



Prof. Dr. Fernando Almeida  
[proffernando.almeida@fiap.com.br](mailto:proffernando.almeida@fiap.com.br)

# DDD (Domain Driven Design) Encapsulamento, Herança e Polimorfismo

# O QUE VAMOS APRENDER HOJE?

1

Revisão sobre Classes e objetos

4

Interfaces

2

Encapsulamento

5

Polimorfismo

3

Herança



# Um pouco sobre Polimorfismo

## Definição

“O paradigma da Orientação a Objetos traz um ganho significativo na **qualidade da produção de software**, porém, os benefícios são alcançados quando as técnicas de **Programação Orientada a Objetos** (POO) são colocadas em prática com o uso de uma tecnologia que nos permita usar todas as características da **Orientação a Objetos**



# Objeto (recapitulando...)

- Entidade do mundo real que tem uma **identidade**



podem representar

- entidades concretas**
- um arquivo
- uma bicicleta
- entidades conceituais**
- uma estratégia de jogo
- uma política de escalonamento de um Sistema Operacional

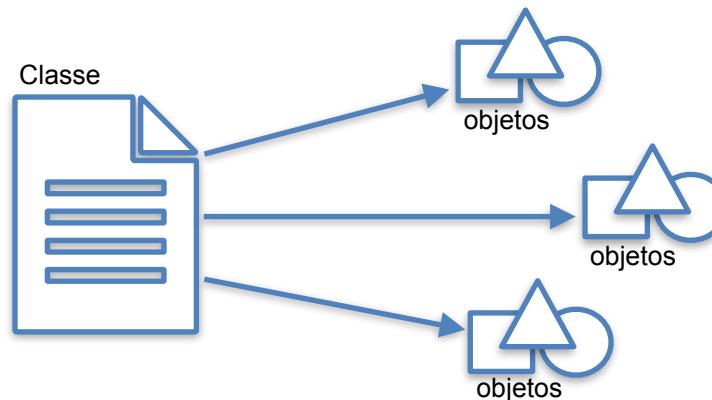
- Objetos são **instâncias de classe**, que determinam qual informação um objeto contém e como ele pode manipulá-la

“Cada objeto, ter sua própria identidade significa que, dois objetos são distintos mesmo que eles apresentem exatamente as mesmas características

“ Um programa desenvolvido com uma linguagem POO manipula **estruturas de dados** através dos objetos, da mesma forma que um programa em linguagem tradicional utiliza variáveis.

# Classe (recapitulando...)

- Estrutura que **abstrai** um conjunto de objetos com características similares
- Define o **comportamento** de seus objetos através de **métodos** e os **estados** possíveis através de **atributos**
- Descreve os **serviços providos** por seus objetos e quais informações eles podem armazenar
- São necessárias para que uma linguagem seja **orientada a objetos**



# Programação Orientada a Objetos

- A Programação Orientada a Objetos tem três pilares:
  - **Encapsulamento**
  - **Herança**
  - **Polimorfismo**

“ Antes de tratarmos destes assuntos se faz necessário o entendimento de alguns conceitos iniciais para que tudo possa ficar claro à medida que a aula for dando andamento

# Herança definição

**“ Herança é um mecanismo da Orientação a Objeto que permite criar novas classes a partir de classes já existentes, aproveitando-se das características existentes na classe a ser estendida.**

# • • • • • + • • • Herança • + . + •

- Promove o reuso e reaproveitamento de código existente
- É possível criar classes derivadas, super classes e subclasses,
  - Subclasses são mais especializadas do que as suas super classes (mais genéricas)
  - Subclasses herdam todas as características de suas super classes (atributos e métodos)
- Java permite apenas o uso de **Herança simples**, mas não permite a implementação de herança múltipla
  - **Para superar essa limitação o Java faz uso do conceito de interfaces**
- Sinônimos:
  - classe base, super tipo, **superclasse**, classe pai e classe mãe
  - classe derivada, subtipo, **subclasse** e classe filha

O que um **aluno, professor e um funcionário** possuem em comum?

## • • • • • + • • • Herança • + . + •

- Aluno, Professor e Funcionário
  - Todos eles são Pessoas e compartilham alguns dados em comum
  - Todos tem nome, idade, endereço, etc...

|  
+ O que diferencia um aluno de outra pessoa qualquer?

