Aula 7 - Fundamentos em Orientação a Objetos II

Docupedia Export

Author:Silio Leonardo (SO/OPM-TS21-BR)
Date:08-May-2024 14:16

Table of Contents

- Herança
- Métodos virtuais e sobrescrita
- Classes abstratas
- Polimorfismo
- Object

1 Herança

Na OO temos 4 pilares principais. Quatro conceitos primordiais que conduzem bastante o comportamento e a forma de raciocínio usado em uma aplicação OO. Já falamos sobre a Abstração e o Encapsulamento, agora iremos falar sobre o que é a Herança em um código OO.
A Herança é a capacidade de um objeto de herdar todas as características de outro. Para isso, basta que sua classe indique de qual outra classe essas características devem ser herdadas. Observe um rápido exemplo:

```
1   public class Main
2   {
3       public static void main(String[] args)
4       {
5           A a = new A();
6           B b = new B();
7           System.out.println(a.getValue());
8           // System.out.println(a.getOtherValue()); Erro
9
10          System.out.println(b.getValue());
11          System.out.println(b.getOtherValue());
12      }
13  }
14
15  // A.java
16  class A
17  {
18      private int value = 2;
19      public int getValue() {
20          return this.value;
21      }
22      public void setValue(int value) {
23          this.value = value;
24      }
25  }
26
27  // B.java
28  class B extends A
29  {
30      private int otherValue = 3;
31      public int getOtherValue() {
32          return this.otherValue;
```

```
    }
    public void setOtherValue(int value) {
        this.otherValue = value;
    }
}
```

B não tinha a propriedade Value, mas ao digitarmos 'B : A' estamos dizendo que tudo que A tem, B também tem. Assim, B agora tem Value e tudo mais que A possa implementar, inclusive métodos. Note que isso não faz com que A seja alterada de maneira alguma.
Podemos utilizar a Herança de muitas formas, mas em geral devemos fazer a pergunta "é um?". Ou seja, se o objeto B é um A, isso significa que B tem tudo que A tem e assim a herança é valida. Mas isso não é 100% perfeito que torna a Herança alvo de várias críticas. Vamos observar 3 exemplos de herança e avaliar isso por nós mesmos. A Herança é a capacidade de um objeto herdar todas as características de outro. Para isso, basta que sua classe indique de qual outra classe essas características devem ser herdadas. Observe um rápido exemplo:

```
public class Main
{
    public static void main(String[] args)
    {
        Gato gato = new Gato("Edjalma");
        Cao cao = new Cao("Stati");

        gato.Miar();
        cao.Latir();
    }
}

// Pet.java
class Pet
{
    Pet(String nome) {
        this.nome = nome;
    }

    private String nome;
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
```

```
}

// Cao.java
class Cao extends Pet
{
    // A classe Pet precisa de um nome para ser construído. Assim todas as suas classes bases precisam
    // de um construtor equivalente. Caso contrário obtemos um erro. Você pode ainda usar 'super(nome)'
    // para chamar a funcionalidade do construtor da classe mãe (Pet).
    public Cao(String nome) {
        super(nome);
    }

    public void Latir() {
        System.out.println("Au!");
    }
}

// Gato.java
class Gato extends Pet
{
    public Gato(String nome) {
        super(nome);
    }

    public void Miar() {
        System.out.println("Miau!");
    }
}
```

E mais um exemplo:

```
public class Main
{
    public static void main(String[] args)
    {
        RelogioDeXadrez relogio = new RelogioDeXadrez();
        relogio.setAcrescimo(2);
        relogio.Iniciar(0, 5, 0);

```

```
        while (true)
        {
            System.out.println(relogio.getMinuto() + ":" + relogio.getSegundo());
            relogio.Tick();
        }
    }
}

// Relogio.java
class Relogio
{
    private int segundo = 0;
    public int getSegundo() {
        return segundo;
    }
    public void setSegundo(int segundo) {
        this.segundo = segundo;
    }

    private int minuto = 0;
    public int getMinuto() {
        return minuto;
    }
    public void setMinuto(int minuto) {
        this.minuto = minuto;
    }

    private int hora = 0;
    public int getHora() {
        return hora;
    }
    public void setHora(int hora) {
        this.hora = hora;
    }

    void adicionar(int segundos, int minutos, int horas)
    {
        this.segundo += segundos;
        if (this.segundo > 59)
```

```
48          {
49              minutos += this.segundo / 60;
50              this.segundo = this.segundo % 60;
51          }
52
53          this.minuto += minutos;
54          if (this.minuto > 59)
55          {
56              horas += this.minuto / 60;
57              this.minuto = this.minuto % 60;
58          }
59
60          this.hora += horas;
61          if (this.hora > 23)
62          {
63              this.hora = this.hora % 24;
64          }
65      }
66
67      void remover(int segundos, int minutos, int horas)
68      {
69          this.segundo -= segundos;
70          if (this.segundo < 0)
71          {
72              this.minuto -= this.segundo / 60 + 1;
73              this.segundo += 60 * (this.segundo / 60 + 1);
74          }
75
76          this.minuto -= minutos;
77          if (this.minuto < 0)
78          {
79              this.hora -= this.minuto / 60 + 1;
80              this.minuto += 60 * (this.minuto / 60 + 1);
81          }
82
83          this.hora -= horas;
84          if (this.hora < 0)
85          {
86              this.hora += 24 * (this.hora / 24 + 1);
```

```
87              }
88          }
89
90          // Passa 1 segundo
91          void Tick()
92          {
93              adicionar(1, 0, 0);
94          }
95      }
96
97      // Timer.java
98      class Timer extends Relogio
99      {
100         // Tem o mesmo nome da função Tick da classe base, assim ela
101         // 'esconde' a existência da antiga função
102         void Tick()
103         {
104             remover(1, 0, 0);
105             if (getHora() == 0 && getMinuto() == 0 && getSegundo() == 0)
106                 Apitar();
107         }
108
109         public void Zerar()
110         {
111             remover(getSegundo(), getMinuto(), getHora());
112         }
113
114         public void Iniciar(int hora, int minuto, int segundo)
115         {
116             Zerar();
117             adicionar(segundo, minuto, hora);
118         }
119
120         protected void Apitar()
121         {
122             System.out.println("O tempo acabaou");
123         }
124     }
125
```

```java
// RelogioDeXadrez.java
class RelogioDeXadrez extends Timer {

    private int acrescimo;
    public int getAcrescimo() {
        return acrescimo;
    }
    public void setAcrescimo(int acrescimo) {
        this.acrescimo = acrescimo;
    }

    public void JogadaFeita()
    {
        adicionar(acrescimo, 0, 0);
    }
}
```

