



Java

## Programação Orientada por Objetos II

### METODOLOGIA

- › Interprete o documento calmamente e com atenção.
- › Acompanhe a execução do exercício no seu computador.
- › Não hesite em consultar o formador para o esclarecimento de qualquer questão.
- › Não prossiga para o ponto seguinte sem ter compreendido totalmente o ponto anterior.
- › Caso seja necessário, execute várias vezes o exercício até ter compreendido totalmente o processo.

### CONTEÚDO PROGRAMÁTICO

1. [Visibilidade](#)
2. [Encapsulamento](#)
3. [Static](#)
4. [Overloading](#)

## 1. Visibilidade

A visibilidade é definida através de três palavras reservadas do Java: `public`, `protected` e `private`. Esta definição aplica-se a todos os métodos e atributos e indica se é acessível ou não fora da própria classe ou classe derivada.

- `public` – indica que o método/atributo é acessível em qualquer classe.
- `protected` – indica que o método/atributo só é acessível numa classe derivada desta.
- `private` – indica que o método/atributo apenas é acessível dentro da classe onde foi declarado.

📁 O modo de acesso `protected` vai ser abordado de forma mais detalhada nos módulos seguintes.

Vamos aprofundar estes princípios continuando o exemplo do "Carro". Assumindo que a matrícula e a marca apenas são definidas na construção do objeto, e que para um determinado objeto "carro" não mudam com o tempo, então fará sentido garantir que não possam ser alteradas. Neste momento não existe qualquer indicação no código nesse sentido.

- › **Altere** o método `main` da classe `TesteCarro` **inserindo** o código salientado:

```
        carro1.marca = "Bmw";  
        carro1.mostrar();  
        carro2.mostrar();  
        carro3.mostrar();  
    }  
}
```

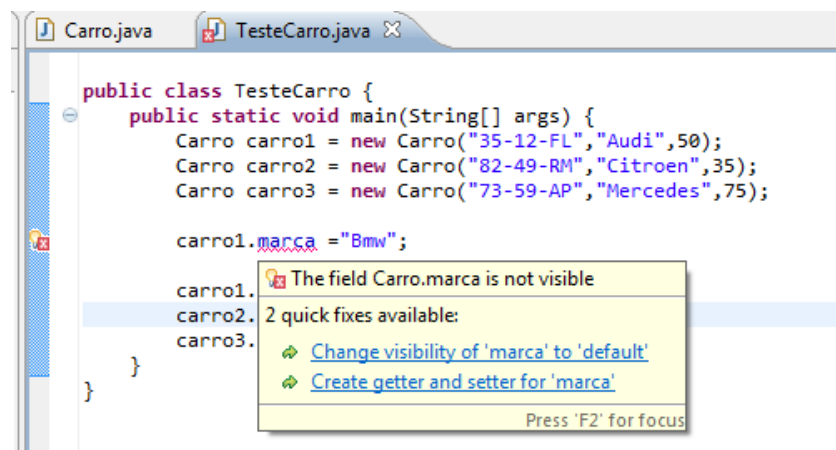
- › **Teste** esta alteração e **confirme** que na consola o "carro1" contém o valor "Bmw" na marca

- › **Altere** agora na classe `Carro` o código realçado:

```
public class Carro {  
    private String marca, matricula;  
    private int velocidade;  
}
```

Acabámos de definir que todos os atributos da classe `Carro` são privados, logo apenas podem ser modificados dentro da própria classe.

- › **Verifique** que após esta alteração é mostrado um **erro** na classe `TesteCarro`



Neste momento o Eclipse indica-nos que estamos a fazer um acesso ilegal ao atributo "marca" do objeto "carro1" porque este é agora privado. Mas então como se modificam os atributos fora da classe? Para isso podemos criar métodos específicos, que podem validar os valores que pretendemos atribuir aos atributos ou incluir outro tipo de lógica.

