UNIP
UNIVERSIDADE PAULISTA

UNICLUSTER

Pilares da LPOO

Carlos Arruda Baltazar
UNIP – Cidade Universitária

Pilares

Para entendermos melhor do que se trata a orientação a objetos, faz-se necessário entender quais são os requerimentos de uma linguagem para ser enquadrada nesse paradigma. Para isso, a linguagem precisa atender a quatro tópicos bastante importantes:

Abstração

A abstração consiste em um dos pontos mais importantes dentro de qualquer linguagem orientada a objetos. Como estamos lidando com uma representação de um objeto real, precisa-se imaginar o que esse objeto irá realizar dentro do programa. São três pontos que devem ser levados em consideração nessa abstração:

Abstração

- **Identidade**: Dar uma identidade única ao objeto é um ponto de extrema importância para que não haja conflito dentro do programa. Na maior parte das linguagens, há o conceito de pacote ou *namespace*, então a identidade de um objeto não pode ser repetida dentro de um pacote. Contudo nada impede que haja objetos com a mesma identidade em pacotes diferentes.

Abstração

- **Propriedades/Atributos**: Em geral na literatura, as propriedades, também conhecidas como atributos dizem respeito as características do objeto. No mundo real, qualquer objeto possui elementos que o caracterizam, na linguagem de programação, essas características são mapeadas pelos atributos.

Abstração

- **Métodos**: Além de elementos que o definem, um objeto também pode executar “ações”. Essas ações ou eventos, na linguagem de programação, são modelados pelos métodos.

Encapsulamento

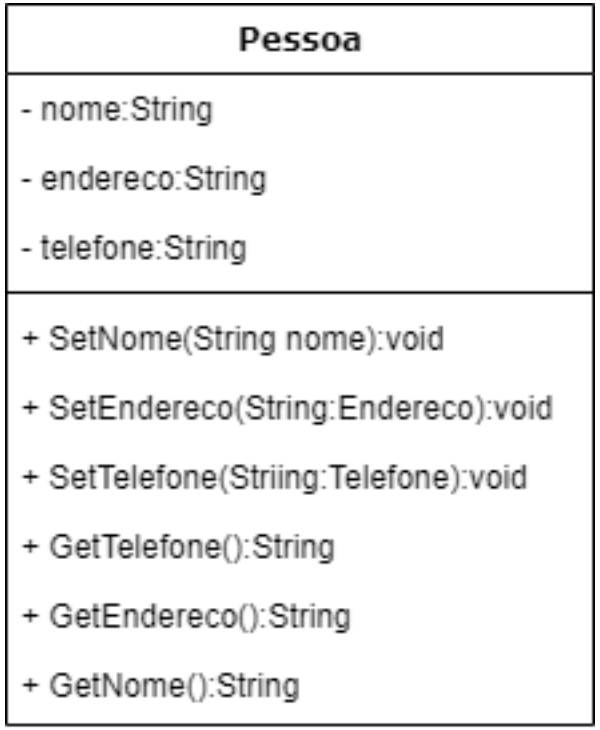
A técnica de encapsulamento empregada na programação orientada a objetos se trata de um elemento que adiciona segurança ao programa. Essa segurança se dá devido a possibilidade de esconder os atributos de um objeto, cirando assim, uma espécie de caixa preta. A programação orientada a objetos geralmente implementa um encapsulamento baseado em atributos privados, ou seja, os atributos de um objeto apenas podem ser acessados, escritos ou alterados utilizando métodos especiais:

Encapsulamento

- ***Setters***: métodos responsáveis por inserir e alterar valores contidos nos atributos.

- ***Getters***: métodos responsáveis por retornar o valor de um dado atributo.

Encapsulamento



Encapsulamento

Essa técnica evita o acesso direto aos atributos de um método, adicionando assim uma cama de segurança ao programa.

Porém, faz necessário ressaltar também que o encapsulamento não se limita aos atributos de um objeto, mas também a trechos de código que desempenham uma função atômica. Esses trechos de códigos devem ser encapsulados em métodos de um objeto.

