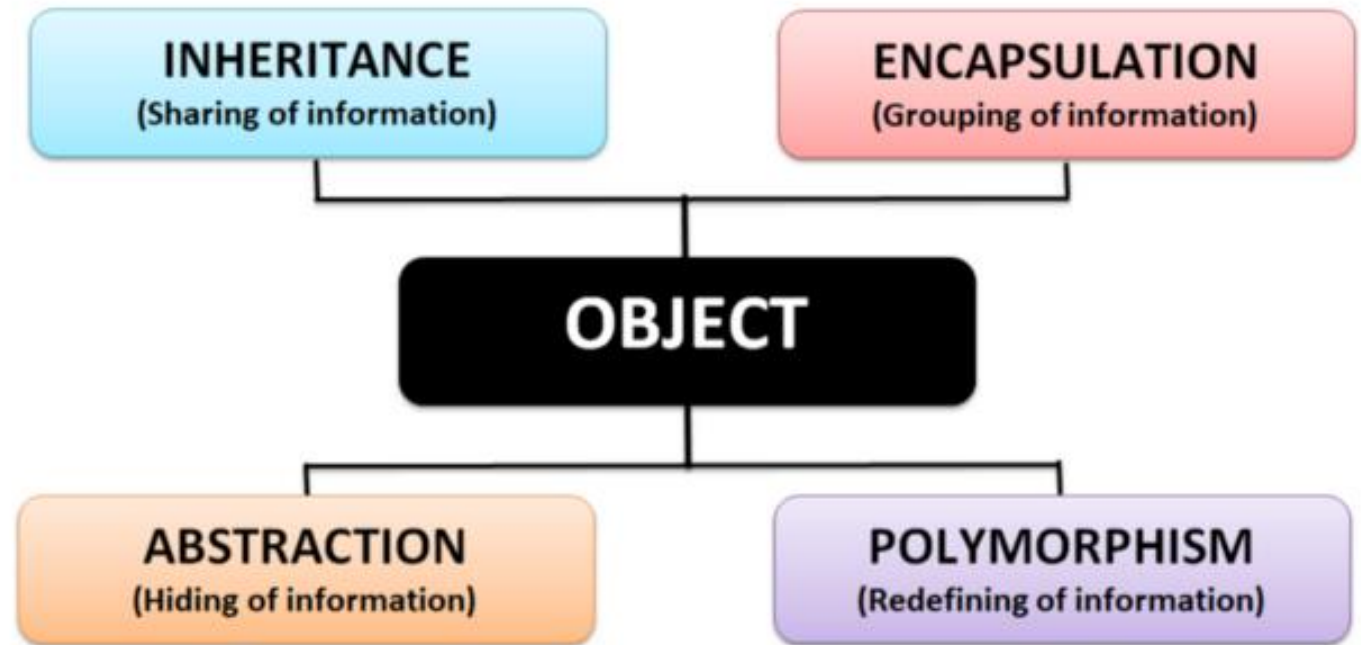




Ciência da **Computação**

Programação Orientada a Objetos
Prof. Luciano Rodrigo Ferretto

