

Aula 4: Métodos construtores e polimorfismo de sobrecarga

I Apresentação

Trabalharemos com os métodos construtores que são executados exclusivamente durante o processo de criação e instanciação do objeto.

Exploraremos também o conceito de polimorfismo de sobrecarga, aplicado sobre operadores e métodos. Ao final, desenvolveremos novas classes e aplicações utilizando os novos conceitos vistos.

Objetivos

- Identificar o conceito de métodos construtores;
- Examinar os conceitos de polimorfismo de sobrecarga.

Primeiras palavras

```
744         error' => $quote['sort_order'],
745     );
746     }
747 }
748
749 $sort_order = array();
750
751 foreach ($quotes as $key => $value) {
752     $sort_order[$key] = $value['sort_order'];
753 }
754
755 array_multisort($sort_order, SORT_ASC, $quotes);
756
757 $this->session->data['lpa']['shipping_methods'] = $quotes;
758 $this->session->data['lpa']['address'] = $address;
759
760 if (empty($quotes)) {
761     $json['error'] = $this->language->get('
762         error_no_shipping_methods');
763 } else {
764     $json['quotes'] = $quotes;
765 }
766
767 if (isset($this->session->data['lpa']['shipping_method']) && !
768     empty($this->session->data['lpa']['shipping_method']) &&
769     isset($this->session->data['lpa']['shipping_method']['code'])
770 ) {
771     $json['selected'] = $this->session->data['lpa']['
772         shipping_method']['code'];
773 } else {
774     $json['selected'] = '';
775 }
776
777 } else {
778     $json['error'] = $this->language->get('error_shipping_methods');
779 }
780
781 $this->response->addHeader('Content-Type: application/json');
```

```
382     if (this.paused = function (e) {
383         if (this.paused = true)
384             return;
385         if (this.$element.find('.next, .prev').length && $.support.transition) {
386             this.$element.trigger($.support.transition.end)
387             this.cycle(true);
388         }
389         this.interval = clearInterval(this.interval);
390         return this;
391     });
392
393     Carousel.prototype.next = function () {
394         if (this.sliding) return;
395         return this.slide('next');
396     };
397
398     Carousel.prototype.prev = function () {
399         if (this.sliding) return;
400         return this.slide('prev');
401     };
402
403     Carousel.prototype.slide = function (type, next) {
404         var $active = this.$element.find('.item.active');
405         var $next = next || this.getItemForDirection(type, $active);
406         var isCycling = this.interval;
407         var direction = type == 'next' ? 'left' : 'right';
408         var fallback = type == 'next' ? 'first' : 'last';
409         var that = this;
410
411         if (!$next.length) {
412             if (!this.options.wrap) return;
413             $next = this.$element.find('.item')[fallback]();
414         }
415         if ($next.hasClass('active')) return (this.sliding = false);
416
417         var relatedTarget = $next[0];
418         var slideEvent = $.Event('slide.bs.carousel', {
419             relatedTarget: relatedTarget,
420             direction: direction
421         });
422         this.$element.trigger(slideEvent);
423     };
424 }
```

Fonte: [Freepik](#)

A programação orientada a objetos permite que possamos controlar a criação de um objeto através dos chamados métodos construtores. Tal característica permite que um método especial, o método construtor, seja executado no momento em que ocorre a criação do objeto (objeto é instanciado) e um conjunto de ações (instruções) podem ser programadas para serem realizadas neste momento.

Entre essas ações, pode-se destacar o recebimento de dados iniciais para serem atribuídos e/ou preparar o objeto para que este esteja apto a atender às necessidades para qual foi criado.

Um método construtor pode ainda ser usado para determinar o tamanho de um vetor que será usado pelo objeto, assim como pré-configurar estruturas de dados de suporte ao objeto que está sendo criado.

Métodos construtores

São métodos especiais executados apenas uma vez por cada objeto criado, pois somente são executados no momento da instanciação / criação do objeto, sendo responsáveis por realizar as ações necessárias para a sua criação (controlar a criação do objeto).

Características dos métodos construtores:

1

São sempre públicos (*public*, característica de encapsulamento – veremos mais adiante), não podendo ter nenhum tipo de restrição;

2

Não existe definição de tipo de retorno, pois métodos construtores não podem retornar valores com a instrução “*return*”, são sem tipo;


3

Devem ser identificados sempre com o mesmo nome da classe;

4

São executados exclusivamente durante o processo de criação / instanciação do objeto, não podendo ser usados pelo objeto após a sua criação.

