

# SOBRECARGA DE MÉTODOS

```
1 package cap06;
2 public class AreaComSobrecarga {
3     public static void main(String args[]) {
4         System.out.println("Área de um quadrado..." + calcularArea(3));
5         System.out.println("Área de um retângulo..." + calcularArea(3, 2));
6         System.out.println("Área de um cubo....." + calcularArea(3, 2, 5));
7     }
8     public static double calcularArea(int x) {
9         return (x * x);
10    }
11    public static double calcularArea(int x, int y) {
12        return (x * y);
13    }
14    public static double calcularArea(int x, int y, int z) {
15        return (x * y * z);
16    }
17 }
```

Todos os três **métodos** da imagem têm **assinaturas** diferentes. Isso significa que, mesmo os métodos tendo os mesmos nomes, o dado enviado pela classe de teste vai saber pra quem enviar.

Não confundir com construtor, os comandos enviados não precisam ser por objetos de referência criados pelos construtores. Eu posso fazer simplesmente isso, pra enviar pro primeiro método, no caso.

```
AreaComSobrecarga.calcularArea(40)

double z = calcularVol.produto(4, 2, 'digite o valor de z(altura)');

system.out.println(calcularArea.somar(20,30));
```

# Declaração de objetos/instância

The image shows two handwritten code snippets and a class diagram. The left snippet is the `main` method of `UsaTelevisor`, and the right snippet is the `Televisor` class definition. Yellow arrows connect the `tv` variable in the `main` method to the `Televisor` class and its attributes. A red box highlights the object creation line, with a legend below it. The class diagram shows the structure of the `Televisor` class.

```
1 package cap07;
2 public class UsaTelevisor {
3     public static void main(String[] Args) {
4         Televisor tv = new Televisor();
5         tv.canal = 150;
6         tv.volume = 3;
7         tv.aumentarVolume();
8         tv.aumentarVolume();
9         tv.trocarCanal(10);
10        tv.mostrar();
11    }
12 }

1 package cap07;
2 public class Televisor {
3     public int volume;
4     public int canal;
5     public void aumentarVolume() {
6         volume++;
7     }
8     public void diminuirVolume() {
9         volume--;
10    }
11    public void trocarCanal(int c) {
12        canal = c;
13    }
14    public void mostrar() {
15        System.out.println("Volume: " + volume + "\nCanal: " + canal);
16    }
17 }
```

**Object Declaration Legend:**

objeto/instância	ATRIBUTO
<code>Televisor tv = new Televisor();</code>	<code>Nome CLASSE</code> <code>Nome OBJETO</code> <code>Nome CONSTRUTOR</code>
<code>tv.canal = 150;</code>	

**Class Diagram: Televisor**

- Attributes:
  - +volume : int
  - +canal : int
- Operations:
  - +aumentarVolume() : void
  - +reduzirVolume() : void
  - +trocarCanal(int canal) : void
  - +mostrar() : String

# STATIC

**Variável/instância** definida dentro de uma classe como STATIC: valor dessa variável é ESTÁTICO, não muda. Por exemplo:

```
public class Celular{
    //sera possível atribuir um valor de número de telefone para
    //cada atributo de objetos de celulares. Ex:

    //Celular celular1 = new Celular();
    //celular1.numero = "11 98814 3436";
    public String numero;

    //Agora, com o static, só será atribuído o valor
    //(nome da empresa) à classe, não ao objeto da classe.
    //ex: Celular.nomeEmpresa = "Xiaomi"

    public static String nomeEmpresa;
}
```

No método segue o mesmo princípio:

```
//teste
Celular.discarNumero("11 98814 3436")
//classe principal:
public static String discarNumero(int x)
```

## Encapsulamento (Data Hidding)

Comandos para acessar o objeto, como `tv.canal` não funcionam mais se a atributo/método for privado. É necessário criar um método público só pra acessar esse atributo/método que ficou privado.