Passemos agora à criação do método que altera a velocidade:

- › **Adicione** à classe `Carro` o método:

```
public void atribuirVelocidade(int velocidade_recebida){  
    if ((velocidade_recebida >= 0) && (velocidade_recebida <= 50)){  
        //apenas atribuir se a velocidade estiver entre 0 e 50  
        velocidade = velocidade_recebida;  
    }  
}
```

- › **Substitua** na classe `TesteCarro` a linha `carro1.marca = "Bmw";` pela seguinte:

```
carro1.atribuirVelocidade(30);
```

› **Compile e execute** a classe `TesteCarro`

O código construído controla o acesso aos atributos dos objetos do tipo "carro", impedindo o programador de definir valores que não fazem sentido. Estas definições são de extrema importância num projeto de grande escala.

## 2. Encapsulamento

O encapsulamento é a divisão das entidades e ações do programa, com o intuito de o tornar mais flexível, fácil de modificar e criar novas funcionalidades. Tendo isto em consideração, no caso do carro, a ideia será abstrair o programador do seu funcionamento interno, de maneira a que tenha apenas acesso aos métodos necessários para quem utiliza um carro e não de acesso ao seu motor. Por questões de coerência fará sentido aplicar o mesmo conceito aos atributos de um carro, não deixando os programadores modificarem livremente, sem quaisquer restrições, as suas características.

A última alteração feita à classe `Carro` aplica já este conceito de encapsulamento, pois conseguimos alterar a velocidade sem aceder diretamente ao atributo. No entanto, falta o método para conhecer a velocidade corrente.

› **Adicione** à classe `Carro` o seguinte método:

```
public int obterVelocidade(){  
    return velocidade;  
}
```

Uma das grandes vantagens do encapsulamento é que a classe em si pode modificar os seus atributos internamente sem que os programadores, que apenas utilizam a classe, se apercebam. Seguindo este conceito na classe `Carro`, podíamos agora retirar o método `atribuirVelocidade` o que garantia que a velocidade apenas era aumentada ou diminuída de acordo com valores dos travões e do acelerador.

Uma boa prática é ter dois métodos para cada atributo, um para obter o valor e outro para o alterar. Obviamente haverá casos em que só um, ou mesmo nenhum destes métodos fará sentido. Um atributo poderá também ser totalmente interno, apenas de auxílio a cálculos internos feitos no objeto.

Vamos praticar um pouco adicionando a definição de cor a esta classe `Carro`:

› **Adicione** o seguinte atributo à classe `Carro`:

```
private String cor;
```

Agora vamos construir os métodos assessores para este atributo:

› **Adicione** mais dois métodos à classe `Carro`:

```
/**
 * Define uma nova cor para o carro
 * @param novacor o nome da cor para o carro
 */
public void atribuirCor(String novacor){
    cor = novacor;
}

/**
 * Obtem a cor corrente do carro
 * @return String que representa a cor
 */
public String obterCor(){
    return cor;
}
```

› Agora **adicione**, na classe `TesteCarro`, antes do fim do método `main`, o código para testar estes métodos:

```
carro1.atribuirCor("Verde");
System.out.println("Nova cor do carro1 - " + carro1.obterCor());
```

› **Compile e teste** a classe

Desta forma, se quiséssemos que a velocidade da classe `Carro` fosse guardada internamente em metros por segundo (m/s), podíamos fazê-lo sem que os programadores que a utilizam se apercebessem. Tendo em conta que para a conversão de **km/h** para **m/s** basta dividir pelo valor **3.6**, seriam necessárias as seguintes alterações:

› **Altere** o código destacado:

📄 Foram omitidos os métodos que não sofreram quaisquer alterações.

```
package carro;
public class Carro {
    private String marca, matricula;
    private double velocidade;
    private String cor;

    private static final double CONVERSAO_METROS_SEGUNDO = 3.6;

    /**
     * acelera o carro no valor de 10km/h
     */
    public void acelerar(){
        velocidade += 10/CONVERSAO_METROS_SEGUNDO;
    }

    public void mostrar() {
        System.out.println();
        System.out.println("Marca -> " + marca);
    }
}
```

```

        System.out.println("Matricula -> " + matricula);
        System.out.println("Velocidade -> " +
            (int)(velocidade*CONVERSAO_METROS_SEGUNDO));
        //conversao para km/h
    }

    public Carro(String matricula_carro, String marca_carro, int velocidade_carro){
        marca = marca_carro;
        matricula = matricula_carro;
        velocidade = velocidade_carro/CONVERSAO_METROS_SEGUNDO;
    }

    public void atribuirVelocidade(int velocidade_recebida){
        if (velocidade_recebida >= 0 && velocidade_recebida <= 50){
            //apenas atribuir se a velocidade estiver entre 0 e 50
            velocidade = velocidade_recebida/CONVERSAO_METROS_SEGUNDO;
        }
    }

    public int obterVelocidade(){
        return (int) (velocidade*CONVERSAO_METROS_SEGUNDO);
    }
}

```

› **Teste** este código através da classe `TesteCarro`

📄 Como deve reparar, o resultado na consola permanece inalterado.

Acabámos de implementar uma alteração interna na classe `Carro`, sem que quem utiliza esta classe sequer se aperceba. Após este código, a classe `Carro` não só mudou o tipo do atributo velocidade como este também mudou para metros por segundo em vez de quilómetros por hora. Esta é uma das grandes vantagens do encapsulamento.

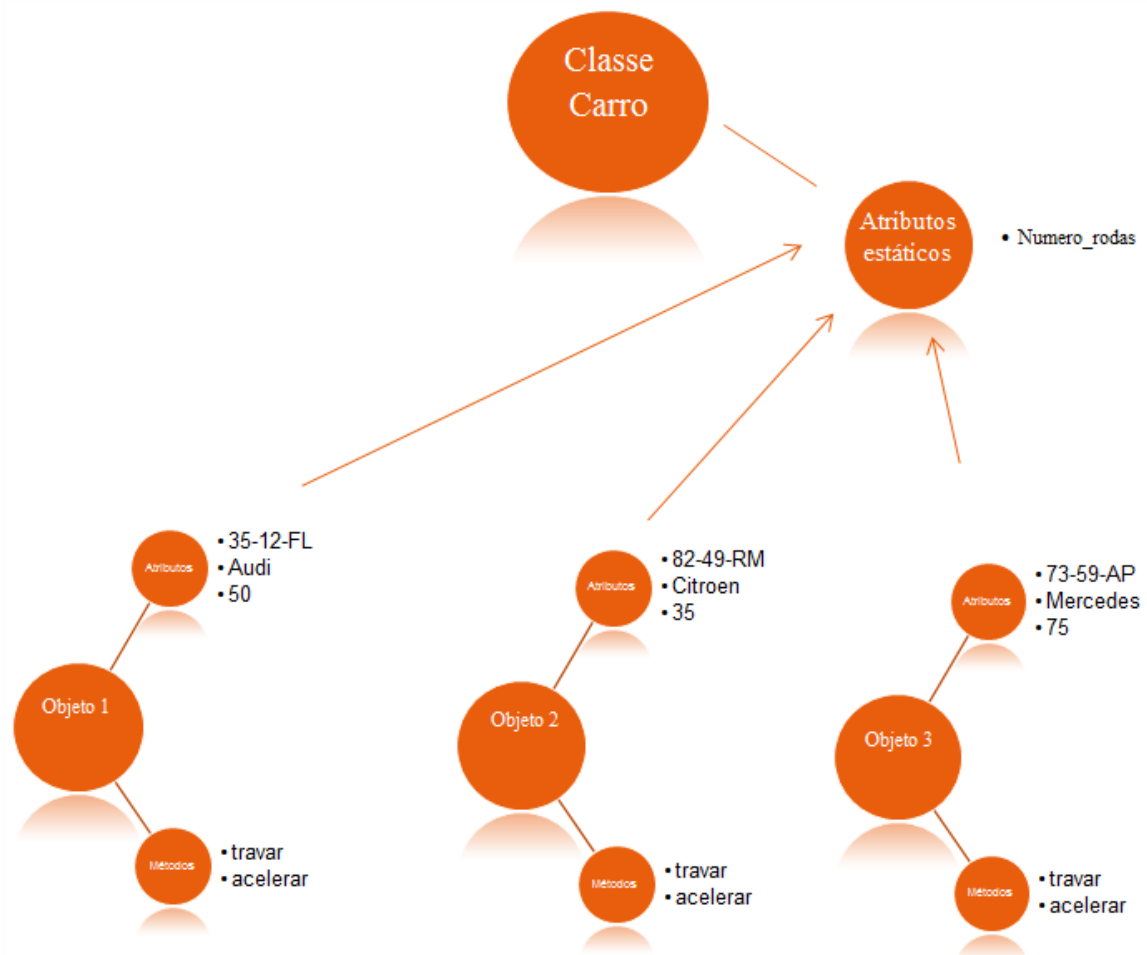
De notar que vemos aqui uma nova instrução:

```
return (int) (velocidade * CONVERSAO_METROS_SEGUNDO);
```