Pilares da Orientação a objetos

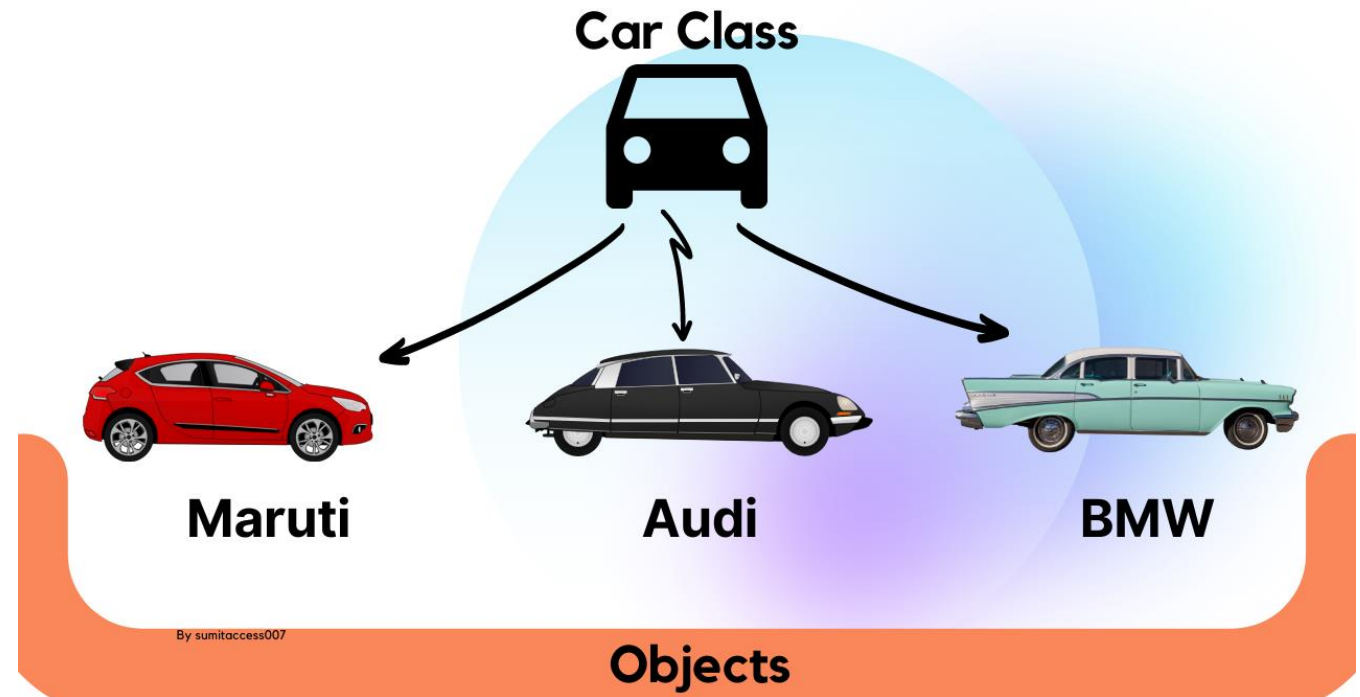
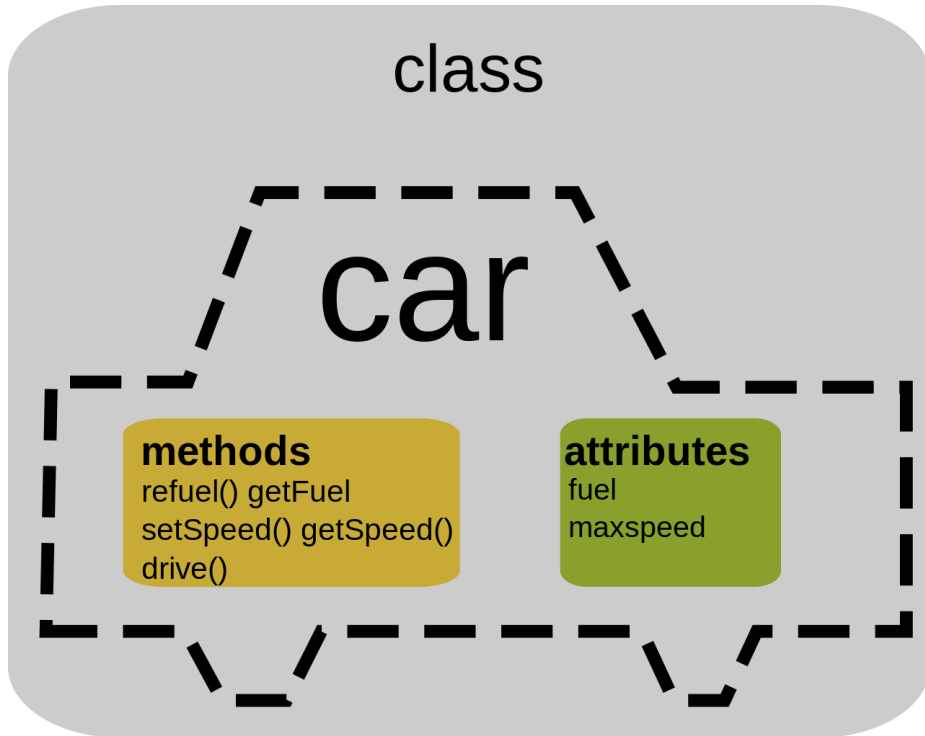