- Aluno possui matrícula, curso, semestre...
- Funcionário possui código do funcionário, data de admissão, salário...
- Professor possui código do professor, informações relacionadas à sua formação...

...é aqui que a herança se torna uma ferramenta de grande utilidade!

# • • • • • + • • Herança • + . + • •

- Podemos criar uma classe **Pessoa**, que possui todos os atributos e métodos comuns a todas as pessoas e herdar estes atributos e métodos em classes mais específicas (do geral para o mais específico)

```
| 1 | public class Pessoa{  
| 2 |     public String nome;  
| 3 |     public int idade;  
| 4 | }
```

- Classe Aluno que hera todos os atributos da classe Pessoa, (podendo incluir novos atributos - número de matrícula)

```
1 | public class Aluno extends Pessoa{  
2 |     public String matricula;  
3 | }
```

# Herança

- Em Java, a palavra-chave usada para indicar herança é **extends**
- A classe **Aluno** agora possui três atributos: *nome*, *idade* e *matrícula*

## Exemplo

```
1 | public class Estudos{  
2 |     public static void main(String args[]){  
3 |         Aluno aluno = new Aluno();  
4 |         aluno.nome = "Aluno Esforçado";  
5 |         aluno.idade = 20;  
6 |         aluno.matricula = "XXXX99999";  
7 |  
8 |         System.out.println("Nome: " + aluno.nome + "\n" +  
9 |             "Idade: " + aluno.idade + "\n" +  
10|              "Matrícula: " + aluno.matricula);  
11|     }  
12| }
```

“Herança nos fornece grande benefício, pois ao concentrarmos as características comuns em uma classe e derivar as classes mais específicas a partir desta, estamos preparados para a adição de novas funcionalidades ao sistema.

A adição de uma nova funcionalidade na superclasse irá automaticamente atualizar as classes derivadas.

• • • • • +

• • • • • .

## + Exemplo 1

```
1 public abstract class Animal {  
2     public abstract void fazerBarulho();  
3 }  
4  
5 public class Cachorro extends Animal {  
6     public void fazerBarulho() {  
7         System.out.println("AuAu!");  
8     }  
9 }  
10  
11 public class Gato extends Animal {  
12     public void fazerBarulho() {  
13         System.out.println("Miau!");  
14     }  
15 }
```

• + • •

## Exemplo 2

```
17 class Veiculo {  
18     public Veiculo() {  
19         System.out.print("Veiculo ");  
20     }  
21  
22     public void checkList() {  
23         System.out.println("Veiculo.checkList");  
24     }  
25  
26     public void adjust() {  
27         System.out.println("Veiculo.adjust");  
28     }  
29  
30     public void cleanup() {  
31         System.out.println("Veiculo.cleanup");  
32     }  
33 }
```

continua...

• • • • • +

• • • • • .

• + continuação...

```
+   34 class Automovel extends Veiculo {  
+   35     public Automovel() {  
+   36         System.out.println("Automovel");  
+   37     }  
+   38  
+   39     public void checkList() {  
+   40         System.out.println("Automovel.checkList");  
+   41     }  
+   42  
+   43     public void adjust() {  
+   44         System.out.println("Automovel.adjust");  
+   45     }  
+   46  
+   47     public void cleanup() {  
+   48         System.out.println("Automovel.cleanup");  
+   49     }  
+   50 }
```

• continua...

continuação...

```
52 class Bicicleta extends Veiculo {  
53     public Bicicleta() {  
54         System.out.println("Bicicleta");  
55     }  
56  
57     public void checkList() {  
58         System.out.println("Bicicleta.checkList");  
59     }  
60  
61     public void adjust() {  
62         System.out.println("Bicicleta.adjust");  
63     }  
64  
65     public void cleanup() {  
66         System.out.println("Bicicleta.cleanup");  
67     }  
68 }
```

- • • • •
- • • Herança - Construtores da Classe superclasse
- + •

- + •
  - Podemos utilizar construtores da super classe com a palavra reservada “super”
  - Dentro do construtor chamamos o construtor da super classe (classe pai)

