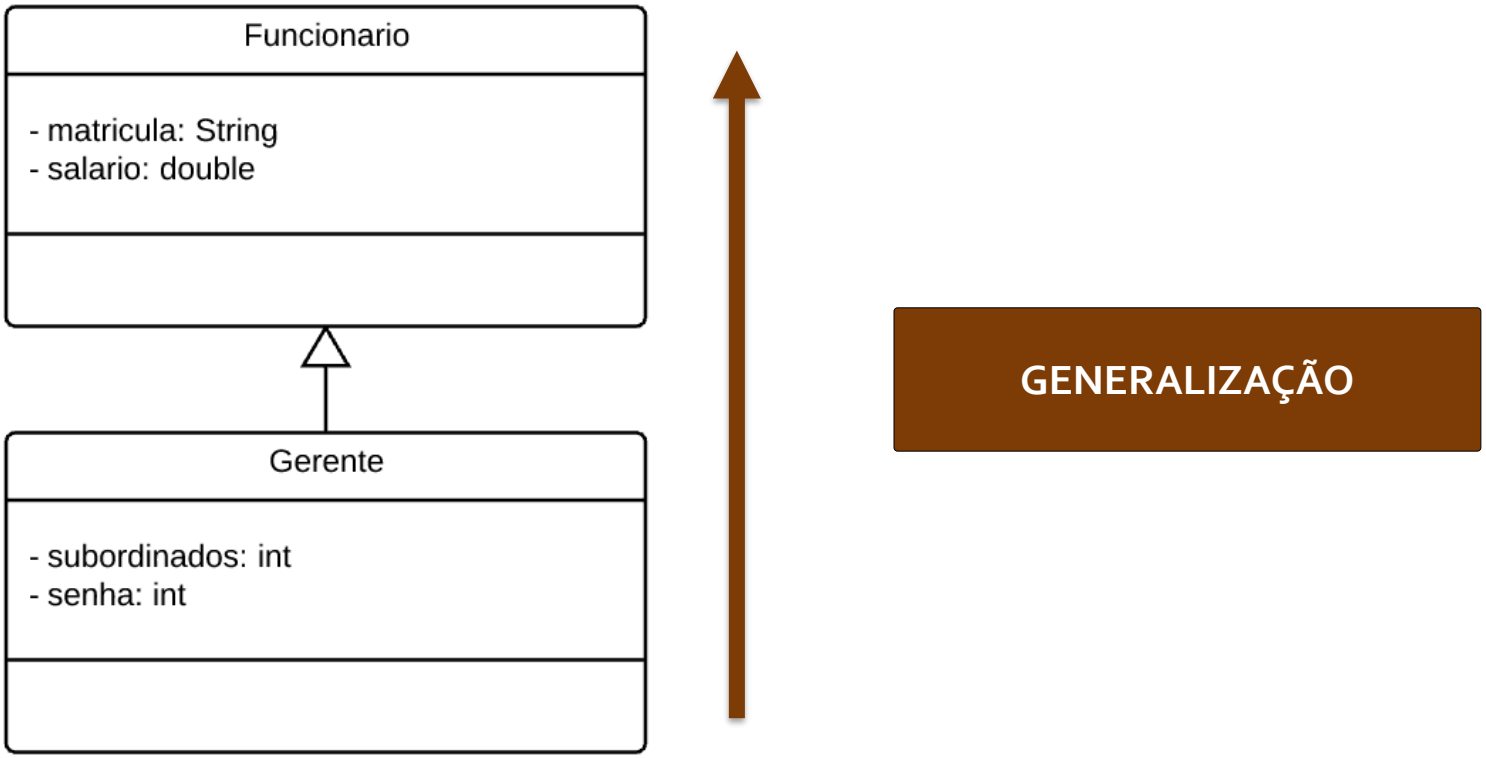
Herança

- A solução para este problema é utilizar herança, de modo que uma classe geral define a estrutura e o comportamento básicos de todos os funcionários, enquanto classes específicas herdam estas características e adicionam o necessário para cada tipo de funcionário.