Abstração

- **Abstração** em Programação Orientada a Objetos é o processo de identificar e definir características essenciais de objetos do mundo real, “traduzindo” (representando) essas características em forma de **classes** e **objetos**.
- Uma **classe** é uma estrutura que define atributos e métodos que representam um tipo de objeto, fornecendo um modelo para criar instâncias desse tipo.
- Um **objeto** é uma instância de uma classe, caracterizado por seus atributos e comportamentos definidos pela classe, e pode interagir com outros objetos por meio de métodos e troca de dados.

Objeto é uma **instância** concreta de uma classe na POO.

Abstração



Abstração - Classes



- Classe serve como um modelo ou plano para criar objetos.
- Ela define as características (**atributos**) e comportamentos (**métodos**) que os objetos desse tipo terão.
- Em termos simples, uma classe pode ser considerada como um "molde" a partir do qual os objetos são criados.
- Ela descreve quais informações um objeto pode armazenar e quais operações ele pode realizar.



Exemplo de Classe

- Pens em um algoritmo para receber os dados do veículo, então o primeiro passo seria abstrair o “veículo” do mundo real para o nosso algoritmo OO.
- Para isso temos que elencar quais as características (atributos) do veículo são necessárias dentro do nosso escopo, e também quais são as ações que serão executadas pelos objetos desse tipo (métodos).

Atributos - Características

- Marca (Chevrolet, Volkswagen, Fiat, Kia, Hyundai, etc...)
- Modelo (Fusca, Cruze, Uno, Cerato, Journey, etc...)
- Ano
- Placa



Exemplo de Classe

```
1  // Abaixo temos um exemplo de uma classe
2  // com o nome "Veiculo" e que possui
3  // 4 (quatro) atributos
4  // Neste exemplo, esta classe não possui métodos
5  class Veiculo {
6      // "class" identifica o início da classe
7      // Atributos e métodos deve ficar entre as "chaves" {}
8      String marca;
9      String modelo;
10     int ano;
11     String placa;
12 }
```


Objeto



- Objeto é uma **instância** concreta de uma classe na POO.
- Ele representa uma entidade específica com características (atributos) e ações (métodos) associadas, conforme definidas na classe da qual foi criado.
- **Em outras palavras, um objeto é a representação real de um conceito abstrato (classe).**

Objeto



- Pense nos objetos como os "indivíduos" ou "casos" que seguem o modelo estabelecido pela classe.
- Cada objeto tem seu próprio conjunto de valores de atributos, que definem seu estado único, e pode executar os métodos associados para realizar ações específicas.
- Pense no exemplo da planta arquitetônica de uma casa. A partir dela podemos criar “ n ” casas, elas terão semelhanças, porém serão casas diferentes.

Objeto



- Por exemplo, usando a classe “Veiculo” definida anteriormente, podemos criar um objeto chamado “uno”.
- Neste exemplo “uno” é um objeto da classe “Veiculo” com os valores atribuídos aos seus atributos.

```
1  public class Main {  
2      public static void main(String[] args) {  
3          Veiculo uno = new Veiculo();  
4          uno.marca = "Fiat";  
5          uno.modelo = "Uno Mille";  
6          uno.ano = 1998;  
7          uno.placa = "ABC-1234";  
8      }  
9  }
```

Objeto



- Na linha 3 estamos declarando uma variável do tipo Veiculo e também estamos atribuindo à ela uma nova instância da classe, ou seja, um **objeto Veiculo**.
- Para **Instanciar** um objeto do tipo “Veiculo” utilizamos a palavra chave **new**.
- Após instanciado, podemos ler/alterar os valores dos atributos e invocar os seus métodos.

```
1  public class Main {  
2      public static void main(String[] args) {  
3          Veiculo uno = new Veiculo();  
4          uno.marca = "Fiat";  
5          uno.modelo = "Uno Mille";  
6          uno.ano = 1998;  
7          uno.placa = "ABC-1234";  
8      }  
9  }
```