## Exemplo

```
1 public class Carro{  
2  
3     private String cor;  
4     private double preco;  
5     private String modelo;  
6  
7     /* CONSTRUTOR PADRÃO */  
8     public Carro(){  
9  
10    }  
11  
12    /* CONSTRUTOR COM 2 PARÂMETROS */  
13    public Carro(String modelo, double preco){  
14        //Se for escolhido o construtor sem a COR do veículo  
15        // definimos a cor padrão como sendo PRETA  
16        this.cor = "PRETA";  
17        this.modelo = modelo;  
18        this.preco = preco;  
19    }  
20  
21    /* CONSTRUTOR COM 3 PARÂMETROS */  
22    public Carro(String cor, String modelo, double preco){  
23        this.cor = cor;  
24        this.modelo = modelo;  
25        this.preco = preco;  
26    }  
27  
28 }
```

continua...

continuação...

```
30 public class Honda extends Carro{  
31  
32     private String motor;  
33  
34     /* CONSTRUTOR PADRÃO */  
35     public Honda(){  
36  
37 }  
38  
39     /* CONSTRUTOR COM PARÂMETROS */  
40     public Honda(String motor, String modelo, double preco){  
41         super(modelo, preco);  
42         this.motor = motor;  
43     }  
44  
45 }  
46  
47 public class Aplicacao {  
48  
49  
50     public static void main(String[] args) {  
51         //Construtor sem parâmetros  
52         Honda hondaFitPreto = new Honda("2.0 Flex", "Honda Accord", "60000");  
53     }  
54 }  
55 }
```

Dentro do construtor da Classe Honda fazemos a chamada ao construtor de dois parâmetros da classe pai (Carro)

# Interface definição

# Interface

- Bloco de código que definindo um **tipo, métodos e atributos (parâmetros)** que “esse” tipo deve possuir
- Na prática: Qualquer classe que quiser ser do tipo definido pela interface deve implementar os métodos dessa interface
- A **interface** não contém nenhum código de implementação, apenas **assinaturas de métodos e atributos** dos métodos
- Define um padrão para especificação do **comportamento** de classes
  - os métodos de uma interface são implementados de maneira particular a cada classe
  - permite expressar comportamento sem se preocupar com a implementação
- Não possui atributos
- **Uma classe pode implementar várias interfaces, mas pode ter apenas UMA superclasse**

```
• • •  
• • +  
• + •  
• +  
1 public class TV {  
2     private int tamanho;  
3     private int canal;  
4     private int volume;  
5     private boolean ligada;  
6     public TV(int tamanho) {  
7         this.tamanho = tamanho;  
8         this.canal = 0;  
9         this.volume = 0;  
10        this.ligada = false;  
11    }  
12    // abaixo vem todos os métodos construtores get e set...  
13    // Encapsulamento  
14 }  
15  
16 public interface ControleRemoto {  
17     void mudarCanal(int canal);  
18     void aumentarVolume (int taxa);  
19     void diminuirVolume (int taxa);  
20     boolean ligar();  
21     boolean desligar();  
22 }
```

Construtor

Interface

# Exemplo de implementação

- Vamos desenvolver duas TVs diferentes, imaginando que fossem duas marcas complementarmente distintas e que uma não tem nenhuma relação com a outra.
- Como ambas as TVs irão implementar a interface Controle Remoto, então, no corpo das duas classes devem conter os métodos da interface

A TV modelo 001 é uma TV simples, sem muitos recursos...

```
1 public class ModeloTV001 extends TV implements ControleRemoto {  
2     public final String MODELO = "TV001"; → definição de constante em Java  
3     public ModeloTV001(int tamanho) {  
4         super(tamanho);  
5     }  
6  
7     public void desligar() {  
8         super.setLigada(false);  
9     }  
10  
11    public void ligar() {  
12        super.setLigada(true);  
13    }  
14  
15    public void aumentarVolume(int taxa) { }  
16    public void diminuirVolume(int taxa) { }  
17    public void mudarCanal(int canal) { }  
18 }
```

A TV modelo X é uma TV mais moderna, apresentará uma mensagem quando o método desligar for invocado

```
1 public class ModeloX extends TV implements ControleRemoto {  
2     public final String MODELO = "TV-X"; → definição de constante em Java  
3  
4     public ModeloSDX(int tamanho) {  
5         super(tamanho);  
6     }  
7  
8     public void desligar() {  
9         System.out.println("Obrigado por Utilizar a Televisão!");  
10        super.setLigada(false);  
11    }  
12  
13    public void ligar() {  
14        super.setLigada(true);  
15    }  
16  
17    public void aumentarVolume(int taxa) { }  
18    public void diminuirVolume(int taxa) { }  
19    public void mudarCanal(int canal) { }  
20 }
```

