

Construtores, Sobrecarga e Herança



Conteudista: Prof. Esp. Alexander Gobbato Albuquerque

Revisão Textual: Prof.^a Dra. Patrícia S. Leite Di Iório

Objetivos da Unidade:

- Abordar a importância dos construtores, sobrecarga e herança e como isso é de fácil implementação e adaptação.

Contextualização

Material Teórico

Material Complementar

Referências



Contextualização

Já estudamos, nos capítulos anteriores, o encapsulamento de informações, a reutilização de códigos. Vimos, também, como criar os métodos, usar encapsulamento e chamar os métodos criados através de objetos. Com os conhecimentos adquiridos, podemos utilizar as técnicas de criação de construtores, que são os métodos invocados na criação dos objetos. Veremos a sobrecarga de métodos, passando por novas assinaturas e também por herança, em que as características são passadas para as classes que desejam utilizar a codificação já desenvolvida.



Material Teórico

Construtores

Construtores são métodos invocados automaticamente quando um objeto é instanciado. São métodos que nunca retornam nada e não possuem tipo. Em todas as instanciações foram utilizadas a palavra reservada, conforme o exemplo abaixo:

```
Carro c1 = new Carro();
```

Conforme comentado nas unidades anteriores, a palavra *new* é a responsável por inicializar o objeto. O método construtor é o método que é chamado toda vez que o objeto é criado, ou seja, quando se utiliza o operador *new*, o primeiro método que é chamado é o construtor do objeto. Ele é responsável por disponibilizar ou alocar o espaço em memória para a utilização do novo objeto.

Obrigatoriamente, o método construtor deve possuir o mesmo nome da classe, se ele não for declarado, por *default* é criado automaticamente.

Logo abaixo demonstrado-se a codificação de dois arquivos:

```
class Carro{  
    private codigo;  
    private nome;  
    private marca;  
  
    Carro(){ // MÉTODO CONSTRUTOR, MESMO NOME DA CLASSE  
        marca = "VALOR PADRÃO"; //INICIALIZA A MARCA COM UM VALOR PADRÃO  
    }  
  
    public void MostraMarca(){  
        System.out.println("Marca: " + marca);  
    }  
}
```

Figura 1 – Arquivo classe

Fonte: Reprodução

```
class UsaCarro{  
  
    public static void main(String args[]){  
        Carro c1 = new Carro(); //INVOCAR O MÉTODO CONSTRUTOR  
        c1.MostraMarca();  
    }  
}
```

Figura 2 – Arquivo para instanciar a classe e invocar o método construtor

Fonte: Reprodução

O método construtor funciona como qualquer método criado em um programa Java, pode receber argumentos no momento da criação, veja os exemplos abaixo.

Suponhamos a necessidade de criar dois objetos da classe carro, com diferentes conteúdos para a variável nome.

```
class Carro{  
    private codigo;  
    private nome;  
    private marca;  
  
    Carro(String n){ // MÉTODO CONSTRUTOR, MESMO NOME DA CLASSE  
        nome = n; //INICIALIZA O NOME COM UM VALOR INFORMADO NO PROGRAMA  
        marca = "VALOR PADRÃO"; //INICIALIZA A MARCA COM UM VALOR PADRÃO  
    }  
  
    public void ExibeInf(){  
        System.out.println("Nome: " + nome);  
        System.out.println("Marca: " + marca);  
    }  
}
```

Figura 3 – Classe carro com recebimento de argumentos

Fonte: Reprodução

```
class UsaCarro{  
  
    public static void main(String args[]){  
        Carro c1 = new Carro("Fiesta"); //INVOCA O MÉTODO CONSTRUTOR  
        Carro c2 = new Carro("EcoSport"); //INVOCA O MÉTODO CONSTRUTOR  
        c1.ExibeInf();  
        c2.ExibeInf();  
    }  
}
```

Figura 4 – Classe UsaCarro com passagem de valores

Fonte: Reprodução

Em Java, existe uma referência para a própria classe através da palavra-chave “this”. Assim, no construtor ao invés de usar “`nome = n`”, podemos usar “`this.nome = nome`”, nesse caso as mesmas variáveis no construtor podem ser usadas nos atributos.

```

class Carro{
    private codigo;
    private nome;
    private marca;

    Carro(String nome, String marca){ // MÉTODO CONSTRUTOR, MESMO NOME DA CLASSE
        this.nome = nome; //INICIALIZA O NOME COM UM VALOR INFORMADO NO PROGRAMA
        this.marca = marca; //INICIALIZA A MARCA COM UM VALOR PADRÃO
    }

    public void ExibeInf(){
        System.out.println("Nome: " + nome);
        System.out.println("Marca: " + marca);
    }
}