Objeto

- Uma mesma classe pode ter infinitos objetos instanciados a partir dela;
- Assim como variáveis primitivas, também podemos declarar e atribuir em duas linhas:

```
Veiculo fusca;  
fusca = new Veiculo();
```

```
Veiculo uno = new Veiculo();  
uno.marca = "Fiat";  
uno.modelo = "Uno Mille";  
uno.ano = 1998;  
uno.placa = "ABC-1234";
```

```
Veiculo fusca = new Veiculo();  
fusca.marca = "Volkswagem";  
fusca.modelo = "Fusca - Série Ouro";  
fusca.ano = 1995;  
fusca.placa = "DEF-5678";
```

Atribuição por Valor X Referência

Atribuição por Valor



- Nas variáveis primitivas nós podemos duplicar os valores entre duas variáveis utilizando o operador de atribuição (=).
- Isso se chama **Atribuição por valor**

```
int numero1 = 5;  
int numero2 = numero1;  
numero2++;  
System.out.println("Número 01 = " + numero1);  
System.out.println("Número 02 = " + numero2);
```

Atribuição por Valor



```
int numero1 = 5;
int numero2 = numero1;
numero2++;
System.out.println("Número 01 = " + numero1);
System.out.println("Número 02 = " + numero2);
```

| Variável | Valor |
|----------|-------|
| numero1 | 5 |
| numero2 | 5 → 6 |

Atribuição por Referência



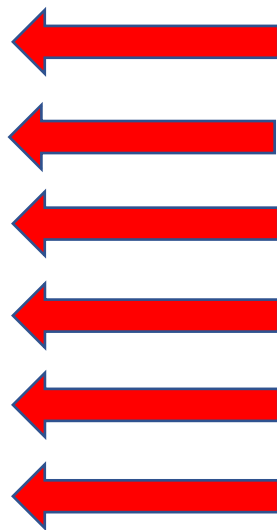
- Objetos em Java (e outras linguagens) são tratados por referências.
- Ou seja, a variável não armazena o objeto, mas sim a referência da memória onde este objeto está alocado

```
Veiculo uno = new Veiculo();
uno.marca = "Fiat";
uno.modelo = "Uno Mille";
uno.ano = 2001;
Veiculo novoUno = uno;
uno.ano = 2014;
System.out.println("O ano do uno é: " + uno.ano);
System.out.println("O ano do novo uno é: " + novoUno.ano);
```

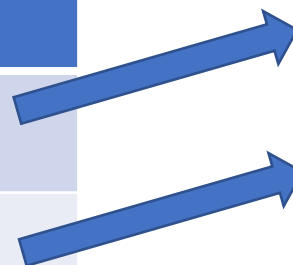


Atribuição por Valor

```
Veiculo uno = new Veiculo();  
uno.marca = "Fiat";  
uno.modelo = "Uno Mille";  
uno.ano = 2001;  
Veiculo novoUno = uno;  
uno.ano = 2014;
```