Ambos os métodos “desligar” possuem a mesma ação que é desligar, porém, cada um executa de forma diferente. Vamos testar na **Classe de Teste**:

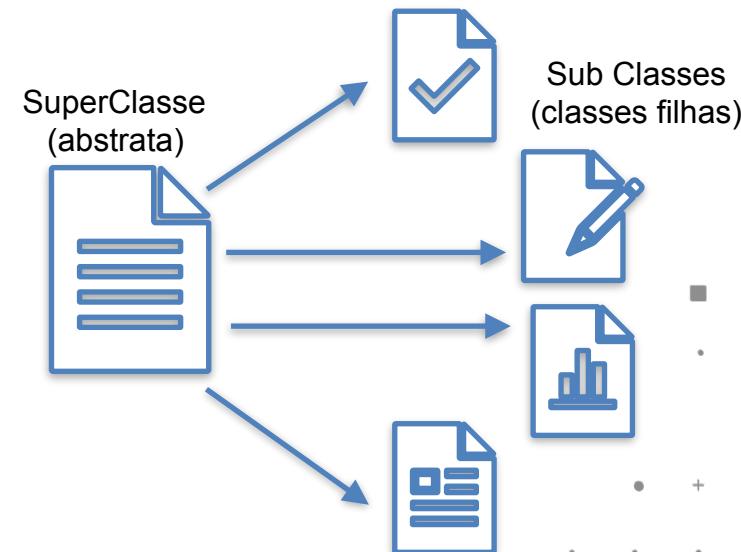
```
1 public class ExemploInterfaceamento {  
2     public static void main(String[] args) {  
3         ModeloTV001 tv1 = new ModeloTV001(21);  
4         ModeloSDX tv2 = new ModeloX (42);  
5         tv1.ligar();  
6         tv2.ligar();  
7         System.out.print("TV1 - modelo " + tv1.MODELO + " está ");  
8         System.out.println(tv1.isLigada() ? "ligada" : "desligada");  
9         System.out.print("TV2 - modelo " + tv2.MODELO + " está ");  
10        System.out.println(tv2.isLigada() ? "ligada" : "desligada");  
11        // ambas as TVs estão ligadas e vamos desligá-las  
12        System.out.println("Desligando modelo " + tv1.MODELO);  
13        tv1.desligar();  
14        System.out.println("Desligando modelo " + tv2.MODELO);  
15        tv2.desligar();  
16    }  
17 }
```

# Classes Abstratas

## definição

# Classe Abstrata

- Especificação **conceitual** para outras classes
- **Nunca será instanciada**
- Apenas fornece um modelo para a geração de outras classes
- Ela nunca está completa... servirá apenas para a criação de funcionalidade genéricas de classes filhas
- Também conhecida como **superclasse**



# Classe Abstrata - exemplo de implementação

- É sabido que **Pessoa Física** e **Pessoa Jurídica** possuem o atributo nome como uma informação comum, porém, dentre dezenas de informações, a que gera uma grande diferença entre as duas são **CPF** (pessoa física) e **CNPJ** (pessoa jurídica)
- Ao invés de definir o nome para as duas classes (redundância), cria-se uma classe abstrata e insere um atributo nome dentro dela
- Haverá a herança das propriedades para as classes filhas - Física (atributo CPF) e Jurídica (atributo CNPJ)
- O atributo nome vem **automaticamente** pela **superclasse**

# Classe Abstrata - especificação

```
1 | public abstract class Pessoa{  
2 |     protected String xNome;  
3 |     protected Pessoa(){  
4 |         xNome = "Sem nome";  
5 |     }  
6 |     protected Pessoa(String nome){  
7 |         xNome = nome;  
8 |     }  
9 |     public String getName(){  
10 |         return xNome;  
11 |     }  
12 | }  
13 | }
```

```
• • 14 public class Fisica extends Pessoa{  
• • 15     private String xCPF;  
• • 16     public Fisica(){  
• + 17         super();  
• • 18     }  
+ • 19     public Fisica(String nome){  
20         super(nome);  
21     }  
22     public String getCPF(){  
23         return xCPF;  
24     }  
25 }  
26
```