```

Figura 5

Fonte: Reprodução

Agora, quando instanciamos um objeto da classe Carro, veja como fica:

```

class UsaCarro{

    public static void main(String args[]){
        Carro c1 = new Carro("Fiesta","Ford"); //INVOCAR O MÉTODO CONSTRUTOR
        Carro c2 = new Carro("EcoSport","Ford"); //INVOCAR O MÉTODO CONSTRUTOR
        c1.ExibeInf();
        c2.ExibeInf();
    }
}

```

Figura 6

Fonte: Reprodução

Sobrecarga

Sobre carregar (do inglês *overload*) trata-se de um método para criar mais métodos com o mesmo nome, porém com assinaturas diferentes. Os parâmetros podem se diferenciar em tipo e/ou em quantidade, assim como o tipo de retorno. Ficará a cargo do compilador escolher de acordo com a lista de argumentos enviados os métodos a serem executados. Vejamos os exemplos abaixo:

```
public class Operadores{  
    public void multiplicar(float num1, float num2){  
        System.out.println("Multiplicação: " + num1 * num2);  
    }  
}
```

Figura 7

Fonte: Reprodução

O método recebe dois parâmetros e o retorno é o resultado impresso na tela, veja o método implementado.

```
public class UsaOperadores{  
    public static void main(String args[]){  
        Operadores op = new Operadores();  
        op.multiplicar(1.5, 3.85);  
    }  
}
```

Figura 8

Fonte: Reprodução

Agora, imagine que precisamos fazer um método que multiplique números inteiros e não números reais. Para o usuário, é transparente, o programador deve realizar uma sobrecarga de métodos e desenvolveria da seguinte forma:

```
public class Operadores{  
    public void multiplicar (float num1, float num2){  
        System.out.println("Multiplicação: " + (num1*num2));  
    }  
  
    public void multiplicar (int num1, int num2){  
        System.out.println("Multiplicação: " + (num1*num2));  
    }  
}
```

Figura 9

Fonte: Reprodução

As possibilidades são ilimitadas e a sobrecarga vai depender das necessidades de cada projeto de classe. Em Java, muitos métodos já são sobrecarregados. Por exemplo, o próprio comando de imprimir que sempre usamos “`System.out.println`” possui várias sobrecargas. Você pode passar como parâmetro um número inteiro, um número real ou mesmo uma *String* que ele consegue imprimir na tela o que passou. Isso quer dizer que existe um método `println` que recebe um `int`, outro que recebe um `double`, outro que recebe uma *String*, além de outros tipos.

Herança

Como o nome sugere, na orientação a objeto, se refere a algo que será herdado. Em Java ocorre quando uma classe herda todas as características, métodos e atributos de outra classe, essa técnica possibilita o compartilhamento e reaproveitamento de recursos definidos anteriormente. A classe principal que irá disponibilizar todos os recursos recebe o nome de superclasse e a classe que herda recebe o nome de subclasse.

Outro recurso da herança, permite que a classe que está herdando, ou subclasse possa aproveitar toda a característica da superclasse, como também implementar novos atributos e métodos.

Essa técnica de herança é muito utilizada em Java, para melhor entendimento sobre herança, observe os exemplos abaixo:

```
1 class Veiculo{  
2     private String nome;  
3     private float velocidade;  
4  
5     public void setNome(String nome){  
6         this.nome = nome;  
7     }  
8  
9     public String getNome(){  
10        return this.nome;  
11    }  
12  
13    public void setVelocidade(float velocidade){  
14        this.velocidade = velocidade;  
15    }  
16  
17    public float getVelocidade(){  
18        return velocidade;  
19    }  
20  
21    public void acelera(){  
22        if(velocidade<=10){  
23            velocidade++;  
24        }  
25    }  
26  
27    public void freia(){  
28        if(velocidade>0)  
29            velocidade--;  
30    }  
31}
```

Figura 10 – Classe veículo

Fonte: Reprodução

A classe Veículo possui duas variáveis de instância (nome e velocidade) e seis métodos. Existe a necessidade de outra classe utilizar as mesmas características da classe Veiculo, mas também implementará métodos que não foram desenvolvidos na superclasse, como, por exemplo, os métodos de ligar e desligar. Com a técnica da herança, as características da classe serão

herdadas na nova classe e, para realizar a herança, utiliza-se a palavra *extends*, ou seja, a subclasse *extends* superclasse, veja o exemplo abaixo:

```
1 class Veiculo1 extends Veiculo{  
2     private boolean ligado;  
3  
4     public void liga(){  
5         ligado = true;  
6     }  
7  
8     public void desliga(){  
9         ligado = false;  
10    }  
11}
```