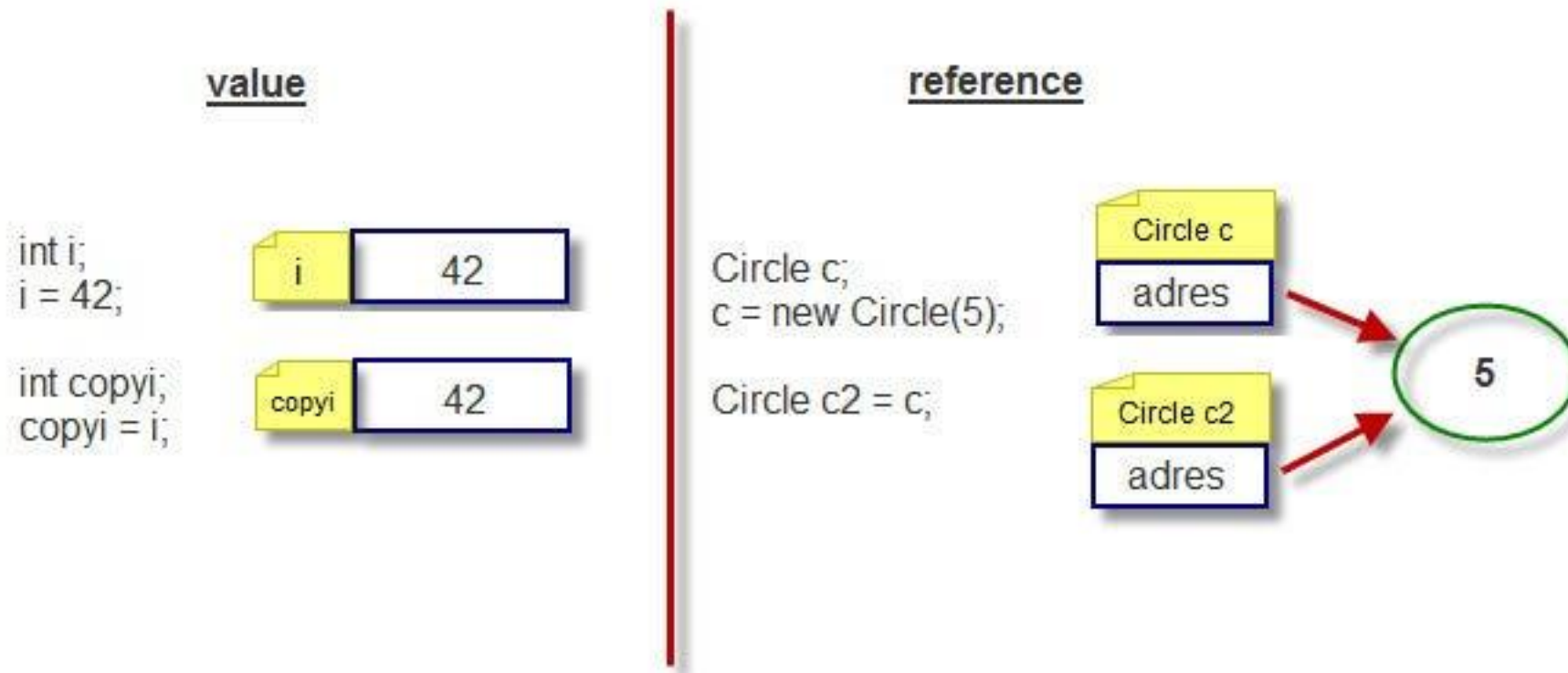
| Variável | Valor |
|----------|------------|
| uno | Veiculo@xx |
| novoUno | Veiculo@xx |



Objeto **Veiculo@xx**

- marca: "Fiat"
- modelo: "Uno Mille"
- ano: ~~2001~~ 2014
- Placa:

Atribuição por Valor x por Referência





Métodos

- Um método em programação orientada a objetos (POO) é um bloco de código que executa uma tarefa específica e pode ser chamado ou **invocado** por um objeto.
- Os métodos permitem que os objetos realizem ações, processem dados e interajam com outros objetos, seguindo o comportamento definido pela classe.
- Em termos mais simples, um método é uma função associada a uma classe que descreve as ações ou operações que os objetos dessa classe podem realizar.
- Eles são essenciais para o encapsulamento de comportamento, permitindo que você modele como os objetos se comportam e interagem no sistema.

Métodos

- A palavra-chave **this** identifica **objeto instanciado que invocou a ação**, ou seja, não terá nenhum efeito sobre outros objetos instanciados de classe igual.
- A partir do objeto (*fusca*) você pode invocar o método.
- Este é um exemplo de um método (função) que não recebe nenhum parâmetro como entrada e não retorna nenhum valor.

```
class Veiculo {  
    String marca;  
    String modelo;  
    int ano;  
    String placa;  
    void calculaTempoUso() {  
        int tempoUso = 2024 - this.ano;  
        System.out.println("O tempo de uso deste carro é: "  
            + tempoUso + " ano(s)");  
    }  
}
```

```
Veiculo fusca = new Veiculo();  
fusca.marca = "Volkswagem";  
fusca.modelo = "Fusca - Série Ouro";  
fusca.ano = 1995;  
fusca.placa = "DEF-5678";  
  
fusca.calculaTempoUso();
```