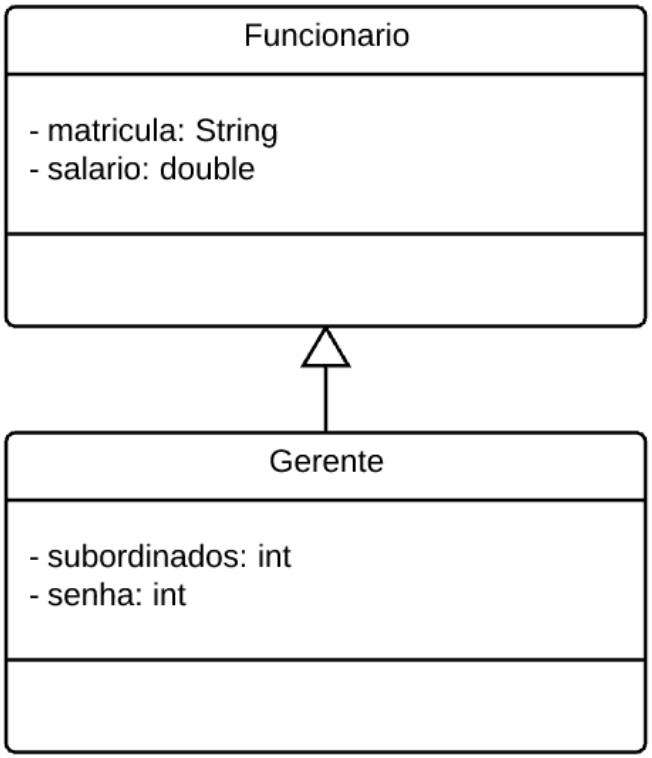
Herança

- A solução para este problema é utilizar herança, de modo que uma classe geral define a estrutura e o comportamento básicos de todos os funcionários, enquanto classes específicas herdam estas características e adicionam o necessário para cada tipo de funcionário.



Herança

- A solução para este problema é utilizar herança, de modo que uma classe geral define a estrutura e o comportamento básicos de todos os funcionários, enquanto classes específicas herdam estas características e adicionam o necessário para cada tipo de funcionário.

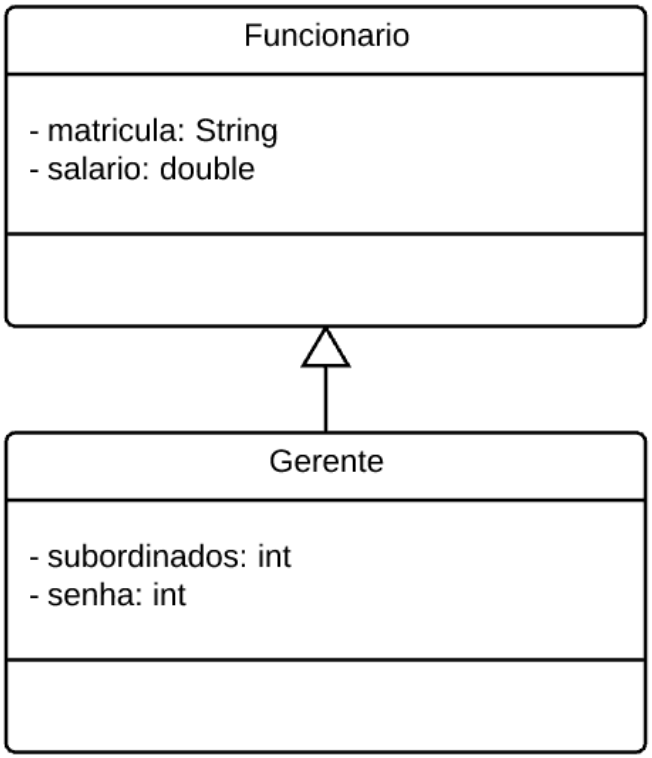


CLASSE-MÃE ou SUPERCLASSE

CLASSE-FILHA ou SUBCLASSE

Herança

- A implementação é feita utilizando a palavra **extends**, onde se define que a classe específica estende a classe geral. Neste caso, gerente estende um funcionário, pois herda suas características e adiciona características adicionais.




```java
public class Funcionario {
    private String matricula;
    private double salario;

    //...
}
```

```java
public class Gerente extends Funcionario {
    private int subordinados;
    private int senha;

    //...
}
```

Herança

- A implementação é feita utilizando a palavra **extends**, onde se define que a classe específica estende a classe geral. Neste caso, gerente estende um funcionário, pois herda suas características e adiciona características adicionais.

```
public class Funcionario {
    private String matricula;
    private double salario;

    //...
}
```

