Aula 3 - Fundamentos em Orientação a Objetos I

Docupedia Export

Author:Silio Leonardo (CtP/ETS)
Date:02-May-2024 14:27

Table of Contents

- Orientação a Objetos e Abstração
- Padrões de Nomenclatura
- Um exemplo OO usando Overload (Sobrecarga)
- Construtores
- Encapsulamento

1 Orientação a Objetos e Abstração

Agora que já compreendemos o básico de Java e como programar no mesmo usando programação estruturada, podemos finalmente aprender como usar a Orientação a Objetos na linguagem, bem como o que é este paradigma de programação.
A Orientação a Objetos é basicamente a separação das implementações e dados de um programa em estruturas chamadas Objetos. Um Objeto é como se fosse qualquer dado que seu computador salva durante a execução de uma aplicação, ou seja, um espacinho na memória com bits de dados. Porém, um objeto pode conter uma grande quantidade de dados, como se fosse um conjunto de informações específicas sobre aquela instância. Outra característica de um objeto é que além de um estado, que é como os dados são apresentados dentro dele, ele também tem várias funções que desempenham uma funcionalidade diferente a depender do estado deste objeto.
Um exemplo poderia ser um cadastro de clientes. Você tem vários clientes, cada um com nome, endereço, cpf, entre outras informações. Em um único bloco de dados você guarda todas essas informações. Perceba que você tem vários objetos que tem a mesma estrutura, porém com dados diferentes. Ainda assim, você tem funções que quando executadas, tem um comportamento diferente a depender do objeto para qual são chamadas. Por exemplo, a função atualizarEndereco vai mudar o endereço de um cliente específico, mas não de todos. Então o comportamento da função depende exclusivamente do objeto associado.
Isso é bem útil, porque sem esse tipo de ferramenta é difícil expressar a existência de vários objetos estruturalmente parecidos, mas com dados diferentes. Precisaríamos criar vários vetores com todos os dados de forma global (como era feito antigamente) para conseguir criar aplicações dessa natureza.
Para criar um objeto precisamos antes de um modelo, esse modelo especifica estruturalmente quais dados teremos bem como quais funções poderemos executar sobre esses objetos. A partir deste modelo iremos criar vários objetos. Este modelo se chama **Classe**. Abaixo, a forma de criar uma classe cliente como mencionada anteriormente em java:

```
1  public class Cliente
2  {
3      String nome;
4      String endereco;
5      String cpf;
6
7      void atualizarEndereco(String novoEndereco)
8      {
9          endereco = novoEndereco;
10     }
11 }
```

O código acima não é executável: Você não deve usar dentro de outras funções, dentro de um For ou If. Ele não é executado em momento algum, é apenas uma declaração de estrutura, mas não um código executado linha a linha. As únicas coisas que eventualmente são executadas são as funções colocadas dentro destas classes.
O nome das variáveis que você vê dentro da classe chamam-se **Atributos** em java; já a função, que não segue mais o exemplo de função da matemática já que seu comportamento varia a depender do estado do objeto, recebe o nome de **Método**.
A utilização do código acima é bem simples:

```
1  public class Main
```

```
2  {
3      public static void main(String[] args)
4      {
5          Cliente cliente1 = new Cliente();
6          cliente1.nome = "Gilmar";
7          cliente1.endereco = "Rua do Gilmar";
8          cliente1.cpf = "12345678-09";
9
10         Cliente cliente2 = new Cliente();
11         cliente2.nome = "Pamella";
12         cliente2.endereco = "Rua da Pamella";
13         cliente2.cpf = "87654321-90";
14
15         cliente2.atualizarEndereco("Avenida da Pamella");
16     }
17 }
```

Note que ao criarmos uma classe, estamos criando um tipo completamente novo. Assim, fazemos duas variáveis cliente1 e cliente2 e usamos a palavra reservada 'new' seguido do nome da classe e parênteses. Note que os dois objetos são diferentes e tem dados diferentes. Usamos a notação de ponto (variável.Atributo/Método) para acessar os campos/métodos dentro da classe. Ao executar o atualizarEndereco do cliente 2 apenas o endereço dele será atualizado.
A capacidade que a OO (Orientação a Objetos) nos da de representar um objeto real com suas características é um dos pilares da OO e chamamos ela de **Abstração**. A partir dessa característica construiremos aplicações bem modularizadas e poderosas.

2 Padrões de Nomenclatura

Agora que conhecemos a classe, vamos rapidamente entender o padrão de nomenclatura utilizada no java:

- O nome de classes dever ser escrito em **PascalCase**.
- Todo resto deve usar **camelCase**.

Verá que existem momentos em que podemos desrespeitar levemente essas regras, mas em geral, as siga.