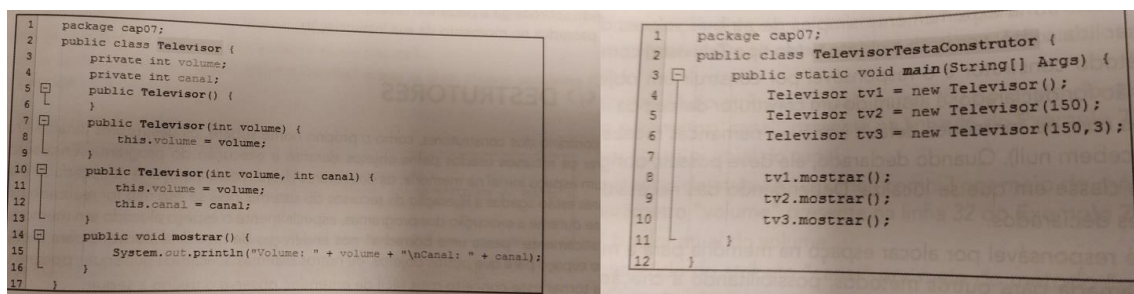
Para cada atributo privado é necessário 1 get e 1 set.

O `This` serve para fazer referência ao valor presente dentro da classe em questão. Então ele vai pegar o valor privado que foi enviado, e não um valor global que foi determinado em uma outra classe.

```
//set padrão:
public void setNomeDoAtributo (tipo_do_atributo nomeDoAtributo){
    this.nomeDoAtributo = nomeDoAtributo;
}

//get padrão:
public tipo_do_atributo getNomeDoAtributo(){
    return nomeDoAtributo;
}
```

Exemplo de código com encapsulamento e construtores:



```
1 package cap07;
2 public class Televisor {
3     private int volume;
4     private int canal;
5     public Televisor() {
6     }
7     public Televisor(int volume) {
8         this.volume = volume;
9     }
10    public Televisor(int volume, int canal) {
11        this.volume = volume;
12        this.canal = canal;
13    }
14    public void mostrar() {
15        System.out.println("Volume: " + volume + "\nCanal: " + canal);
16    }
17 }
```

```
1 package cap07;
2 public class TelevisorTesteConstrutor {
3     public static void main(String[] Args) {
4         Televisor tv1 = new Televisor();
5         Televisor tv2 = new Televisor(150);
6         Televisor tv3 = new Televisor(150,3);
7
8         tv1.mostrar();
9         tv2.mostrar();
10        tv3.mostrar();
11    }
12 }
```

# HERANÇA

Super(): Os Atributos dos objetos (variáveis) são herdadas, uma vez que JÁ estão presentes na super Classe. No exemplo abaixo as variáveis COR e TAMANHO são da classe mãe – foram criadas no construtor de lá.

A variável TIPO não existia anteriormente, portanto, é criada na subclasse. Como a classe tem as três variáveis, TIPO, COR e TAMANHO, o construtor dela vai “importar” essas duas variáveis da classe mãe com o SUPER.

```
1 package cap07;
2 public class Bola {
3     private String cor;
4     private int tamanho;
5
6     public Bola(String cor, int tamanho) {
7         this.cor = cor;
8         this.tamanho = tamanho;
9     }
10
11     public void mostrar() {
12         System.out.println(cor);
13         System.out.println(tamanho);
14     }
15 }
```

- Classe BolaFutebol

```
1 package cap07;
2 public class BolaFutebol extends Bola {
3     private String tipo;
4     public BolaFutebol(String cor, int tamanho, String tipo) {
5         super(cor, tamanho);
6         this.tipo = tipo;
7     }
8
9     public void mostrar() {
10         super.mostrar();
11         System.out.println(tipo);
12     }
13 }
```

Dois tipos de polimorfismo:

**(1) Overriding/Sobreposição** : redefinição de métodos ENTRE classe mãe/filha. No exemplo abaixo, a classe MÃE tem um método para comer, assim como as classes FILHAS também têm. Todos os métodos têm o mesmo parâmetro, nome e assinatura! Porém, ao serem chamados, trarão resultados **específicos**. Isso acontece porque o método da superclasse é substituído pelo método da subclasse.

```
public class Animal {
    public void comer() {
        System.out.println( "Animal Comendo..." );
    }
}

public class Cao extends Animal {
    public void comer() {
        System.out.println( "Cão Comendo..." );
    }
}

public class Tigre extends Animal {
    public void comer() {
        System.out.println( "Tigre Comendo..." );
    }
}
```

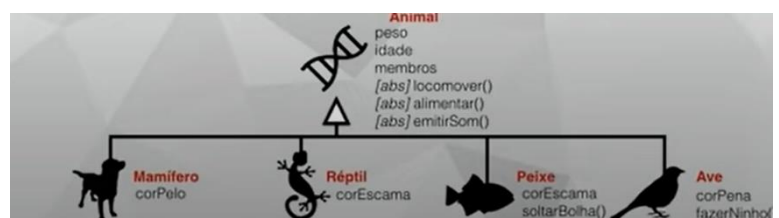
```
public class Test {

    public void fazerAnimalComer( Animal animal ) {
        animal.comer();
    }

    public static void main( String[] args ) {
        Test t = new Test();
        t.fazerAnimalComer( new Animal() );
        t.fazerAnimalComer( new Cao() );
        t.fazerAnimalComer( new Tigre() );
    }
}
```

```
classe abstrata Animal
    protegido peso: Real
    protegido idade: Inteiro
    protegido membros: Inteiro
    publico metodo abstrato locomover()
    publico metodo abstrato alimentar()
    publico metodo abstrato emitirSom()
FimClasse

classe Ave estende Animal
    privado corPena: Caractere
    @Sobrep
    publico metodo locomover()
        Escreva("Voando")
    fimMetodo
    @Sobrep
    publico metodo alimentar()
        Escreva("Comendo frutas")
    fimMetodo
    @Sobrep
    publico metodo emitirSom()
        Escreva("Som de ave")
    fimMetodo
    publico metodo fazerNinho()
        Escreva("Construiu um ninho")
    fimMetodo
FimClasse
```



(2) **Overloading/ Sobrecarga** : sobrecarga de métodos de uma MESMA classes: métodos com assinaturas DIFERENTES (mudar o tipo de parâmetro [float, int, etc] ou mudar a quantidade de argumentos [int x, int y //int x]).

**reagir()**

falar frase	agradável: abanar e latir agressiva: rosar
horário do dia	manhã: abanar tarde: abanar e latir noite: ignorar
dono	é dono: abanar não é: rosar e latir
idade e peso	novo e leve: abanar novo e pesado: latir velho e leve: rosar velho e pesado: ignorar

```

classe Cachorro estende Lobo
publico metodo reagir(frase: Caractere)
fimMetodo
publico metodo reagir(hora, min: Inteiro)
fimMetodo
publico metodo reagir(dono: Logico)
fimMetodo
publico metodo reagir(idade: Inteiro,
peso: Real)
fimMetodo
FimClasse
  
```

Assinaturas diferentes

Por exemplo, dentro da classe CACHORRO, que é classe filha de animal, podem existir diferentes tipos de reações (reagir() ) – como exemplificado na imagem. A partir disso, os parâmetros diferentes conseguem diferenciar os tipos de reações possíveis. Então, se o parâmetro DONO for FALSO, a reação será de rosar e latir

```

class HelperService {

    private String formatNumber(int value) {
        return String.format("%d", value);
    }

    private String formatNumber(double value) {
        return String.format("%.3f", value);
    }

    private String formatNumber(String value) {
        return String.format("%.2f", Double.parseDouble(value));
    }

    public static void main(String[] args) {
        HelperService hs = new HelperService();
        System.out.println(hs.formatNumber(500));
        System.out.println(hs.formatNumber(89.9934));
        System.out.println(hs.formatNumber("550"));
    }
}
  
```

Nesse exemplo, para cada tipo de valor de entrada, o método o tratará de uma forma específica (retornar como inteiro, 3 casas ou convertê-lo de String para double com 2 casas.