 [Classe: Pessoa](#)

 Clique no botão acima.

Classe: Pessoa

```
public class Pessoa {
    String nome, identidade;
    int idade;
    public Pessoa(String nome, String identidade, int idade) {
        setNome( nome );
        setIdentidade( identidade );
        setIdade( idade );
    }
}
```

```

    public String getNome() {
        return nome;
    }

    public void setNome( String no ) {
        if (!no.isEmpty()){
            nome = no;
        }
    }

    public String getIdentidade() {
        return identidade;
    }

    public void setIdentidade( String id ) {
        if (!id.isEmpty()){
            identidade = id;
        }
    }

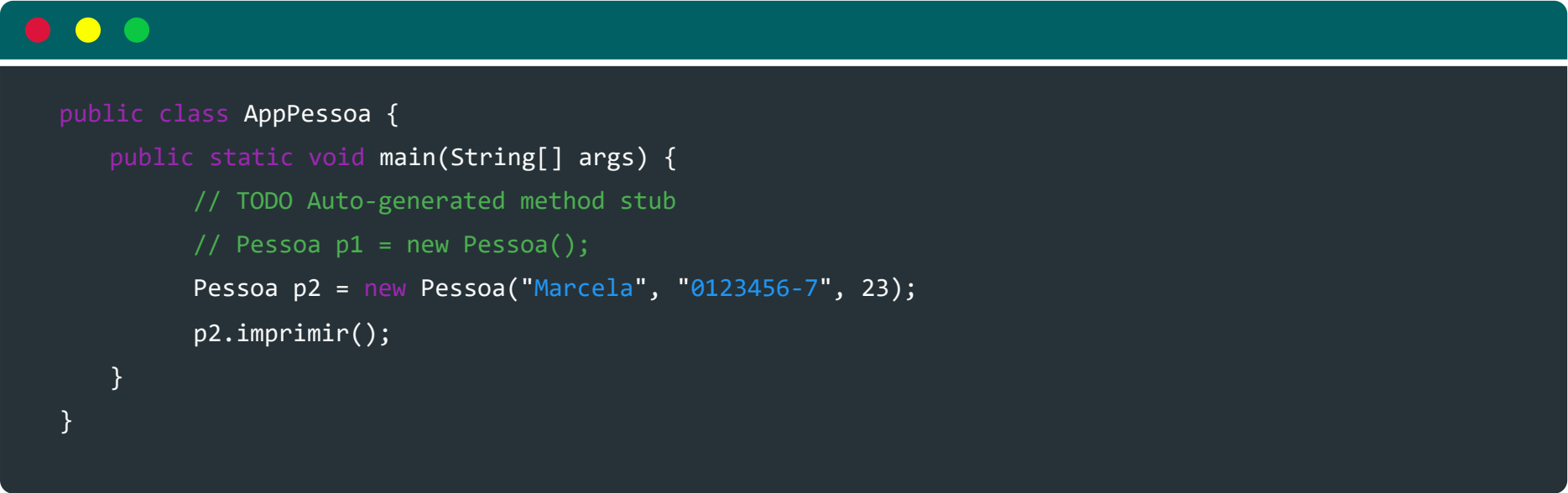
    public int getIdade() {
        return idade;
    }

    public void setIdade( int id ) {
        if (id > 0){
            idade = id;
        }
    }

    public void imprimir() {
        System.out.println("Pessoa:");
        System.out.println("Nome = " + nome);
        System.out.println("Identidade = " + identidade);
        System.out.println("Idade = " + idade);
    }
}

```

Aplicação: AppPessoa



```

public class AppPessoa {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // Pessoa p1 = new Pessoa();
        Pessoa p2 = new Pessoa("Marcela", "0123456-7", 23);
        p2.imprimir();
    }
}

```

Saída: Console

```

Pessoa:
Nome = Marcela
Identidade = 0123456-7
Idade = 23

```

01 Na classe Pessoa, o método construtor:

```
public Pessoa(String nome, String identidade, int idade)
```

O método é público, não possui tipo de retorno antes no nome identificador do método, seu identificador é igual ao nome da classe, por isso começou por letra maiúscula e só será usado para criar o objeto (instanciar);