Figura 11

Fonte: Reprodução

A classe **Veículo1** estende (*extends*) as funcionalidades da classe **Veículo**, ou seja, a classe **Veículo1** está herdando os métodos *set* e *get* e também os métodos **acelera()** e **freia()**. Veja a implementação do código:

```
1 import javax.swing.*;  
2 public class UsaVeiculo  
3 {  
4     public static void main(String args[]){  
5  
6         Veiculo1 v = new Veiculo1();  
7  
8         v.liga();  
9  
10        v.setNome(JOptionPane.showInputDialog("Digite o nome:"));  
11  
12        v.setVelocidade(Integer.parseInt(JOptionPane.showInputDialog("Digite a Velocidade:")));  
13  
14        JOptionPane.showMessageDialog(null,"Velocidade Atual: " + v.getVelocidade());  
15  
16        v.acelera();  
17  
18        JOptionPane.showMessageDialog(null,"Velocidade Atual: " + v.getVelocidade());  
19  
20        v.freia();  
21  
22        JOptionPane.showMessageDialog(null,"Velocidade Atual: " + v.getVelocidade());  
23  
24  
25        v.desliga();  
26    }
```

Figura 12

Fonte: Reprodução

Ao instanciar o **objeto v**, o mesmo receberá as variáveis de instância e todos os métodos presentes nas classes **Veículo** e **Veículo1**, desse modo o objeto v terá acesso às variáveis nome, velocidade, ligado e os métodos freia, acelera, liga e desliga.

Com base na modelagem UML abaixo, veremos como ficará o desenvolvimento das classes **MembroUniversidade**, **Aluno**, **Bolsista**, **Funcionário** e **Professor**.