2 Métodos virtuais e sobrescrita

Melhor que esconder é sobrescrever. Os comportamentos dos métodos Java podem ser reescritos para ter um novo comportamento na classe filha.

```
1  // Printer.java
2  class Printer
3  {
4      protected void printSuperior() {
5          System.out.printl("A seguir uma mensagem:");
6      }
7
8      protected void printInferior() {}
9
10     // Não pode ser reescrito
11     final void print(string message)
12     {
13         printSuperior();
14         System.out.println(message);
15         printInferior()
16     }
17 }
18
19 // BeautyPrinter.java
20 class BeautyPrinter extends Printer
21 {
22     @Override
23     protected void printSuperior()
24     {
25         System.out.println("--------------------");
26     }
27
28     @Override
29     protected void printInferior()
30     {
31         System.out.println("--------------------");
32     }
33 }
```

3 Classes abstratas

Além disso, você pode criar classes abstratas - classes que não podem ser instanciadas. Isso é perfeito para situações onde a classe mãe não existe na prática. Você ainda pode colocar implementações abstratas na classe. Isso significa que você pode adicionar um método abstrato que você não implementa, apenas declara. Todas as classes base são obrigadas a implementar aquela função. Isso é extremamente útil em muitos cenários. Observe um interessante exemplo:

```
1   abstract class Language
2   {
3       abstract String TranslateNewGame();
4       abstract String TranslateQuit();
5       abstract String TranslateLoadGame();
6       abstract String TranslateOptions();
7   }
8
9   class English extends Language
10  {
11      @Override
12      String TranslateNewGame() {
13          return "New Game";
14      }
15      @Override
16      String TranslateQuit() {
17          return "Quit";
18      }
19      @Override
20      String TranslateLoadGame() {
21          return "Load Game";
22      }
23      @Override
24      String TranslateOptions() {
25          return "Options";
26      }
27  }
28
29  class Portuguese extends Language
30  {
31      @Override
32      String TranslateNewGame() {
33          return "Novo Jogo";
```

```
34      }
35      @Override
36      String TranslateQuit() {
37          return "Sair";
38      }
39      @Override
40      String TranslateLoadGame() {
41          return "Carregar Jogo";
42      }
43      @Override
44      String TranslateOptions() {
45          return "Opções";
46      }
47  }
```

4 Polimorfismo

Além disso, existe outro recurso poderosíssimo na Orientação a Objetos que é o nosso quarto pilar: O **Polimorfismo**. O Polimorfismo é a capacidade de uma referência/variável apresentar diferentes comportamentos a depender do objeto nele contido, apesar de seu tipo ser um único. Isso será possível pois existe um conceito chamado **Variância**. A Variância permite que coloquemos objetos da classe filha em uma variável da classe mãe. Considerando o exemplo de tradução visto acima observe que isso seria possível:

```
Language lang = null;

System.out.println("Select your language:");
System.out.println("1 - English")
System.out.println("2 - Portugues")
String selected = scanner.next();

if (selected == "1")
{
    lang = new English();
}
else if (selected == "2")
{
    lang = new Portuguese();
}
else
{
    System.out.println("Error: Unknown Input.");
    return;
}

WriteLine($"1 - {lang.TranslateNewGame()}");
WriteLine($"2 - {lang.TranslateLoadGame()}");
WriteLine($"3 - {lang.TranslateOptions()}");
WriteLine($"4 - {lang.TranslateQuit()}");
```

Embora seja um variável do tipo Language que nem mesmo pode ser instanciado, lang tem comportamentos distintos a depender do objeto que ele recebe. Isso é o fenômeno chamado de **Polimorfismo**. Muito melhor do que criar uma variável que armazena o id da linguagem (1, 2, etc.) e realizar um 'if' toda vez que quiser escrever algo.

5 Object

Isso tudo os leva ao Object, um tipo fundamental no java. Todos os objetos por referência java herdam de Object, e consequentemente tem tudo que nele tem e podem ser colocados em variáveis do tipo:

```
1  Cliente client = new Cliente();
2  Object obj = client;
3
4  client.Nome = "Gilmar";
5  obj.Nome = "Pamella"; // Erro: Object não contém uma definição de 'Nome'
6
7  Cliente x = (Cliente)obj; // Se obj não tive um objeto do tipo cliente estourará um erro
8  x.Saldo = 100000;
```