Exemplo de Métodos Retornando valores

- Os métodos podem retornar dados, para isso você precisa definir o tipo de dados que deseja retornar e utilizar a palavra chave **return**;
- O retorno pode ser armazenado em uma variável ou então ser redirecionado para outros métodos em outras classes/objetos
- Como já visto no método *main* e no método de exemplo anterior, quando não há retorno devemos utilizar a palavra chave **void**.

```
class Veiculo {  
    String marca;  
    String modelo;  
    int ano;  
    String placa;  
    int calculaTempoUso() {  
        int tempoUso = 2024 - this.ano;  
        return tempoUso;  
    }  
}
```

```
int tempoUsoUno = uno.calculaTempoUso();  
System.out.println("O tempo de uso do Uno é: "  
    + tempoUsoUno);  
System.out.println("O tempo de uso do Fusca é: "  
    + fusca.calculaTempoUso());
```



Exemplo de Métodos Recebendo Parâmetros

- Assim como um método pode retornar um valor, os métodos também podem **receber valores**, e nesse caso é no plural mesmo, pois podemos receber vários parâmetros.
- O parâmetro pode ter como origem uma variável qualquer ou pode ser um valor primitivo.

```
class Veiculo {
    String marca;
    String modelo;
    int ano;
    String placa;
    int calculaTempoUso(int anoBase) {
        int tempoUso = anoBase - this.ano;
        return tempoUso;
    }
}

int anoBase = LocalDate.now().getYear();
int tempoUsoUno = uno.calculaTempoUso(anoBase);
System.out.println("O tempo de uso do Uno é: "
    + tempoUsoUno);
System.out.println("O tempo de uso do Fusca é: "
    + fusca.calculaTempoUso(anoBase));
```



Exemplo de Métodos com Objetos

- Um método pode, tanto, receber um **objeto** como parâmetro, quanto, retornar um **objeto** na sua expressão return.

```
class Veiculo {  
    String marca;  
    String modelo;  
    int ano;  
    String placa;  
  
    Veiculo cloneMe() {  
        Veiculo veiculoDestino = new Veiculo();  
        veiculoDestino.marca = this.marca;  
        veiculoDestino.modelo = this.modelo;  
        veiculoDestino.ano = this.ano;  
        veiculoDestino.placa = this.placa;  
        return veiculoDestino;  
    }  
}
```

```
Veiculo outroFusca = fusca.cloneMe();
```




Métodos Trabalhando com Objetos

- Quando o parâmetro passado ao método é um **objeto**, esta é uma *chamada por referência*, diferente de parâmetros de tipos primitivos, que utilizam *chamada por valor*.
- Na chamada por valor, uma cópia do valor real do argumento é passada para a função ou método. Isso significa que qualquer modificação feita ao parâmetro dentro da função não afetará o valor original fora da função. Isso é comum em linguagens como C, C++, Java (para tipos primitivos) e outras.
- Na chamada por referência, um ponteiro ou referência ao valor real do argumento (e não o valor do argumento) é passada para o parâmetro. Isso significa que qualquer modificação feita ao parâmetro dentro da função afetará o valor original fora da função. Isso é comum em linguagens como C++ (com uso de ponteiros), C#, Java (para tipos de Classe – Tipos de Referência) e outras.
- No Java a chamada por valor é realizada para os tipos primitivos, e a chamada por referência para os tipos de Classe.



Exemplo de Métodos com Objetos

- Neste exemplo temos um método que recebe um objeto Veiculo como parâmetro, **cria um novo Objeto (instancia)** e copia os valores de seus atributos.
- No segundo método o objeto recebido por parâmetro, ou seja, sua referência é apenas passada para uma nova variável do tipo Veiculo, a qual é retornada, ou seja, a referência do parâmetro de entrada aponta para a mesma referência do retorno.

```
Veiculo cloneFromOther(Veiculo veiculoOrigem) {  
    Veiculo veiculoDestino = new Veiculo();  
    veiculoDestino.marca = veiculoOrigem.marca;  
    veiculoDestino.modelo = veiculoOrigem.modelo;  
    veiculoDestino.ano = veiculoOrigem.ano;  
    veiculoDestino.placa = veiculoOrigem.placa;  
    return veiculoDestino;  
}  
Veiculo cloneFromOtherWrong(Veiculo veiculoOrigem) {  
    Veiculo veiculoDestino = veiculoOrigem;  
    return veiculoDestino;  
}
```



Métodos Construtores

- O **método construtor**, também conhecido simplesmente como "**construtor**", é utilizado para inicializar e configurar objetos quando eles são criados a partir de uma classe.
- O construtor é uma função especial que é executada automaticamente quando um novo objeto é instanciado, permitindo que você realize qualquer inicialização necessária para o objeto.
- O construtor é responsável por inicializar os atributos e propriedades de um objeto quando ele é criado. Isso garante que o objeto comece em um estado consistente e utilizável.