3 Um exemplo OO usando Overload (Sobrecarga)

Overload é uma das capacidades da OO em que dentro de classes podemos ter várias funções com mesmo nome, desde que tenham parâmetros diferentes. Abaixo segue um exemplo legal onde podemos usar a OO para representar uma classe para horários e, adicionalmente, criamos um método Adicionar, que avança no tempo para podermos alterar o horário armazenado.
Importante: Não basta o nome dos parâmetros ser diferente, mas sim a quantidade ou os tipos devem diferir.

```java
// Horario.java
public class Horario
{
    int hora = 0;
    int minuto = 0;
    int segundo = 0;

    public void adicionar(int horas, int minutos, int segundos)
    {
        this.segundo += segundos;
        if (this.segundo > 59)
        {
            minutos += segundo / 60;
            this.segundo %= 60;
        }

        this.minuto += minutos;
        if (this.minuto > 59)
        {
            horas += minuto / 60;
            this.minuto %= 60;
        }

        this.hora += horas;
        if (this.hora > 23)
        {
            this.hora %= 24;
        }
    }

    public void adicionar(int minutos, int segundos)
    {
```

```
33          adicionar(0, minutos, segundos);
34      }
35
36      public void adicionar(int segundos)
37      {
38          adicionar(0, 0, segundos);
39      }
40
41      public void adicionar(Horario horario)
42      {
43          adicionar(horario.hora, horario.minuto, horario.segundo);
44      }
45
46      public String Formatar()
47      {
48          return String.format("%s:%s:%s",
49              hora, minuto, segundo
50          );
51      }
52  }
53
54  // Main.java
55  public class Main
56  {
57      public static void main(String[] args)
58      {
59          Horario h1 = new Horario();
60          Horario h2 = new Horario();
61
62          h1.adicionar(20, 40, 30); //h1 = 20:40:30
63          h2.adicionar(20, 30); //h2 = 00:20:30
64
65          h1.adicionar(h2); //h1 = 21:01:00, h2 inalterado
66          h2.adicionar(h1); //h2 = 00:20:30 + 21:01:00 = 21:21:30, h1 inalterado
67
68          System.out.println(h2.Formatar()); //21:21:30
69      }
70  }
```

4 Construtores

Além disso, podemos fazer construtores, que são funções chamadas no momento que os objetos são criados a partir das classes. É possível ainda usar a sobrecarga de métodos para ter vários construtores diferentes. Esses construtores são usados, em geral, para inicializar os objetos e não costuma ter código muito pesado dentro deles. Observe o exemplo anterior, agora com construtores:

```
1   public class Horario
2   {
3       int hora = 0;
4       int minuto = 0;
5       int segundo = 0;
6
7       public Horario() // Construtores não tem retorno e o nome é o nome da classe
8       {
9           // Vazio não altera os dados
10      }
11
12      public Horario(int horas, int minutos, int segundos)
13      {
14          // Geralmente inicializamos 1 a 1, mas aqui preferimos usar a função Adicionar que já está pronta
15          // hora = horas;
16          // minuto = minutos;
17          // segundo = segundos;
18          adicionar(horas, minutos, segundos);
19      }
20
21      public void adicionar(int horas, int minutos, int segundos)
22      {
23          this.segundo += segundos;
24          if (this.segundo > 59)
25          {
26              minutos += segundo / 60;
27              this.segundo %= 60;
28          }
29
30          this.minuto += minutos;
31          if (this.minuto > 59)
32          {
33              horas += minuto / 60;
```

```
34                  this.minuto %= 60;
35              }
36
37              this.hora += horas;
38              if (this.hora > 23)
39              {
40                  this.hora %= 24;
41              }
42          }
43
44          public void adicionar(int minutos, int segundos)
45          {
46              adicionar(0, minutos, segundos);
47          }
48
49          public void adicionar(int segundos)
50          {
51              adicionar(0, 0, segundos);
52          }
53
54          public void adicionar(Horario horario)
55          {
56              adicionar(horario.hora, horario.minuto, horario.segundo);
57          }
58
59          public String Formatar()
60          {
61              return String.format("%s:%s:%s",
62                  hora, minuto, segundo
63              );
64          }
65      }
```

5 Encapsulamento

Observe o exemplo do horário e responda: Não é perigoso que um programador desavisado tente executar um código como o abaixo?

```
horario.segundo += 1;
```

O código simples somaria um segundo no campo Segundo. O grande problema é que isso poderia fazer com que o valor de segundos chegasse a mais de 60 sem adicionar um valor no minuto. O programador não sabe ou esquece que o horário deve manter-se consistente e que deve utilizar o método Adicionar para fazer isso. E qual é o problema? Isso pode acarretar em Bugs. Este caso é simples, mas o mesmo problema pode escalar de formas astronômicas.
Outra situação: Imagine que você cria um sistema que utiliza uma classe da seguinte forma:

```
class Cliente
{
    String login;
    String senha;
}
```