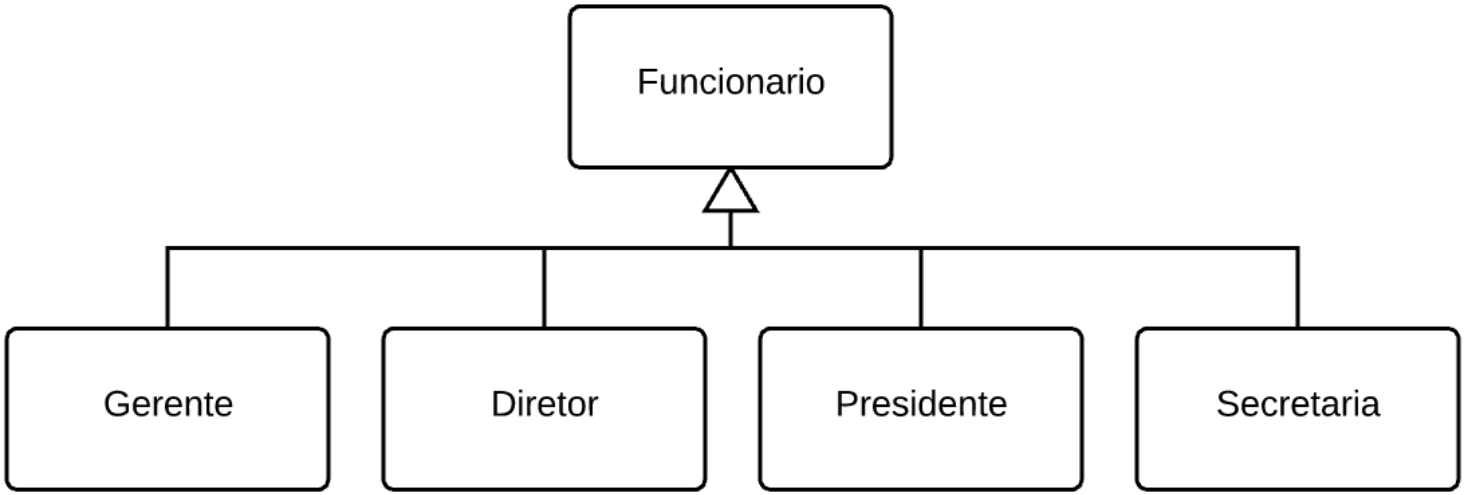
```
public class Gerente extends Funcionario {
    private int subordinados;
    private int senha;

    //...
}
```

```
public static void main(String[] args) {
    Gerente g = new Gerente();
    g.setMatricula("123456");
    g.setSalario(4500);
    g.setSubordinados(10);
    g.setSenha(1234);
}
```

Um objeto da classe **Gerente** tem acesso a tudo o que for público na classe **Funcionario**. Porém, um objeto da classe **Funcionario** não herda nada da classe **Gerente**.

Herança

- A estrutura pode crescer, incluindo diferentes tipos de funcionários. Todos eles herdam as características gerais da classe **Funcionario** e definem suas características específicas.



Herança – sobrescrita de métodos

- **Exemplo:** todo o funcionário da empresa tem direito a uma gratificação de natal, que consiste em 50% do seu salário. Logo, podemos implementar o método que determina a gratificação da classe **Funcionario**, pois todos os funcionários tem direito a ela.

```
public class Funcionario {
    private String matricula;
    private double salario;

    public double gratificacao() {
        return this.salario * 0.5;
    }

    //...
}
```

```
public class Gerente extends Funcionario {
    private int subordinados;
    private int senha;

    //...
}
```

```
public static void main(String[] args) {
    Gerente g = new Gerente();
    g.setMatricula("123456");
    g.setSalario(4500);
    g.setSubordinados(10);
    g.setSenha(1234);
    System.out.println(g.gratificacao()); //Imprimirá 2250.0
}
```

Herança – sobrescrita de métodos

- **Exemplo:** se a gratificação concedida aos gerentes da empresa for diferente da gratificação concedida aos demais funcionários (75%, por exemplo). Como resolver este problema?

- **Opção 1:** criar um segundo método chamado **gratificacaoGerente()**.
  - Problema 1: a classe Gerente possuirá dois métodos de gratificação, deixando-a confusa e permitindo a chamada do método errado.
  - Problema 2: caso a gratificação do diretor seja diferente, um terceiro método deve ser criado, e assim sucessivamente.

- **Melhor opção:** reescrever o método **gratificacao()** na classe **Gerente**.