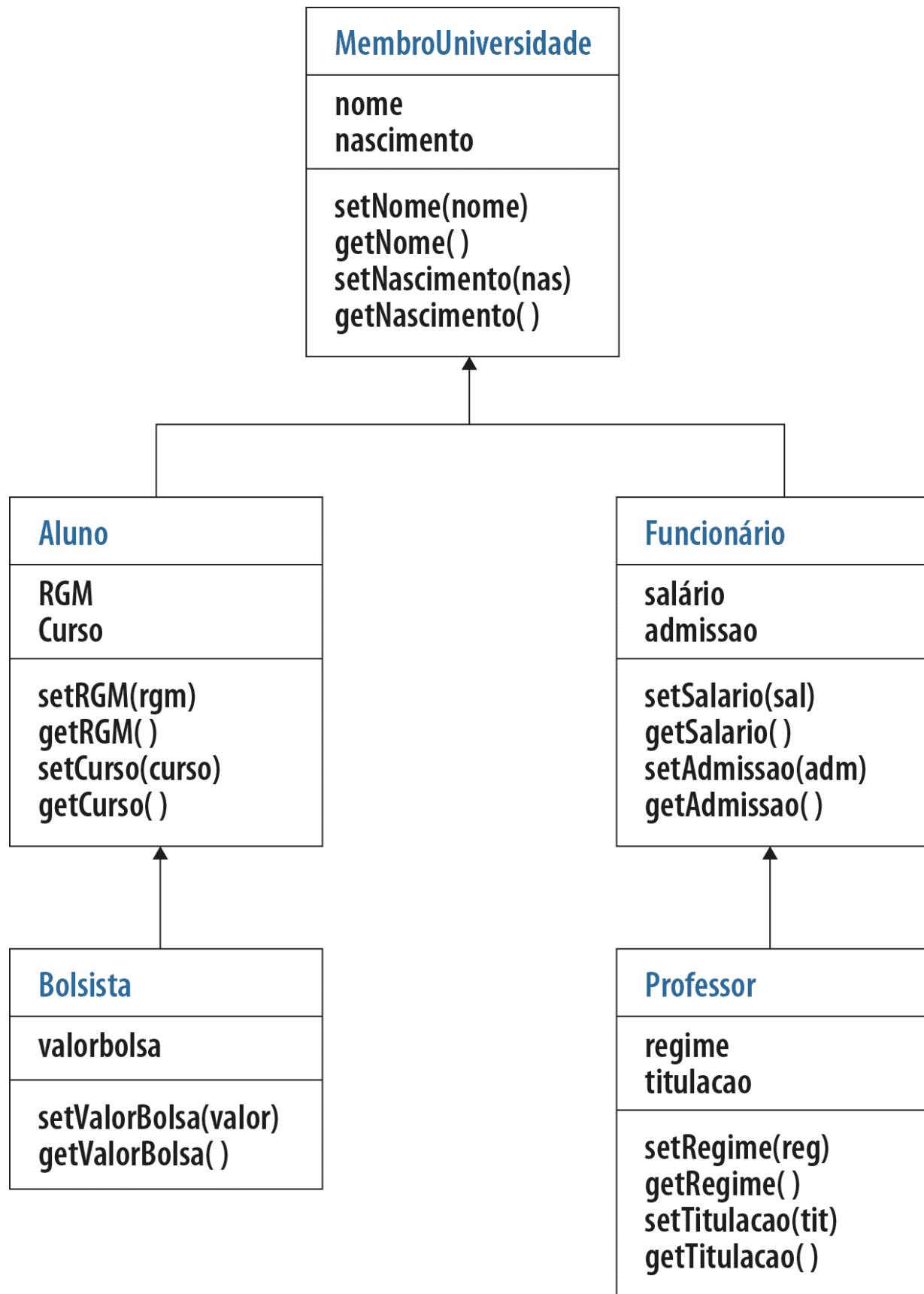


Figura 13

Logicamente, começaremos pela classe principal (superclasse).

```
public class MembroUniversidade {  
    private String nome;  
    private String nascimento;  
  
    public MembroUniversidade(String nnome, String nnascimento) {  
        nome = nnome;  
        nascimento = nnascimento;  
    }  
    public MembroUniversidade() {  
        nome = "";  
        nascimento = "";  
    }  
    public void setNome(String nnome) {  
        nome = nnome;  
    }  
    public String getNome() {  
        return nome;  
    }  
    public void setNascimento(String nnascimento) {  
        nascimento = nnascimento;  
    }  
    public String getNascimento() {  
        return nascimento;  
    }  
}
```

Figura 14

Fonte: Reprodução

Vamos, agora, implementar a classe Aluno que herda todas as características de **MembroUniversidade**. Observe a palavra *extends*, essa palavra indica que a classe Aluno herda as características da classe **MembroUniversidade**. Pode-se observar que a classe aluno tem seus próprios atributos como RGM e curso. A palavra chave “super” refere-se à superclasse da classe. O comando “super()” está chamando o construtor da classe pai, nesse exemplo a classe pai é a **MembroUniversidade**.

```
public class Aluno extends MembroUniversidade {  
    private String RGM;  
    private String curso;  
  
    public Aluno(String no, String na, String r, String cur) {  
        super(no, na);  
        RGM = r;  
        curso = cur;  
    }  
  
    public Aluno() {  
        super();  
        RGM = "";  
        curso = "";  
    }  
  
    public void setRGM(String r) { ... }  
  
    public String getRGM() { ... }  
  
    public void setCurso(String cur) { ... }  
  
    public String getCurso() { ... }  
}
```

Figura 15

Fonte: Reprodução

Veja, agora, como fica a classe Bolsista que será subclasse de Aluno.

```
public class Bolsista extends Aluno {  
    private float valorbolsa;  
  
    public Bolsista(String no, String na, String r, String cur, float v) {  
        super(no, na, r, cur);  
        valorbolsa = v;  
    }  
  
    public Bolsista() {  
        super();  
        valorbolsa = 0;  
    }  
  
    public void setValorBolsa(float v) { ... }  
  
    public float getValorBolsa() { ... }  
}
```

Figura 16

Fonte: Reprodução



Material Complementar

Indicações para saber mais sobre os assuntos abordados nesta Unidade:

Leitura

Lesson: Object-Oriented Programming Concepts

Clique no botão para conferir o conteúdo.

ACESSE

Livros

Aprenda Programação Orientada a Objetos em 21 Dias

SINTES, T. Aprenda Programação Orientada a Objetos em 21 dias. 1 ed. São Paulo: Pearson Education do Brasil, 2002, v. 1.

Java como Programar

DEITEL, P.; DEITEL, H. Java Como Programar. 8 ed. São Paulo: Pearson Education do Brasil, 2010.

Core Java

HORSTMANN, C. S.; CORNELL, G. **Core Java**. 8 ed. São Paulo: Pearson Education do Brasil, 2010, v. 1.

Referências

DEITEL, P.; DEITEL, H. **Java Como Programar**. 8 ed. São Paulo: Pearson Education do Brasil, 2010.

FURGERI, S. **Java 2 – Ensino Didático: Desenvolvendo e Implementando Aplicações**. São Paulo: Érica, 2002.

HORSTMANN, C. S.; CORNELL, G. **Core Java**. 8 ed. São Paulo: Pearson Education do Brasil, 2010, v. 1.

SINTES, T. **Aprenda Programação Orientada a Objetos em 21 dias**. 1 ed. São Paulo: Pearson Education do Brasil, 2002, v. 1.