02 A partir do momento em que um método construtor é criado, a classe só poderá ser instanciada se usarmos um método construtor existente. Por isso, o objeto Pessoa p1 não pode ser criado e sua criação foi comentada na aplicação, pois este método tenta utilizar um método construtor que não existe na classe;

03 O objeto p2 usa um método construtor existente e por isso pode ser criado;

04 Com o uso do método construtor, os dados recebidos como parâmetros puderam ser utilizados para realizar as atribuições nos atributos do objeto, determinando os valores de suas propriedades no momento da criação do objeto;

05 Os métodos setIdentidade (String id) e setIdade (int id) podem ter o mesmo identificador para o parâmetro porque o parâmetro id é declarado em diferentes métodos e, sendo assim, ele é válido internamente em cada um dos métodos separadamente.

O processo de compilação de uma classe cria um método construtor vazio quando não for encontrado nenhum método construtor. Desta forma, nos exemplos anteriores, as classes Aluno e Carro não tinham métodos construtores, então o compilador criou respectivamente os métodos a seguir para as classes Aluno e Carro:


```
public Aluno () {}  
  
e  
  
public Carro () {}
```

Quando não temos um construtor em uma classe, um construtor VAZIO é criado no processo de compilação.

Polimorfismo de sobrecarga

Polimorfismo quer dizer muitas formas. O polimorfismo de sobrecarga permite o emprego de operadores e identificadores de várias formas, sendo então necessária uma contextualização para que seja realizada a operação adequada. Este contexto está ligado ao emprego do operador, método etc., de acordo com uma situação.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

 Clique nos botões para ver as informações.

```
int b=5, c=3;
int a = b + c; // o símbolo de + neste contexto está realizando uma operação
               // de soma entre valores inteiros, no caso b + c (inteiros)
double x=3.5, y = 2.2;
double z = x + y; // temos um novo contexto, pois a operação agora será
                  // realizada entre dois valores reais, no caso x + y
```

A mudança de contexto faz com que as operações a serem realizadas sejam diferentes, pois toda linguagem de programação possui diferentes formas de realizar as operações de soma inteira e real. Desta forma, a expressão aritmética a seguir utiliza as duas operações conjuntamente:

```
double z = ( 2 + 5 ) / ( 3.5 + 1.5 );
```

Na primeira operação de soma, os operandos são inteiros, então a operação a ser realizada será de uma soma inteira, para somente depois ser realizada a operação de soma real. Desta forma, teremos em um instante a seguinte situação:

```
double z = ( 7 ) / ( 5.0 );
```

Assim, a operação de divisão será real e não inteira porque existe um operando real.

Comentário

Em Java, todas as operações aritméticas serão realizadas em função dos tipos dos operandos, e a operação será inteira apenas se ambos os operandos foram inteiros. Caso contrário (um operando inteiro e outro real ou dois operandos reais), a operação será real.

O operador + é um dos mais usados, sendo um bom exemplo de sobrecarga de operadores, pois pode ser utilizado de várias e diferentes formas em função do contexto:

1. Concatenação: String nome = "João" + " da " + "Silva";
2. Soma inteira: int a = 3 + 4;
3. Soma real: double b = 1.3 + 2.7;
4. Incremento: x++; ou ++x;
5. Concatenação entre textos e valores: System.out.println("Idade" + p2.getIdade());

Agora imagine a seguinte instrução:

```
System.out.println("Valor =" + (( 3 + 4 ) + ( 1.3 + 2.7 ) + (++x)));
```

A sobrecarga de operadores está sendo usada de diferentes formas em uma mesma instrução. Cada contexto será avaliado para que seja executada a operação adequada em cada caso.

Toda expressão é avaliada sintaticamente, assim como cada contexto será avaliado individualmente no momento da execução.

Polimorfismo de sobrecarga de métodos



A sobrecarga de métodos permite que possamos ter mais de um método com o mesmo identificador em uma mesma classe. Isso só é possível em razão da avaliação do contexto no momento da execução. Vamos levar em consideração que eu desejo realizar o cálculo da área de um quadrado e de um retângulo em uma mesma classe.