Depois de anos seu sistema é comprado e agora exige-se que a senha seja criptografada por questões de segurança. Isso faz com que você faça algo do tipo:

```
class Cliente
{
    String login;
    private String senha;

    void definirSenha(String value)
    {
        String senhaCriptografada = "";
        // Criptografa o value e guarda na variável senhaCriptografada
        senha = senhaCriptografada;
    }
}
```

Ainda assim, você esquece de trocar algumas partes do software, as atribuições de senha pela função DefinirSenha e pior ainda, desavisados, os seus colegas acabam realizando implementações esquecendo-se de usar o DefinirSenha. Isso acarreta que agora várias senhas salvas estão sem criptografia e muito pior que isso: Você não sabe exatamente quais estão criptografadas e quais são só estranhas.
Isso pode causar muita dor de cabeça, e gerar muitos bugs. Graças a isso, a OO nos trás o pilar do **Encapsulamento** - o segundo pilar, depois da abstração, de um total de 4 pilares da OO. Para aplicá-lo usaremos a palavra reservada 'private' que esconde as estruturas de um código:

```
Cliente c = new Cliente();
c.senha = "Xispita";        //Erro pois senha é agora privada
```

```
c.definirSenha("Xispita");  // Ok
System.out.println(c.ObterSenha());

class Cliente
{
    String login;
    private String senha;

    void definirSenha(String value)
    {
        String senhaCriptografada = "";
        // Criptografa o value e guarda na variável senhaCriptografada
        senha = senhaCriptografada;
    }

    string obterSenha()
    {
        return senha;
    }
}
```

Na verdade, nenhum campo deve ser público. Devemos usar essas funções de Definir e Obter, quando possível, para acessar os valores. Usamos o inglês Get e Set para as mesmas:

```
Cliente c = new Cliente();
c.SetLogin("Gilmar");
c.SetSenha("Xispita");

class Cliente
{
    private String login;
    private String senha;

    void setSenha(String value)
    {
        string senhaCriptografada = "";
        // Criptografa o value e guarda na variável senhaCriptografada
        senha = senhaCriptografada;
    }
```

```
16
17      String getSenha()
18      {
19          return senha;
20      }
21
22      void setLogin(String value)
23      {
24          login = value;
25      }
26
27      String getLogin()
28      {
29          return login;
30      }
31  }
```

Note que as funções GetLogin e SetLogin parecem inúteis, diferente da senha. Mas é justamente isso que é interessante. Em qualquer momento que precisemos alterar a forma de como o código conversa com os dados de login, basta alterar esses métodos. Ou seja, não precisamos reestruturar todo o código para que as coisas funcionem.
Voltando ao nosso exemplo de Classe Horario, poderíamos reestruturá-la. Note que o Set pode ter uma implementação peculiar:

```
1   public class Horario
2   {
3       private int hora = 0;
4       private int minuto = 0;
5       private int segundo = 0;
6
7       public Horario() // Construtores não tem retorno e o nome é o nome da classe
8       {
9           // Vazio não altera os dados
10      }
11
12      public Horario(int horas, int minutos, int segundos)
13      {
14          // Geralmente inicializamos 1 a 1, mas aqui preferimos usar a função Adicionar que já está pronta
15          // hora = horas;
16          // minuto = minutos;
17          // segundo = segundos;
```

```
18          adicionar(horas, minutos, segundos);
19      }
20
21      int getHora() {
22          return hora;
23      }
24
25      int getMinuto() {
26          return minuto;
27      }
28
29      int getSegundo() {
30          return segundo;
31      }
32
33      void setHora(int value) {
34          hora = 0;
35          adicionar(value, 0, 0);
36      }
37
38      void setMinuto(int value)
39      {
40          minuto = 0;
41          adicionar(value, 0);
42      }
43
44      void setSegundo(int value)
45      {
46          segundo = 0;
47          adicionar(value);
48      }
49
50      public void adicionar(int horas, int minutos, int segundos)
51      {
52          this.segundo += segundos;
53          if (this.segundo > 59)
54          {
55              minutos += segundo / 60;
56              this.segundo %= 60;
```

```
57          }
58
59          this.minuto += minutos;
60          if (this.minuto > 59)
61          {
62              horas += minuto / 60;
63              this.minuto %= 60;
64          }
65
66          this.hora += horas;
67          if (this.hora > 23)
68          {
69              this.hora %= 24;
70          }
71      }
72
73      public void adicionar(int minutos, int segundos)
74      {
75          adicionar(0, minutos, segundos);
76      }
77
78      public void adicionar(int segundos)
79      {
80          adicionar(0, 0, segundos);
81      }
82
83      public void adicionar(Horario horario)
84      {
85          adicionar(horario.hora, horario.minuto, horario.segundo);
86      }
87
88      public String Formatar()
89      {
90          return String.format("%s:%s:%s",
91              hora, minuto, segundo
92          );
93      }
94  }
```