Sub Classe "Fisica"

```
27 public class Juridica extends Pessoa{  
28     private String xCNPJ;  
29     public Juridica(){  
30         super();  
31     }  
32     public Juridica(String nome){  
33         super(nome);  
34     }  
35     public String getCNPJ(){  
36         return xCNPJ;  
37     }  
38     public String getName(){  
39         return super.getName();  
40     }  
41 }  
42
```

Sub Classe "Jurídica"

# Classe Abstrata - Testando os métodos

```
43 class Principal{  
44     public static void main(String[] args){  
45         Fisica pessoa1 = new Fisica("Daniel");  
46         System.out.println (pessoa1.getNome());  
47         System.out.println (pessoa1.getCPF());  
48         Juridica pessoa2 = new Juridica();  
49         System.out.println (pessoa2.getNome());  
50         System.out.println (pessoa2.getNPJ());  
51     }  
52 }
```

# Modificadores de Acesso **public, protected e private**

“ Em POO, **modificador de acesso** é a palavra-chave que define como um atributo, método ou classe será visto no contexto que estiver inserido

# Modificadores de Acesso

- Geralmente utilizam-se modificadores de acesso para privar os atributos do acesso direto, tornando-os privados
- Implementa-se métodos públicos que acessar e alteram os atributos
- Métodos privados são usados apenas por outros métodos que são públicos, e que podem ser chamados a partir de outro objeto da mesma classe a fim de não repetir código em mais de um método

# Modificadores de Acesso

- **Public:** visível para todas as outras classes, subclasses e pacotes do projeto Java
- **Protected:** visível para todas as outras classes e subclasses que pertencem ao mesmo pacote. O pacote da classe não tem acesso ao membro
- **Private:** visível apenas na própria classe
- **Package-Private:** modificador padrão quando outro não for definido.
  - Acessível na própria classe, nas classes e subclasses do mesmo pacote.
  - Normalmente utilizado para **construtores** e **métodos** que só devem ser acessados pelas classes e subclasses do pacote, constantes estáticas que são úteis apenas dentro do pacote em que estiver inserido

## Exemplo

```
1 public class MinhaClasse { //classe public
2     private int inteiro; //atributo inteiro private
3     protected float decimal; //atributo float protected
4     boolean ativado; //atributo booleano package-private
5 }
```

# Resumo

- Por padrão, a linguagem Java permite acesso aos membros apenas ao pacote em que ele se encontra
- Definições:

Modificador	Classe	Pacote	Subclasse	Globalmente
Public	Sim	Sim	Sim	Sim
Protected	Sim	Sim	Sim	Não
Sem modificador (padrão)	Sim	Sim	Não	Não
Private	Sim	Não	Não	Não

# Encapsulamento definição

# Encapsulamento

- Encapsulamento vem de **encapsular** (em POO significa "separar em partes", o mais isolado possível)
- A ideia é tornar o software mais **flexível**, fácil de modificar e de criar novas implementações
- Serve para **controlar o acesso** aos atributos e métodos de uma classe
- Forma mais eficiente de **proteger os dados** manipulados dentro da classe, além de determinar onde esta classe poderá ser manipulada
- Exemplo:
  - *private* (nível mais restritivo) - utilizados para um membro particular
  - Não se deve permitir acesso público aos membros, exceto em casos de constantes

# Encapsulamento

- Dividido em dois níveis:
  - **Nível de classe:** acesso de uma **classe inteira** que pode ser *public* ou *package-private* (padrão)
  - **Nível de membro:** acesso de **atributos** ou **métodos** de uma classe que podem ser *public*, *private*, *protected* ou *package-private* (padrão)
- Para ter um método encapsulado utiliza-se um modificador de acesso que geralmente é *public*, além do tipo de retorno.
- Para ter acesso a algum atributo ou método que esteja encapsulado, utiliza-se o conceito de *get* (recupera o valor do atributo) e *set* (atribuição de valor a um atributo)

exemplo

```
1 | private String atributo1 = new String();  
2 | private String atributo2 = new String();  
3 | public String getAtributo1(){  
4 |     return this.atributo1;  
5 | }  
6 | public String getAtributo2(){  
7 |     return this.atributo2;  
8 | }
```