Para realizar o cálculo da área do quadrado, eu preciso apenas do valor da base do quadrado. Assim, o método área ficaria da seguinte forma:

```
public int area ( int base ){  
    return ((int) Math.pow(base,2)); // Math.Pow calcula a base elevada a 2  
}
```

Já para realizar o cálculo da área do retângulo, eu preciso do valor da base e da altura do quadrado. Assim, o método área ficaria da seguinte forma:

```
public int area ( int base, int altura ){  
    return ( base * altura );  
}
```

Assinaturas

Esses dois métodos podem conviver na mesma classe, uma vez que eles possuem diferentes assinaturas. A assinatura de um método é determinada pelo tipo de parâmetros e pela ordem em que estes foram declarados. Desta forma, a assinatura do primeiro método é:

```
area ( int );
```

e do segundo:

```
area ( int , int );
```

Diante da diferença de assinaturas, podemos ter dois diferentes contextos para o uso do método de cálculo da área:

System.out.println("Área = " + area(5));

No primeiro contexto, é chamado para executar o método área com um único parâmetro e neste caso a avaliação em tempo de execução irá determinar que deve ser usado o cálculo da área do quadrado. Ou seja, aquele que recebe um valor inteiro como parâmetro, e a resposta será: 25.


System.out.println("Área = " + area(5, 6));

No segundo contexto, é chamado para executar o método área com um dos parâmetros e, neste caso, a avaliação em tempo de execução irá determinar que deve ser usado o cálculo da área do retângulo. Ou seja, aquele que recebe dois valores inteiros como parâmetro, e a resposta será: 30.

Atenção

Com o uso da sobrecarga de métodos você poderá criar quantos métodos com o mesmo identificador (nome) quiser em uma mesma classe, desde que eles não possuam a mesma assinatura de método.

Métodos

 Clique no botão acima.

Se quisesse incluir um método para calcular a área de uma circunferência, você não poderia incluir nesta classe, pois ele teria a mesma assinatura do método do cálculo da área do quadrado:

```
public static int area ( int raio ){
    return ((int) Math.PI * Math.pow(raio,2));
    // Math.Pow calcula o raio elevado a 2
}
```

O método teria a mesma assinatura do método área do quadrado e, no momento da execução, não haveria como saber qual dos dois deveria ser executado, pois ambos teriam o mesmo contexto:

```
System.out.println("Área Quadrado = " + area( 5 ));
```

```
System.out.println("Área Circunferência = " + area( 4 ));
```

A linguagem Java não teria como definir qual método executar, já que ambos têm a mesma assinatura e a linguagem não é suficientemente inteligente para tentar buscar isso em algum outro lugar que não o contexto da chamada do método:

```
area( 5 )
    e
area( 4 )
```

Como ambos possuem o mesmo contexto, os métodos com a mesma assinatura não podem compartilhar a mesma classe.

Exemplos de polimorfismo de sobrecarga válidos para uma mesma classe:

```
int meuMetodo( int a, double b, String c ) {
    return 1;
}
int meuMetodo ( double b, String c, int a ){
    return 2;
}
int meuMetodo ( String c, int a, double b ) {
    return 3;
}
int meuMetodo ( String c, double b, int a ) {
```

```
return 4;  
}
```

As assinaturas são respectivamente:

```
meuMetodo ( int , double , String )  
meuMetodo ( double , String , int )  
meuMetodo ( String , int , double )  
meuMetodo ( String , double , int )
```

Todos os métodos acima, apesar de possuírem a mesma quantidade de parâmetros, têm assinaturas diferentes que serão executadas em função de contexto diferentes, respectivamente:

```
int g = meuMetodo( 2, 2.25, "Casa" );  
int h = meuMetodo( 2.25, "Casa", 2 );  
int i = meuMetodo( "Casa", 2, 2.25 );  
int j = meuMetodo( "Casa", 2.25, 2 );
```

Os valores armazenados em g, h, i e j serão respectivamente: 1, 2, 3 e 4.

I A sobrecarga de métodos construtores

Métodos construtores são métodos e também podem ser sobrecarregados.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Uma classe que possui mais de um método construtor é uma classe que oferece diferentes formas de criação para os seus objetos.

Outra forma de uso de mais de um construtor é para manter a compatibilidade de uma classe com suas aplicações antigas.