Métodos Construtores

- Métodos construtores tem o mesmo nome da classe e não tem tipo de retorno.
- **Construtor Padrão:** Se uma classe não fornecer nenhum construtor, o compilador Java criará um construtor padrão sem parâmetros para você. Esse construtor padrão inicializa os membros do objeto com valores padrão.
- Normalmente, usamos um construtor para fornecer valores iniciais para as variáveis de instância definidas pela classe ou para executar algum outro procedimento de inicialização necessário à criação de um objeto totalmente formado.

```
class Veiculo {  
    String marca;  
    String modelo;  
    int ano;  
    String placa;  
    // Método Construtor  
    Veiculo(){  
  
    }  
  
    Veiculo(String marca, String modelo, int ano, String placa) {  
        this.marca = marca;  
        this.modelo = modelo;  
        this.ano = ano;  
        this.placa = placa;  
    }  
}
```



Sobrecarga de Métodos

- Em Java, dois ou mais métodos da mesma classe podem compartilhar o mesmo nome, contanto que suas declarações de parâmetros sejam diferentes.
- Quando é esse o caso, diz-se que os métodos estão sobrecarregados e o processo é chamado de sobrecarga de método. A sobrecarga de métodos é uma das maneiras pelas quais Java implementa o **polimorfismo**.
- Em geral, para sobrecarregar um método, só temos que declarar versões diferentes dele. O compilador se incumba do resto. Porém, é preciso prestar atenção em uma restrição importante: **o tipo e/ou a quantidade dos parâmetros de cada método sobrecarregado devem diferir. Não é o bastante dois métodos diferirem apenas em seus tipos de retorno.**
- Mas os métodos sobrecarregados também podem diferir em seus tipos de retorno. Quando um método sobrecarregado é chamado, sua versão, cujos parâmetros coincidem com os argumentos da chamada, é executada.

Sobrecarga de Métodos

- Neste exemplo temos dois métodos com o nome **calculaTempoUso**, porém com assinaturas diferentes.
- Um deles não recebe nenhum parâmetro e o outro recebe um valor do tipo primitivo int.
- O que irá definir qual método será executado são os parâmetros que serão passados, ou não, na chamada do método.
- Normalmente métodos sobrescritos fazem chamadas entre si, porém isso não é obrigatório, depende muito do contexto.

```
int calculaTempoUso() {  
    int anoAtual = LocalDate.now().getYear();  
    int tempoUso = calculaTempoUso(anoAtual);  
    return tempoUso;  
}  
  
int calculaTempoUso(int anoBase) {  
    int tempoUso = anoBase - this.ano;  
    return tempoUso;  
}
```

```
System.out.println(  
    "O tempo de uso do Fusca é: "  
    + fusca.calculaTempoUso());  
System.out.println(  
    "O tempo de uso do Fusca em 2030 será: "  
    + fusca.calculaTempoUso(2030));
```

O tempo de uso do Fusca é: 29

O tempo de uso do Fusca em 2030 será: 35



Sobrecarga de Métodos - Construtor

- O método construtor também pode ser Sobrecarregado.
- Nesse caso, quando desejamos que um método invoque o outro, então utilizamos a palavra-chave **this**.
- Assim como nos outros métodos, não há a necessidade de um método construtor sobrecarregado chamar outro.
- Neste exemplo, quando instanciamos um objeto, podemos optar em passar os parâmetros ou não.

```
// Método Construtor Padrão
Veiculo(){
    //Aqui invocamos o método construtor com os parâmetros
    this("Marca desconhecida", "Modelo desconhecido",
        LocalDate.now().getYear(), "");
}

// Método construtor recebendo parâmetros
Veiculo(String marca, String modelo, int ano, String placa) {
    this.marca = marca;
    this.modelo = modelo;
    this.ano = ano;
    this.placa = placa;
}
```

```
Veiculo uno = new Veiculo();
Veiculo fusca = new Veiculo("Volkswagen", "Fusca - Série Ouro", 1995, "DEF=5678");
```