Herança – sobrescrita de métodos

- **Exemplo:** se a gratificação concedida aos gerentes da empresa for diferente da gratificação concedida aos demais funcionários (75%, por exemplo). Isso pode ser feito reescrevendo o método **gratificacao()** na classe **Gerente**.

```
public class Funcionario {
    private String matricula;
    private double salario;

    public double gratificacao() {
        return this.salario * 0.5;
    }

    //...
}
```

```
public class Gerente extends Funcionario {
    private int subordinados;
    private int senha;

    public double gratificacao() {
        return this.getSalario() * 0.75;
    }

    //...
}
```

Todo o objeto da classe **Funcionario** executará o método **gratificacao()** da sua classe e todo o objeto da classe **Gerente** executará o método **gratificacao()** da sua classe.

Herança – sobrescrita de métodos

- **Exemplo:** se a gratificação concedida aos gerentes da empresa for diferente da gratificação concedida aos demais funcionários (75%, por exemplo). Isso pode ser feito reescrevendo o método **gratificacao()** na classe **Gerente**.

```
public class Funcionario {
    private String matricula;
    private double salario;

    public double gratificacao() {
        return this.salario * 0.5;
    }

    //...
}
```

```
public class Gerente extends Funcionario {
    private int subordinados;
    private int senha;

    public double gratificacao() {
        return this.getSalario() * 0.75;
    }

    //...
}
```

Perceba que a classe Gerente, apesar de herdar os atributos de Funcionario, não pode acessá-los diretamente, pois eles são privados (o acesso é feito pelo método acessor correspondente). Uma solução para isso seria utilizar outro modificador de acesso (protected – protegido).

Herança – sobrescrita de métodos

- **Exemplo:** se a gratificação concedida aos gerentes da empresa for diferente da gratificação concedida aos demais funcionários (75%, por exemplo). Isso pode ser feito reescrevendo o método **gratificacao()** na classe **Gerente**.

```
public class Funcionario {
    private String matricula;
    private double salario;

    public double gratificacao() {
        return this.salario * 0.5;
    }

    //...
}
```

```
public class Gerente extends Funcionario {
    private int subordinados;
    private int senha;

    public double gratificacao() {
        return this.getSalario() * 0.75;
    }

    //...
}
```

É possível adicionar a anotação **@Override**, que indica que o método foi sobrescrito da sua classe-pai.

```
@Override
public double gratificacao() {
    return this.getSalario() * 0.75;
}
```

Herança – sobrescrita de métodos

- **Exemplo:** se a gratificação concedida aos gerentes da empresa for diferente da gratificação concedida aos demais funcionários (75%, por exemplo). Isso pode ser feito reescrevendo o método **gratificacao()** na classe **Gerente**.

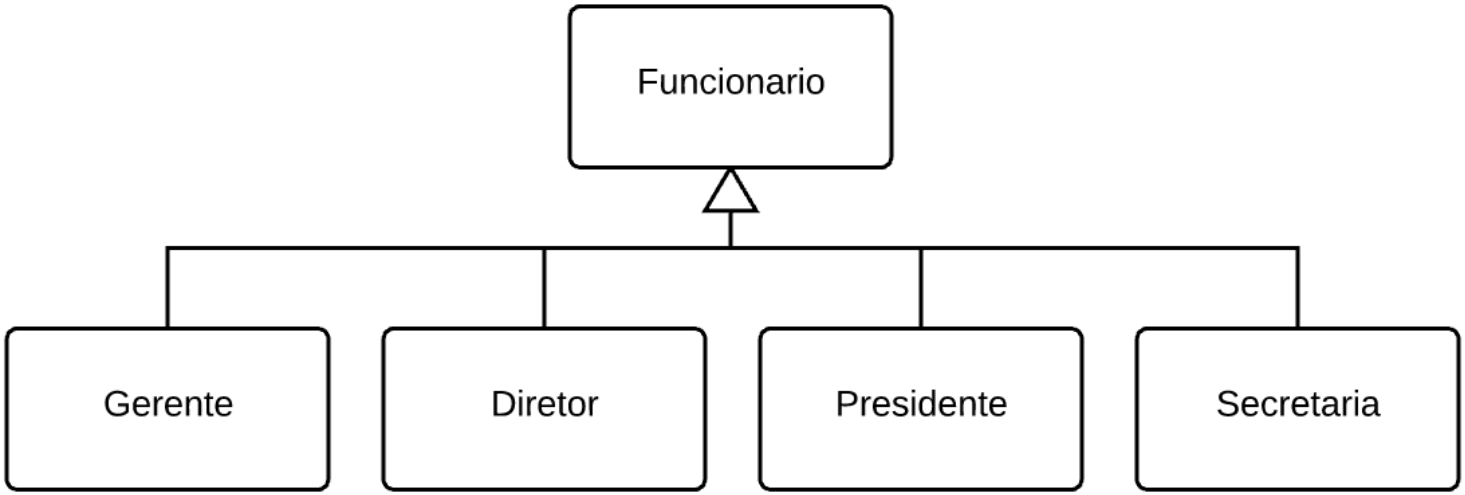
```
public class Funcionario {
    private String matricula;
    private double salario;

    public double gratificacao() {
        return this.salario * 0.5;
    }

    //...
}
```

```
public class Gerente extends Funcionario {
    private int subordinados;
    private int senha;

    public double gratificacao() {
        return this.getSalario() * 0.75;
    }

    //...
}
```

```
Gerente g = new Gerente();
g.setSalario(1000);

Funcionario f = new Funcionario();
f.setSalario(1000);

System.out.println(g.gratificacao()); //Imprimirá 750.0
System.out.println(f.gratificacao()); //Imprimirá 500.0
```