Se analisarmos a classe Carro, vista como exemplo anteriormente, podemos notar que ela não possui nenhum método construtor. Podemos então criar alguns métodos construtores para esta classe e preservar a aplicação antiga, criando e analisando uma classe nova (evoluída) e as duas aplicações, a antiga e a nova:

Classe: Carro (atualizada com cinco métodos construtores).

```
import java.util.Scanner;

public class Carro {
    // use as regras da boa prática em programação Java
    // para os identificadores da classe, dos atributos e dos métodos
    String fabricante, modelo, cor, placa;
    double valor;
    int numeroPortas, anoFabricacao, anoModelo;

    public Carro() { }
    public Carro(String placa, double valor) {
        this.placa = placa;
        this.valor = valor;
    }

    public Carro(String modelo, String cor, String placa, double valor) {
        this.modelo = modelo;
        this.cor = cor;
        this.placa = placa;
        this.valor = valor;
    }

    public Carro(String fabricante, String modelo, String cor, String placa, double valor) {
        this.fabricante = fabricante;
        this.modelo = modelo;
        this.cor = cor;
        this.placa = placa;
        this.valor = valor;
    }

    public Carro(String fabricante, String modelo, String cor, String placa, double valor, int
numeroPortas, int anoFabricacao, int anoModelo) {
        this.fabricante = fabricante;
        this.modelo = modelo;
        this.cor = cor;
        this.placa = placa;
        this.valor = valor;
        this.numeroPortas = numeroPortas;
        this.anoFabricacao = anoFabricacao;
        this.anoModelo = anoModelo;
    }

    public String getFabricante () {
        return fabricante;
    }

    public void setFabricante (String fab) {
        if(!fab.isEmpty()) {
            fabricante = fab;
        }
    }
}
```

```
}

public String getModelo () {
    return modelo;
}

public void setModelo (String mod) {
    if(!mod.isEmpty()) {
        modelo = mod;
    }
}

public String getCor () {
    return cor;
}

public void setCor (String co) {
    if(!co.isEmpty()) {
        cor = co;
    }
}

public String getPlaca () {
    return placa;
}

public void setPlaca (String pla) {
    if(!pla.isEmpty()) {
        placa = pla;
    }
}

public double getValor () {
    return valor;
}

public void setValor (double val) {
    if(val > 0) {
        valor = val;
    }
}

public int getNumeroPortas () {
    return numeroPortas;
}

public void setNumeroPortas (int nump) {
    if(nump > 0) {
        numeroPortas = nump;
    }
}

public int getAnoFabricacao () {
    return anoFabricacao;
}

public void setAnoFabricacao (int anof) {
    if(anof > 0) {
        anoFabricacao = anof;
    }
}
```

```

    }

    public int getAnoModelo () {
        return anoModelo;
    }

    public void setAnoModelo (int anom) {
        if(anom > 0) {
            anoModelo = anom;
        }
    }

    public void imprimir (){
        System.out.println( "Fabricante : " + getFabricante() );
        System.out.println( "Modelo : " + getModelo() );
        System.out.println( "Cor : " + getCor() );
        System.out.println( "Placa : " + getPlaca() );
        System.out.println( "Valor : " + getValor() );
        System.out.println( "Número de Portas : " + getNumeroPortas() );
        System.out.println( "Ano de fabricação: " + getAnoFabricacao() );
        System.out.println( "Ano do Modelo : " + getAnoModelo() );
    }

    public void entradaDados () {
        Scanner entrada = new Scanner( System.in );
        // O objeto Scanner deve ficar local ao método
        // o objeto Scanner para entrada de dados não é um atributo do carro
        // é apenas um objeto auxiliar a entrada de dados
        System.out.println("Digite o Fabricante do carro :");
        setFabricante( entrada.nextLine() );
        System.out.println("Digite o Modelo do carro :");
        setModelo( entrada.nextLine() );
        System.out.println("Digite a Cor do carro :");
        setCor( entrada.nextLine() );
        System.out.println("Digite a Placa do carro :");
        setPlaca( entrada.nextLine() );
        System.out.println("Digite o Valor do carro :");
        setValor( Double.parseDouble( entrada.nextLine() ) );
        System.out.println("Digite o Número de Portas do carro :");
        setNumeroPortas( Integer.parseInt( entrada.nextLine() ) );
        System.out.println("Digite o Ano de fabricação do carro :");
        setAnoFabricacao( Integer.parseInt( entrada.nextLine() ) );
        System.out.println("Digite o Ano do Modelo do carro :");
        setAnoModelo( Integer.parseInt( entrada.nextLine() ) );
    }
}