# Exemplo:

```
1 public class Pessoa{  
2     private String nome;  
3     private String sobrenome;  
4     private String dataNasc;  
5     private String rg;  
6     private String[] telefones;  
7  
8     public String getName(){  
9         return nome;  
10    }  
11    public void setName(String n){  
12        nome = n;  
13    }  
14    public String getSobrenome(){  
15        return sobrenome;  
16    }  
17    public void setSobrenome(String s){  
18        sobrenome = s;  
19    }
```

Arrays em Java  
(será abordado posteriormente)

continua...

## Continuação...

```
20 |     public String getDataNasc(){
21 |         return dataNasc;
22 |     }
23 |     public void setDataNasc(String d){
24 |         dataNasc = d;
25 |     }
26 |     public String getRg(){
27 |         return rg;
28 |     }
29 |     public void setRg(String r){
30 |         r = rg;
31 |     }
32 |     String[]
33 |     public String getTelefones(){
34 |         return telefones;
35 |     }
36 |     public void setTelefones(String[] telefones){
37 |         telefones[] = telefones;
38 |     }
```

Arrays em Java  
(será abordado posteriormente)

# Polimorfismo

## definição

**“Polimorfismo** é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a **mesma identificação, assinatura, mas com comportamentos distintos**, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse

# Polimorfismo

- **Polimorfismo** significa “muitas formas”
- Capacidade de um objeto poder ser referenciado de várias formas (maneira como nos referimos a ele)
- Promove a reutilização contínua dos códigos, possibilita algo assumir várias formas
- Termo definido em linguagens orientadas a objeto como **Java, C# e C++**
- Permite ao desenvolvedor usar o mesmo elemento de formas diferentes
- Denota uma situação na qual um objeto pode se comportar de maneiras diferentes ao receber uma mensagem
- **Tanto o polimorfismo quanto herança são referências no ramo de reutilização de código, pois trabalham em conjunto**
- Existem dois tipos de polimorfismo:
  - Sobrecarga (*overload*)
  - Sobreposição (*override*)

# Sobrecarga de métodos (*overload*)

- Conceito de polimorfismo que consiste basicamente em criar variações de um mesmo método (métodos com nomes totalmente iguais em uma classe, porém, com argumentos diferentes)
- A decisão sobre qual método deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução (**mecanismo de ligação tardia**)
  - **Ligação tardia** ocorre quando o método a ser invocado é definido durante a execução do programa
- Através do mecanismo de sobrecarga, dois métodos de uma classe podem ter o mesmo nome, desde que suas assinaturas sejam diferentes

# Sobrecarga de métodos (*overload*)

- Vamos implementar uma calculadora simples que some apenas dois valores do mesmo tipo por vez.

```
1 public class calculadora{  
2     public int calcula( int a, int b){  
3         return a+b;  
4     }  
5     public double calcula( double a, double b){  
6         return a+b;  
7     }  
8     public String calcula( String a, String b){  
9         return a+b;  
10    }
```

O método calcula sobrecarregado com variações de tipos de soma

# Sobrecarga de métodos (*overload*)

- Exemplo completo

```
1 public class calculadora{  
2     public int calcula(int a,int b){  
3         return a+b;  
4     }  
5     public double calcula(double a,double b){  
6         return a+b;  
7     }  
8     public String calcula(String a,String b){  
9         return a+b;  
10    }  
11    public static void main(String args[]){  
12        calculadora calc= new calculadora();  
13        System.out.println(calc.calcula(1,1));  
14        System.out.println(calc.calcula(2.0,6.1));  
15        System.out.println(calc.calcula("vi","ram?"));  
16    }  
17 }
```

# Sobrecarga em Construtores

- **Sobrecarga** é muito utilizada em **construtores**, pois consistem em linhas de códigos que serão sempre executadas quando uma classe for instanciada (criação do objeto)

# Sobrecarga em Construtores

```
1 public class calculadora{  
2  
3     private String modelo;  
4     private String marca;  
5     private String uso;  
6  
7     //Sobrecarga de construtores.  
8     public calculadora(){  
9         // esse é o construtor padrão que o programa cria para todas as classes.  
10    }  
11    public calculadora(String marca, String modelo){  
12        this.marca=marca;  
13        this.modelo=modelo;  
14    }  
15  
16    public calculadora(String marca, String modelo, String uso){  
17        this.marca=marca;  
18        this.modelo=modelo;  
19        this.uso=uso;  
20    }
```

continua...