O `(int)` força uma conversão para inteiro do valor que se encontra à sua frente. Neste caso estamos a converter o resultado da multiplicação para inteiro e, de seguida, retornar esse valor. Esta alteração pode ser realizada entre a maior parte dos tipos básicos.

### 3. Static

Esta palavra tem aparecido ao longo dos vários módulos, mas o que faz ao certo? O **static** define se o método ou atributo no qual foi introduzido se refere à instância ou ao tipo (classe). Se o atributo se referir à instância, então para cada nova instância é incluído esse atributo com o valor que a instância (objeto) o definir. Se este for **static** então ele fica definido na Classe (tipo) e é como se fosse partilhado entre todas as instâncias do mesmo tipo. Se imaginarmos por exemplo um atributo "número de rodas", fará sentido ser definido ao nível da classe e não ao nível dos objetos criados, uma vez que todos os objetos do tipo "Carro" devem ter o mesmo número de rodas. Isto é exemplificado na figura seguinte.



Passemos à implementação deste conceito.

› Abra a classe `Carro`

› Adicione o seguinte atributo:

```
public static int numeroRodas = 4;
```

› Agora **adicione** no fim do método `main` da classe `TesteCarro` as seguintes instruções:

```
System.out.println("Num rodas de Carro - " + Carro.numeroRodas);  
Carro.numeroRodas = 10; //teste de alteracao do valor atributo  
System.out.println("Num rodas Carro - " + Carro.numeroRodas);
```

› Compile e teste a classe

📌 Repare na forma como é obtido e modificado o valor do atributo. É feito com o nome da classe e não através de um objeto em específico. Por este motivo depende-se que o atributo `numeroRodas` pertence diretamente à classe `Carro` e não aos objetos criados através dela.

 Não devemos esquecer as práticas de boa programação cumprindo as regras de encapsulamento.

› Troque na classe `Carro` a **visibilidade** do atributo `numero_rodas` para `private`

› Agora **adicione** o seguinte método assessor:

```
/**
 * Obtem o numero de rodas de um Carro
 * @return numero de rodas
 */
public static int obterNumeroRodas(){
    return numero_rodas;
}
```


› No método `main` da classe `TesteCarro` **substitua**:

```
System.out.println("Num rodas de Carro - " + Carro.numero_rodas);
Carro.numero_rodas = 10;
System.out.println("Num rodas Carro - " + Carro.numero_rodas);
```