Herança – sobrescrita de métodos

- Com isso, a classe **Funcionario** define uma implementação geral para o método **gratificacao()** e os diferentes tipos de funcionários podem definir suas próprias implementações através da reescrita do método. Novos tipos de funcionários podem ser facilmente incluídos na estrutura (flexibilidade).



Herança – sobrescrita de métodos

- **Exemplo:** se a gratificação concedida aos gerentes da empresa for igual à dos demais funcionários, mas com um acréscimo de R$500,00.

- Uma solução consiste em copiar a implementação da gratificação dos funcionários e acrescentar os R$500,00.

```
@Override
public double gratificacao() {
    return this.getSalario() * 0.5 + 500;
}
```

- No entanto, se a gratificação dos funcionários for alterada (para 60%, por exemplo), o método da classe Gerente também deverá ser alterado. O mesmo ocorre se a solução for aplicada a outros tipos de funcionários.

Herança – sobrescrita de métodos

- **Exemplo:** se a gratificação concedida aos gerentes da empresa for igual à dos demais funcionários, mas com um acréscimo de R$500,00.

- Uma solução melhor consiste em chamar o método **gratificacao()** da classe-mãe (**Funcionario**) e acrescentar os R$500,00.

```
@Override
public double gratificacao() {
    return super.gratificacao() + 500;
}
```

- O acesso à classe-mãe é feito pelo comando **super**, que devolve a instância da superclasse da herança, permitindo a execução do método implementado nela. Neste caso, permitindo o acesso ao método **gratificacao()** da classe **Funcionario**.
- Esta técnica é comumente utilizada quando o método da classe-filha deve fazer "algo mais" em relação à implementação da classe-mãe.

Polimorfismo

- Na herança desenvolvida, um gerente **É UM** funcionário.
  - **Exemplo:** se um funcionário for chamado para representar a empresa em uma entrevista, um gerente pode fazê-lo, pois o gerente é um funcionário (semântica).
- Uma variável do tipo **Funcionario** armazena uma referência a um **Funcionario**. Logo, ela pode armazenar uma referência a um **Gerente**, pois este é um **Funcionario**.

```
Gerente g1 = new Gerente();
Funcionario f1 = g1;
Funcionario f2 = new Gerente()
```

Polimorfismo

- Na herança desenvolvida, um gerente **É UM** funcionário.
  - **Exemplo:** se um funcionário for chamado para representar a empresa em uma entrevista, um gerente pode fazê-lo, pois o gerente é um funcionário (semântica).
- Uma variável do tipo **Funcionario** armazena uma referência a um **Funcionario**. Logo, ela pode armazenar uma referência a um **Gerente**, pois este é um **Funcionario**.

```
Gerente g1 = new Gerente();
Funcionario f1 = g1;
Funcionario f2 = new Gerente()
```

- **Polimorfismo:** capacidade de um objeto poder ser referenciado de várias formas.
- Neste caso, o objeto **f1** pode armazenar uma referência a um objeto da classe **Funcionario** ou uma referência a um objeto da classe **Gerente**.
  - Se tivéssemos mais classes estendendo **Funcionario**, ele poderia armazenar uma referência a um objeto de qualquer uma dessas classes (**várias formas**).