## continuação...

```
21 public int calcula(int a,int b){  
22     return a+b;  
23 }  
24 public double calcula(double a,double b){  
25     return a+b;  
26 }  
27 public String calcula(String a,String b){  
28     return a+b;  
29 }  
30 public static void main(String args[]){  
31     calculadora calc= new calculadora("optpex","N110","Empresarial");  
32     calculadora cald= new calculadora("Zion","Neo1");  
33     System.out.println(calc.calcula(900,1000));  
34     System.out.println(calc.calcula(99.0,100.1));  
35     System.out.println(calc.calcula("Sobrecarga de "," construtores"));  
36     System.out.println("calculadora 1 Marca: "+calc.marca+" Modelo:  
37         "+calc.modelo+" Uso: "+calc.uso);  
38     System.out.println("calculadora 2 Marca: "+cald.marca+" Modelo:  
39         "+cald.modelo);  
40 }  
41 }
```

# Sobreposição de métodos (*override*)

- Conceito de **polimorfismo** que nos permite reescrever um método, ou seja, podemos reescrever nas classes filhas métodos criados inicialmente na classe pai
- Os métodos que serão sobrepostos, diferentemente dos sobrecarregados, devem possuir o mesmo nome, tipo de retorno e quantidade de parâmetros do método inicial
- Será implementado como especificações da classe atual, podemos adicionar algo a mais
- Veremos a **Classe Veículo** (a seguir)

## Classe Veículo

```
1 public abstract class Veiculo{  
2  
3     public String modelo;  
4     public float velocidade;  
5     public int passageiros;  
6     public float combustivel;  
7  
8     public Veiculo(){  
9    }  
10    public Veiculo(String m,int p,float c){  
11        this.modelo=m;  
12        this.passageiros=p;  
13        this.combustivel=c;  
14    }  
15}
```

• • continua...

• • • • • +  
• • Continuação...

```
16     public void setVelocidade(float v){  
17         velocidade=v;  
18     }  
19     public float getVelocidade(){  
20         return velocidade;  
21     }  
22     public void setPassageiros(int p){  
23         passageiros=p;  
24     }  
25     public int getPassageiros(){  
26         return passageiros;  
27     }  
28     public void acelera(){  
29         System.out.println("Acelera");  
30     }  
31     public void freia(){  
32         System.out.println("Freia");  
33     }  
34 }
```

# Exemplo de Sobreposição

```
1 public class carro extends Veiculo
2     private int marcha;
3     public carro(){
4 }
5     public carro(String m,int p,float c)
6         super(m,p,c);
7     }
8     public void setVelocidade(float v)
9         velocidade=v;
10    if(velocidade>20 && velocidade<40){
11        marcha=2;
12    }
13    if(velocidade<20)
14        marcha=1;
15    }
16
17    if(velocidade>40 && velocidade<60){
18        marcha=3;
19    }
20
21    if(velocidade>60 && velocidade<70){
22        marcha=4;
23    }
24
25    if(marcha>70){
26        marcha=5;
27    }
28
29 }
```

```
30
31     public void acelera(){
32         setVelocidade(getVelocidade()+2.f);
33     }
34     public void freia(){
35         setVelocidade(getVelocidade()-2.f);
36     }
37     public static void main(String args[]){
38         carro corsa= new carro("Hatch",5,50.f);
39         carro audi= new carro();
40
41         int a,b=10;
42         System.out.println("Modelo: "+corsa.modelo+" Total de passageiros:
43             "+corsa.passageiros+" Tanque de combustivel:
44             "+corsa.combustivel+" Litros \n");
45         System.out.println("Acelerando o carro\n");
46
47         for(a=0;a<=10;a++){
48             corsa.acelera();
49             System.out.println("marcha: "+corsa.marcha+" Velocidade:
50                 "+corsa.getVelocidade());
51         }
52         System.out.println("\nFreiando o carro\n");
53         while(b!=0){
54             corsa.freia();
55             System.out.println("marcha: "+corsa.marcha+" Velocidade:
56                 "+corsa.getVelocidade());
57             b--;
58         }
59     }
60 }
```