```

Aplicação antiga AppCarro.

```

public class AppCarro {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Carro car1 = new Carro();
        car1.entradaDados();
        car1.imprimir();
        Carro car2 = new Carro();
        car2.entradaDados();
    }
}

```

```
        car2.imprimir();  
        Carro car3 = new Carro();  
        car3.entradaDados();  
        car3.imprimir();  
    }  
}
```

A execução da aplicação não foi afetada pelas mudanças na classe porque foi criado o construtor vazio `public Carro () {}` que garantiu a compatibilidade:

Digite o Fabricante do carro :

GM

Digite o Modelo do carro :

Ônix

Digite a Cor do carro :

branca

Digite a Placa do carro :

GMG1C00

Digite o Valor do carro :

45000

Digite o Número de Portas do carro :

4

Digite o Ano de fabricação do carro :

2018

Digite o Ano do Modelo do carro :

2019

Fabricante : GM

Modelo : Ônix

Cor : branca

Placa : GMG1C00

Valor : 45000.0

Número de Portas : 4

Ano de fabricação: 2018

Ano do Modelo : 2019

Digite o Fabricante do carro :

VW

Digite o Modelo do carro :

Polo

Digite a Cor do carro :

preta

Digite a Placa do carro :

VWB0100

Digite o Valor do carro :

50000

Digite o Número de Portas do carro :

2

Digite o Ano de fabricação do carro :

2017

Digite o Ano do Modelo do carro :

2017

Fabricante : VW

Modelo : Polo

Cor : preta

Placa : VWB0100

Valor : 50000.0

Número de Portas : 2

Ano de fabricação: 2017

Ano do Modelo : 2017

Digite o Fabricante do carro :

Ford

Digite o Modelo do carro :

Ecosport

Digite a Cor do carro :

branca

Digite a Placa do carro :

FOR1D00

Digite o Valor do carro :

78000

Digite o Número de Portas do carro :

5

Digite o Ano de fabricação do carro :

2019

Digite o Ano do Modelo do carro :

2019

Fabricante : Ford

Modelo : Ecosport

Cor : branca

Placa : FOR1D00

Valor : 78000.0

Número de Portas : 5

Ano de fabricação: 2019

Ano do Modelo : 2019

Aplicação usando diferentes construtores para criar os objetos: AppCarro.

```
public class AppCarroNovo {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Carro car1 = new Carro();  
        car1.entradaDados();  
        car1.imprimir();  
        Carro car2 = new Carro("AAA1A00", 25000);  
        car2.imprimir();  
        Carro car3 = new Carro("Logan", "Azul", "ABC1E00", 32000);  
        car3.imprimir();  
        Carro car4 = new Carro("Audi", "A5", "Prata", "AUD0I00", 123000);  
        car4.imprimir();  
        Carro car5 = new Carro("Fiat", "Argo", "Verde", "ABB1I00", 42000, 5, 2018, 2019);  
        car5.imprimir();  
    }  
}
```

Execução da nova aplicação:

Digite o Fabricante do carro :

VW

Digite o Modelo do carro :

Virtus

Digite a Cor do carro :

Prata

Digite a Placa do carro :

RI00B01

Digite o Valor do carro :

60000

Digite o Número de Portas do carro :

4

Digite o Ano de fabricação do carro :

2019

Digite o Ano do Modelo do carro :

2019

Fabricante : VW

Modelo : Virtus

Cor : Prata

Placa : RI00B01

Valor : 60000.0

Número de Portas : 4

Ano de fabricação: 2019

Ano do Modelo : 2019

Fabricante : null

Modelo : null

Cor : null

Placa : AAA1A00

Valor : 25000.0

Número de Portas : 0

Ano de fabricação: 0

Ano do Modelo : 0

Fabricante : null

Modelo : Logan

Cor : Azul

Placa : ABC1E00

Valor : 32000.0

Número de Portas : 0

Ano de fabricação: 0

Ano do Modelo : 0

Fabricante : Audi

Modelo : A5

Cor : Prata

Placa : AUD0I00

Valor : 123000.0

Número de Portas : 0

Ano de fabricação: 0

Ano do Modelo : 0

Fabricante : Fiat

Modelo : Argo

Cor : Verde

Placa : ABB1I00

Valor : 42000.0
Número de Portas : 5
Ano de fabricação: 2018
Ano do Modelo : 2019