Polimorfismo

- Se o método **gratificacao()** for chamado no exemplo abaixo, qual valor imprimirá: 500 (50%) ou 750 (75%)?

```
Funcionario f = new Gerente();
f.setSalario(1000);
System.out.println(f.gratificacao());
```

Polimorfismo

- Se o método **gratificacao()** for chamado no exemplo abaixo, qual valor imprimirá: 500 (50%) ou 750 (75%)?

```
Funcionario f = new Gerente();
f.setSalario(1000);
System.out.println(f.gratificacao());
```

- A decisão sobre qual método executar é feito em tempo de execução. O Java verifica qual a classe do objeto que está sendo referenciado dentro da variável e executa o respectivo método.
- Neste caso, executará o método implementado dentro da classe **Gerente** (que é a classe da referência armazenada em **f**), imprimindo o valor de **750**.

Polimorfismo

- O polimorfismo é útil quando queremos definir um método genérico para todos os funcionários, independente do seu tipo (gerente, diretor, secretária, etc.).
- A classe abaixo controla o total de gratificações concedidas. O método registro recebe um funcionário e computa a gratificação do mesmo.

```
public class ControleGratificacoes {
    private double totalGratificacoes = 0;

    public void registro(Funcionario f) {
        this.totalGratificacoes += f.gratificacao();
    }

    public double getTotalGratificacoes() {
        return this.totalGratificacoes;
    }
}
```

Polimorfismo

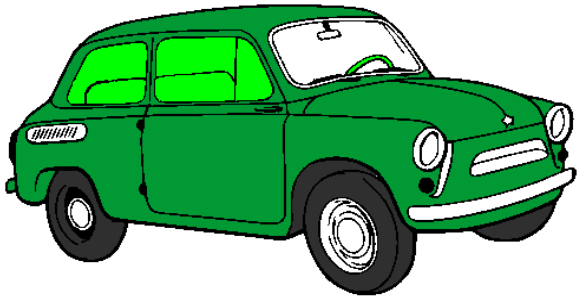
- O polimorfismo é útil quando queremos definir um método genérico para todos os funcionários, independente do seu tipo (gerente, diretor, secretária, etc.).
- A classe abaixo controla o total de gratificações concedidas. O método registro recebe um funcionário e computa a gratificação do mesmo.

```
public class ControleGratificacoes {
    private double totalGratificacoes = 0;

    public void registro(Funcionario f) {
        this.totalGratificacoes += f.gratificacao();
    }

    public double getTotalGratificacoes() {
        return this.totalGratificacoes;
    }
}
```

O método registro recebe uma referência a um **Funcionario**, chamando seu método **gratificacao()**. Ou seja, ele pode receber referências a qualquer classe que estende **Funcionario**, verificando a referência recebida e chamando o método correto.