# Polimorfismo

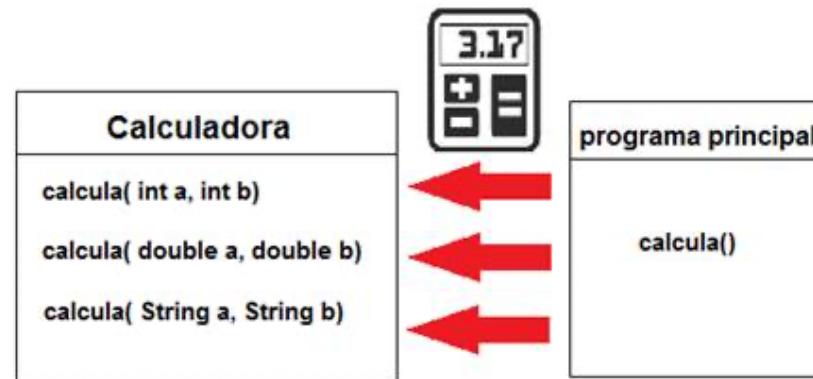
- Em POO, **polimorfismo** denota uma situação na qual um objeto pode se **comportar de maneira diferente ao receber uma mensagem**, dependendo do seu tipo de criação
- Exemplo:
  - A operação “mover” quando aplicada a uma janela de um sistema de interfaces tem um comportamento distinto do que quando aplicada a uma peça de um jogo de xadrez
  - A linguagem de programação deve ser capaz de selecionar o método correto a partir do nome da operação, classe do objeto e argumento para a operação
  - **Em Java, o polimorfismo se manifesta apenas em chamadas de métodos**

## Exemplo

```
abstract class Mamífero {  
    public abstract double obterCotaDiariaDeLeite();  
}  
  
class Elefante extends Mamífero {  
    public double obterCotaDiariaDeLeite(){  
        return 20.0;  
    }  
}  
  
class Rato extends Mamifero {  
    public double obterCotaDiariaDeLeite() {  
        return 0.5;  
    }  
}  
  
class Aplicativo {  
    public static void main(String args[]){  
        System.out.println("Polimorfismo\n");  
        Mamifero mamifero1 = new Elefante();  
        System.out.println("Cota diaria de leite do elefante: " + mamifero1.obterCotaDiariaDeLeite());  
        Mamifero mamifero2 = new Rato();  
        System.out.println("Cota diaria de leite do rato: " + mamifero2.obterCotaDiariaDeLeite());  
    }  
}
```

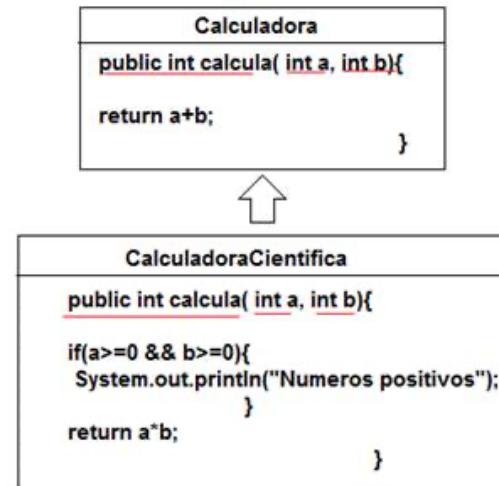
# Comparativo entre Sobrecarga e Sobreposição

- Sobrecarga está ligada a variância de estados de um método (conjunto de opções que o programa principal tem para escolher quando recebe os parâmetros passados pelo usuário)



# Comparativo entre Sobrecarga e Sobreposição

- Sobreposição funciona por meio de herança (podemos ter um método genérico e transformá-lo em específico), implementando novas funcionalidades pertinentes a classe à qual ele está. O nome do método e a lista de argumentos devem ser totalmente iguais aos da classe herdada.



# OBRIGADO

FIAP

Copyright © 2021 | Prof. Dr. Fernando Luiz de Almeida

Todos os direitos reservados. Reprodução ou divulgação total ou parcial deste documento, é expressamente  
proibido sem consentimento formal, por escrito, do professor/autor.



Até a próxima aula