Notas:


- 01** O primeiro método construtor criado foi o vazio, para garantir a compatibilidade com a aplicação antiga;
- 02** Foram incluídos mais quatro métodos construtores seguindo o conceito da sobrecarga de métodos;
- 03** Na nova aplicação, foram criados cinco diferentes objetos, cada um usando um construtor diferente;
- 04** Ambas as aplicações funcionaram apesar da alteração;

I Polimorfismo de sobrecarga e a evolução das classes

Com o polimorfismo de sobrecarga podemos criar diferentes implementações para métodos com o mesmo identificador (nome) em uma mesma classe.

Vamos imaginar que uma classe chamada Login fosse usada por vários de seus sistemas:

 [Classe: Login](#)

 Clique no botão acima.

Aplicação antiga AppCarro.

```
public class Login {  
    String nome;  
    String nomeLogin;  
    String senha;  
    int nivelAcesso; // nível de acesso do usuário ao sistema  
    public Login( String nl, String s ){  
        setNivelAcesso( verificaLogin( nl, s ) );  
    }  
    public int verificaLogin( String nolog, String sem ){  
        int na=0;  
        if( nolog.equals( "carneiro5" ) && sen.equals( "123456" ) ){  
            na = 10;  
            setNome( "André" );  
        }  
    }  
}
```

```

    }
    else {
        na = 0;
    }
    return na;
}
public String getNome() {
    return nome;
}
public void setNome( String no ) {
    if( !no.isEmpty() ){
        nome = no;
    }
}
public String getNomeLogin() {
    return nomeLogin;
}
public void setNomeLogin( String nl ) {
    if( !nl.isEmpty() ){
        nomeLogin = nl;
    }
}
public String getSenha() {
    return senha;
}
public void setSenha( String sen ) {
    if( !sen.isEmpty() ){
        senha = sen;
    }
}
public int getNivelAcesso() {
    return nivelAcesso;
}
public void setNivelAcesso( int na ) {
    if( na >= 0 ){
        nivelAcesso = na;
    }
}
}
}

```

Aplicação: AppLogin.

```

public class AppLogin {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Login lg1 = new Login( "carneiro5" , "123456" );
        System.out.println( "O seu nome é: " + lg1.getNome() );
        System.out.println( "O seu nivel de acesso é: " + lg1.getNivelAcesso() );
        Login lg2 = new Login( "kkk" , "000000" );
        System.out.println( "O seu nome é: " + lg2.getNome() );
        System.out.println( "O seu nivel de acesso é: " + lg2.getNivelAcesso() );
    }
}

```

Execução:


O seu nome é: André
O seu nível de acesso é: 10
O seu nome é: null
O seu nível de acesso é: 0

Imagine a situação: você tem um novo cliente, e seus funcionários fazem o *login* não apenas com o nome de login e a senha, mas também utilizando um dispositivo eletrônico para geração de senhas (*token*).

A sua classe não iria funcionar com este novo contexto. Neste caso, a programação orientada a objetos nos ajuda muito, pois faremos uma atualização na classe Login e ela será capaz, não só de atender a esta nova demanda, mas de continuar a atender os antigos clientes.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

 [Classe atualizada: Login](#)