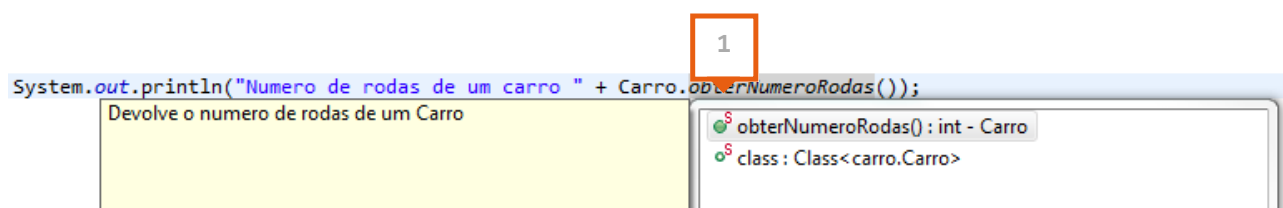
Por:

```
System.out.println("Numero de rodas de um carro " + Carro.obterNumeroRodas());
```

› **Teste** a classe

 Repare como apenas definimos um método para obter o valor, pois assumimos que não faz sentido alterar o número de rodas de um carro.

À medida que introduzimos o método, o Eclipse identifica a informação de que o método é **Estático** <sup>1</sup> através do "s" vermelho.



## 4. Overloading

Overloading traduz-se em português para "sobrecarga de métodos" e consiste em definir métodos com o mesmo nome de outros já existentes. Mas se têm o mesmo nome, como conseguimos saber qual é executado quando o referimos no código? A resposta reside nos seus parâmetros. Para se fazer overload a um método é necessário que novo método com o mesmo nome defina parâmetros diferentes, em número ou em tipo.

Seguindo este conceito podemos implementar overload para o método acelerar que recebe um valor para a aceleração.

› **Introduza** no método `main` da classe `TesteCarro` o código:

```
carro1.acelerar();
System.out.println("Nova velocidade do carro1 " + carro1.obterVelocidade());
carro1.acelerar();
System.out.println("Nova velocidade do carro1 " + carro1.obterVelocidade());
```

› **Teste** esta classe

📄 Como definido anteriormente, o método `acelerar` aumenta a velocidade do carro em 10.

› **Adicione** um novo método à classe `Carro`:

```
/**
 * acelera o carro com o valor recebido como parametro
 * desde que este seja inferior a 50km/h.
 * E tambem feita a conversao para m/s para representacao interna
 * @param factor o valor que ira acelerar a velocidade
 */
public void acelerar(int factor){
    if(factor < 50){
        //definir um valor maximo de aceleracao
        velocidade += factor/CONVERSAO_METROS_SEGUNDO;
    }
}
```

› Agora **execute** a classe `TesteCarro` e **confirme** que não houve alterações

Para podermos invocar este método é necessário passar o parâmetro de aceleração:

› **Adicione** ao fim do método `main` da classe `TesteCarro` o seguinte código:

```
carro1.acelerar(30);
System.out.println("Nova velocidade do carro1 " + carro1.obterVelocidade());
```

› **Execute** a classe

📄 Observe que agora o carro sofreu uma aceleração de 30km/h.



Este conceito é especialmente útil em construtores porque, desta forma, conseguimos ter várias formas disponíveis para construir um objeto.

Vamos adicionar mais um construtor (overload) para a classe `Carro`:

- › **Adicione** o seguinte método à classe `Carro`:

```
/**
 * Criar um novo objeto carro sem especificar velocidade, inicializando-a com o valor 0
 * @param matricula_carro matricula do carro
 * @param marca_carro marca do carro
 */
public Carro(String matricula_carro , String marca_carro){
    marca = marca_carro;
    matricula = matricula_carro;
    velocidade = 0;
}
```

Após esta adição já nos é possível construir um objeto do tipo carro definindo a velocidade, ou deixando a mesma por preencher.

- › **Acrescente** o seguinte código antes do fim do método `main` da classe `TesteCarro`:

```
Carro carro4 = new Carro("61-44-ZA", "Renault");
carro4.mostrar();
```

- › **Compile e execute** a classe

- ☐ Confirme que a velocidade deste último carro se encontra com o valor **0**.

⚠ É obrigatório manter o tipo de dados de retorno no overload de um método.