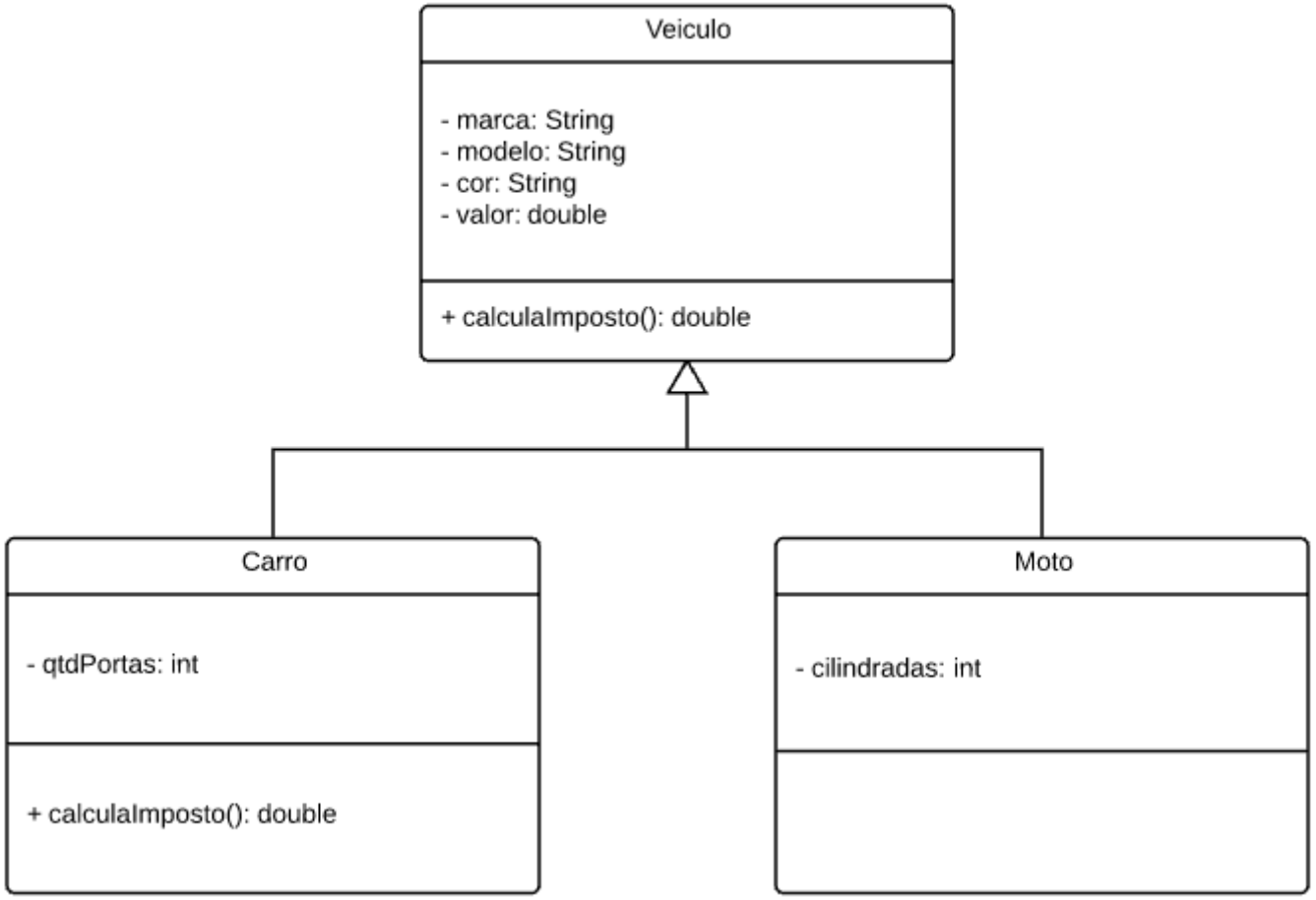
Exemplo – veículos

- Considere duas entidades: carro e moto. Um carro possui uma marca, um modelo, uma cor, um valor e um número de portas. Uma moto possui uma marca, um modelo, uma cor, um valor e uma quantidade de cilindradas. Como as classes possuem replicação de código, podemos definir uma classe geral e estendê-la nas classes Carro e Moto.
- Além dos atributos, estas duas entidades possuem em comum um método para cálculo do seu imposto, que corresponde a 2% do seu valor. Especificamente para carros, é acrescido R$ 800,00 ao seu imposto.



Exemplo – veículos

- Estrutura de classes



Exemplo – veículos

- Classe **Veiculo**

```
public class Veiculo {
    private String marca;
    private String modelo;
    private String cor;
    private double valor;

    public double calculaImposto() {
        return this.valor * 0.02;
    }

    public Veiculo() {}

    public Veiculo(String marca, String modelo, String cor, double valor) {
        this.marca = marca;
        this.modelo = modelo;
        this.cor = cor;
        this.valor = valor;
    }

    //Métodos acessores

}
```

Exemplo – veículos

- Classe **Veiculo**

```
public class Veiculo {
    private String marca;
    private String modelo;
    private String cor;
    private double valor;

    public double calculaImposto() {
        return this.valor * 0.02;
    }

    public Veiculo() {}

    public Veiculo(String marca, String modelo, String cor, double valor) {
        this.marca = marca;
        this.modelo = modelo;
        this.cor = cor;
        this.valor = valor;
    }

    //Métodos acessores

}
```

A classe **Veiculo** define os atributos de qualquer veículo e uma implementação padrão para o método **calculaImposto()**. Também são definidos construtores para a classe.

Exemplo – veículos

- Classe **Carro**

```
public class Carro extends Veiculo {
    private int numPortas;

    @Override
    public double calculaImposto() {
        return super.calculaImposto() + 800;
    }

    public Carro() {
        super();
    }

    public Carro(int numPortas, String marca, String modelo, String cor, double valor) {
        super(marca, modelo, cor, valor);
        this.numPortas = numPortas;
    }

    //Métodos acessores

}
```

Exemplo – veículos

- Classe **Carro**

```
public class Carro extends Veiculo {
    private int numPortas;

    @Override
    public double calculaImposto() {
        return super.calculaImposto() + 800;
    }

    public Carro() {
        super();
    }

    public Carro(int numPortas, String marca, String modelo, String cor, double valor) {
        super(marca, modelo, cor, valor);
        this.numPortas = numPortas;
    }

    //Métodos acessores

}
```

Na classe **Carro,** é adicionado o atributo específico de um carro (número de portas) e o método **calculaImposto()** é sobrescrito, adicionando os R$ 800,00 à implementação padrão.