 Clique no botão acima.

```
public class Login {  
    String nome;  
    String nomeLogin;  
    String senha;  
    int nivelAcesso; // nível de acesso do usuário ao sistema  
    public Login(String nl, String s) {  
        setNivelAcesso(verificaLogin(nl, s));  
    }  
    public Login(String nl, String s, int token) {  
        setNivelAcesso(verificaLogin(nl, s, token));  
    }  
    public int verificaLogin(String nolog, String sem) {  
        int na = 0;  
        if (nolog.equals("carneiro5") && sem.equals("123456")) {  
            na = 10;  
            setNome("André");  
        }  
        else {  
            na = 0;  
        }  
        return na;  
    }  
    public int verificaLogin(String nolog, String sen, int tk) {
```

```

        int na = 0;
        if (nolog.equals("pereira") && sen.equals("246810") &&
            verificarToken(tk)) {
            na = 8;
            setNome("Maria");
        }
        else {
            na = 0;
        }
        return na;
    }
    public boolean verificarToken(int tk) {
        if (tk == 1000 || tk == 2000 || tk == 3000) {
            return true;
        }
        else {
            return false;
        }
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String no) {
        if (!no.isEmpty()) {
            nome = no;
        }
    }
    public String getNomeLogin() {
        return nomeLogin;
    }
    public void setNomeLogin(String nl) {
        if (!nl.isEmpty()) {
            nomeLogin = nl;
        }
    }
    public String getSenha() {
        return senha;
    }
    public void setSenha(String sem) {
        if (!sem.isEmpty()) {
            senha = sem;
        }
        this.senha = senha;
    }
    public int getNivelAcesso() {
        return nivelAcesso;
    }
    public void setNivelAcesso(int na) {
        if (na >= 0) {
            nivelAcesso = na;
        }
    }
}

```

Aplicação: AppLogin com um objeto usando o novo construtor.

```

public class AppLogin {

```



```
public static void main( String[] args ) {  
    // TODO Auto-generated method stub  
    Login lg1 = new Login( "carneiro5" , "123456" );  
    System.out.println( "O seu nome é: " + lg1.getNome() );  
    System.out.println( "O seu nível de acesso é: " + lg1.getNivelAcesso());  
  
    Login lg2 = new Login( "kkk" , "000000" );  
    System.out.println( "O seu nome é: " + lg2.getNome() );  
    System.out.println( "O seu nível de acesso é: " + lg2.getNivelAcesso() );  
  
    Login lg3 = new Login( "pereira" , "246810" , 2000 );  
    System.out.println( "O seu nome é: " + lg3.getNome());  
    System.out.println( "O seu nível de acesso é: " + lg3.getNivelAcesso() );  
}  
}
```

Execução:

```
O seu nome é: André  
O seu nível de acesso é: 10  
O seu nome é: null  
O seu nível de acesso é: 0  
O seu nome é: Maria  
O seu nível de acesso é: 8
```

Notas:

- 01** A classe Login agora possui dois diferentes construtores e dois diferentes métodos verificaLogin, ambos sobrecarregados. A versão anterior foi preservada porque ainda é usada pelos sistemas dos antigos clientes. Entretanto, com a inclusão dos novos métodos, a classe foi atualizada e também passou a atender ao cliente novo;
- 02** Foi incluído ainda o método verificarToken, que só é usado pelo cliente novo. Sua inclusão na classe não atrapalha em nada os sistemas dos clientes antigos;
- 03** A aplicação agora pode instanciar (criar) objetos das duas formas, com e sem o *token*, sem que uma atrapalhe a outra.

As classes na programação orientada a objetos evoluem conforme precisamos de mais atributos e métodos.

Entretanto, se mantivermos os métodos necessários para os sistemas mais antigos, essa evolução não afetará os outros sistemas e teremos uma melhoria na classe, facilitando a sua evolução e sua manutenção, uma vez que, ao realizar qualquer

melhoria em uma classe, basta recompilar as aplicações que estas se tornarão atualizadas.

Atividades

1) Dada a classe abaixo, implemente um método construtor que receba o nome, o peso e altura de um atleta.

```
public class Atleta {  
    String nome;  
    double peso, altura, imc;  
}
```

2) Analisando os métodos abaixo (para uma mesma classe), podemos afirmar que foi aplicado o conceito de polimorfismo de sobrecarga? Justifique.

```
double calculo(double p, double a) {  
    return p+a;  
}  
double calculo(double p, double a) {  
    return a*p;  
}  
double calculo(double p, double a, double t) {  
    return p-a+t;  
}
```

Notas

Referências

TEXTO

Próxima aula

- Herança.

Explore mais

- Pesquise na internet sites, vídeos e artigos relacionados ao conteúdo visto.

- Em caso de dúvidas, converse com seu professor online por meio dos recursos disponíveis no ambiente de aprendizagem.