Ciência da



Encapsulamento

- **Encapsulamento** é um dos quatro pilares da programação orientada a objetos (POO).
- Ele se refere à prática de esconder os detalhes internos de uma classe e expor apenas a interface necessária para interagir com essa classe.
- Isso é feito definindo os membros da classe como públicos, privados ou protegidos, e fornecendo métodos para acessar e manipular esses membros.
- Os principais objetivos do encapsulamento são:
 - **Controle de Acesso:** Controlar quem pode acessar e modificar os atributos da classe. Isso ajuda a evitar alterações indesejadas e a garantir que os dados sejam usados corretamente.
 - **Proteção dos Dados:** Garantir que os dados internos da classe não sejam corrompidos ou acessados indevidamente.
 - **Flexibilidade:** Permitir que a implementação interna da classe seja alterada sem afetar o código que a utiliza. Isso é possível porque a interface pública (métodos públicos) permanece a mesma.



Encapsulamento – Controle de Acesso

- Em Java, o encapsulamento é alcançado através da definição de **modificadores de acesso** nos membros da classe e da disponibilização de métodos **getters** (para acessar os atributos) e **setters** (para modificar os atributos), quando apropriado.
- Aqui estão os principais modificadores de acesso em Java:
 - **public**: O membro é acessível de qualquer lugar.
 - **private**: O membro é acessível somente dentro da própria classe.
 - **protected**: O membro é acessível dentro da própria classe, dentro do mesmo pacote, e em suas subclasses em outros pacotes.
 - **default** (sem modificador): O membro é acessível apenas dentro do mesmo pacote.

Encapsulamento – Getters e Setters

- A utilização mais comum de Encapsulamento nos algoritmos OO são os **getters** e **setters**.
- São métodos responsáveis por realizar as a consulta e edição dos atributos da classe, que por boas práticas de programação devem ter seu modificador de acesso do tipo **private**.
- Os nomes dos métodos não obrigatoriamente devem seguir o padrão **get** e **set**, porém, por convenção se utiliza o padrão ao lado.
- **A partir de agora, sempre iremos utilizar esta boa prática.**

```
1 public class VeiculoComEncapsulamento {
2     private String marca;
3     private String modelo;
4     private int ano;
5     private String placa;
6     public String getMarca() {
7         return marca;
8     }
9     public void setMarca(String marca) {
10        this.marca = marca;
11    }
12    public String getModelo() {
13        return modelo;
14    }
15    public void setModelo(String modelo) {
16        this.modelo = modelo;
17    }
18    public int getAno() {
19        return ano;
20    }
21    public void setAno(int ano) {
22        this.ano = ano;
23    }
24    public String getPlaca() {
25        return placa;
26    }
27    public void setPlaca(String placa) {
28        this.placa = placa;
29    }
30 }
```

Atividade: Sistema de Gestão de Biblioteca

Desenvolva um sistema de gestão de biblioteca em Java que inclua as seguintes classes:

- **Livro:** Representa um livro na biblioteca. Deve incluir atributos como título, autor, ano de publicação e número de páginas.
- **Biblioteca:** Representa a biblioteca em si. Deve incluir métodos para adicionar um livro ao acervo, remover um livro, buscar um livro pelo título, listar todos os livros disponíveis.
- **Main:** Classe principal onde você irá instanciar objetos das classes Livro e Biblioteca para testar o funcionamento do sistema.

Model View Control

- Nessa nossa atividade vamos assumir que:
 - Classe Main será nossa View (através do Console) e Control, pois irá enviar/receber os dados para/do o usuário
 - Classe Biblioteca será nossa Model, a qual será responsável por armazenar os livros (Pode ser em uma List) e também fazer as regras de negócio necessárias.
 - Classe Livro será nossa Entidade principal.