Exemplo – veículos

- Classe **Moto**

```
public class Moto extends Veiculo {
    private int cilindradas;

    public Moto() {
        super();
    }

    public Moto(int cilindradas, String marca, String modelo, String cor, double valor) {
        super(marca, modelo, cor, valor);
        this.cilindradas = cilindradas;
    }

    //Métodos acessores

}
```

Exemplo – veículos

- Classe **Moto**

```
public class Moto extends Veiculo {
    private int cilindradas;

    public Moto() {
        super();
    }

    public Moto(int cilindradas, String marca, String modelo, String cor, double valor) {
        super(marca, modelo, cor, valor);
        this.cilindradas = cilindradas;
    }

    //Métodos acessores

}
```

Não é necessário sobrescrever o método **calculaImposto()** na classe **Moto**, pois não há nenhum comportamento específico da moto no cálculo do imposto.

Exemplo – veículos

- Classe **Aplicacao**

```
public class Aplicacao {

    private List<Veiculo> veiculos = new ArrayList<Veiculo>();

    //Métodos que utilizam a estrutura de herança e polimorfismo

}
```

Métodos desejados

- Criação de registros de carros e motos e armazenamento na lista polimórfica.
- Verificação de veículos de uma determinada marca.
- Apresentação dos veículos e seus valores de impostos.
- Apresentação de todos os carros da lista.

Exemplo – veículos

- Criação de registros de carros e motos e armazenamento na lista polimórfica

```
private void criaRegistros() {
    Carro c1 = new Carro(2, "VW", "Gol", "prata", 25000);
    Carro c2 = new Carro(4, "Fiat", "Uno", "branco", 20000);
    Carro c3 = new Carro(4, "Renault", "Clio", "preto", 32000);
    Carro c4 = new Carro(2, "Fiat", "147", "amarelo", 8000);

    Moto m1 = new Moto(150, "Honda", "CG", "azul", 7000);
    Moto m2 = new Moto(150, "Yamaha", "YBR", "vermelho", 12000);

    veiculos.add(c1);
    veiculos.add(c2);
    veiculos.add(c3);
    veiculos.add(c4);
    veiculos.add(m1);
    veiculos.add(m2);
}
```

Objetos das classes **Carro** e **Moto** são criados e armazenados em uma lista de objetos da classe **Veiculo**. Isso é polimorfismo.

Exemplo – veículos

- Verificação de veículos de uma determinada marca

```
private void veiculosDaMarca(String marca) {
    int qtd = 0;
    for(Veiculo v: veiculos) {
        if(v.getMarca().equals(marca))
            qtd++;
    }

    JOptionPane.showMessageDialog(null, "A marca " + marca + " possui " + qtd + " veículos!");
}
```

A lista é percorrida, independente da referência que se encontra a cada iteração (carro ou moto). Pela herança, é garantido que todos os objetos possuem o método **getMarca()**.

Exemplo – veículos

- Apresentação dos veículos e seus valores de impostos

```
private void mostraImpostos() {
    String texto = "";
    for(Veiculo v : veiculos) {
        texto += v.getMarca() + " " + v.getModelo() + "(" + v.getValor() + "): "
                                                    + v.calculaImposto() + "\n";
    }

    JOptionPane.showMessageDialog(null, texto);
}
```

Em tempo de execução, o Java verifica qual a referência armazenada em **v** e executa o respectivo método **calculaImposto()**. Isto é, o método implementado em **Veiculo** ou em **Carro** é executado.

Exemplo – veículos

- Apresentação de todos os carros da lista

```
private void mostraCarros() {
    String texto = "";
    for(Veiculo v: veiculos) {
        if(v instanceof Carro)
            texto += v.getMarca() + " " + v.getModelo() + ", cor " + v.getCor() + "\n";
    }
    JOptionPane.showMessageDialog(null, texto);
}
```

O comando **instanceOf** verifica se o objeto (esquerda) é do tipo da classe desejada (direita), retornando verdadeiro ou falso.

Referências

CAELUM. **Apostila Java e Orientação a Objetos**. Curso FJ-11, 2016.

DEITEL, H. M. **Java: como programar**. H. M Deitel e P. J. Deitel - 8a ed. Porto Alegre: Prentice-Hall, 2010.

Leitura complementar

TutorialsPoint Java (http://www.tutorialspoint.com/java).