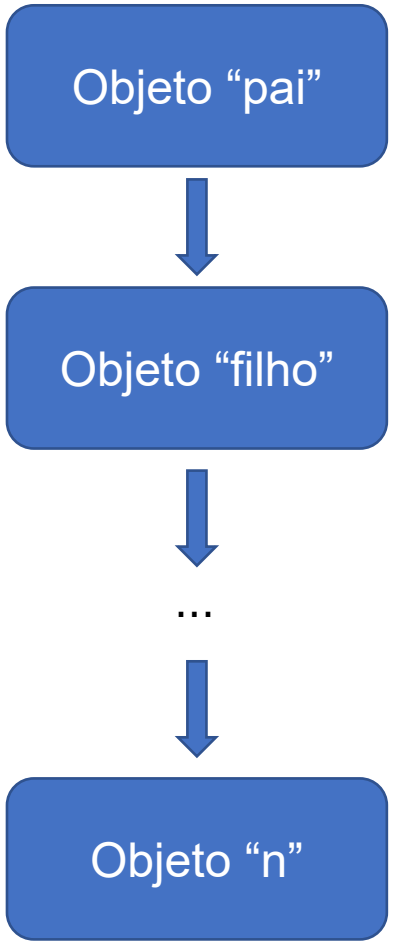
Herança

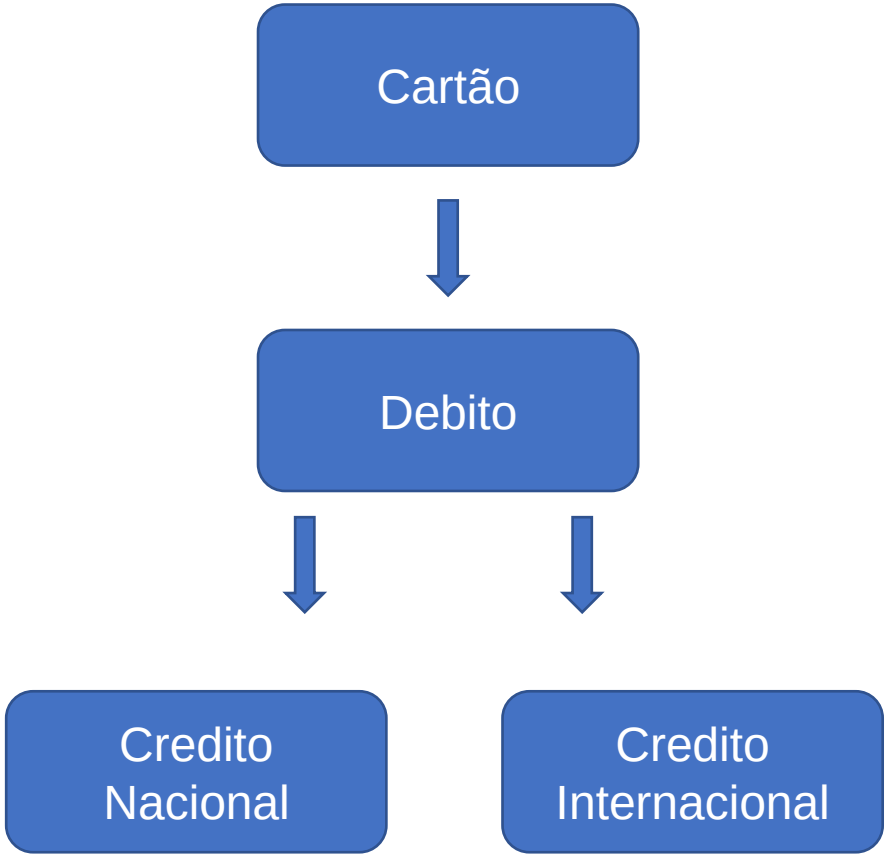
Uma das grandes vantagens da programação orientada a objetos é a reutilização de código dentro de um programa. O conceito de “herança” é um dos pilares que garante o reuso de código otimizando a produção da aplicação em tempo de economizando linhas de código.

Herança

Para se entender esse conceito é necessário compreender a herança pressupõe uma hierarquia entre objetos. Essa afirmação quer dizer que: um objeto “filho”, que por sua vez está mais abaixo na hierarquia, irá “herdar” os atributos e métodos de um objeto “pai”. Esse cenário representa uma herança direta, onde um objeto herda aspectos de um objeto logo acima de acordo com a hierarquia, os outros demais cenários tratam-se de uma herança indireta.

Herança


Objeto “pai”
Objeto “filho”
...
Objeto “n”


Cartão
Debito
Credito Nacional
Credito Internacional

Herança - Exemplo

Por exemplo, Imagine que dentro de uma organização empresarial, o sistema de RH tenha que trabalhar com os diferentes níveis hierárquicos da empresa, desde o funcionário de baixo escalão até o seu presidente. Todos são funcionários da empresa, porém cada um com um cargo diferente. Mesmo a secretária, o pessoal da limpeza, o diretor e o presidente possuem um número de identificação, além de salário e outras características em comum.

Herança - Exemplo

Essas características em comum podem ser reunidas em um tipo de classe em comum, e cada nível da hierarquia ser tratado como um novo tipo, mas aproveitando-se dos tipos já criados, através da herança. **Os subtipos, além de herdarem todas as características de seus supertipos, também podem adicionar mais características, seja na forma de variáveis e/ou métodos adicionais**

Herança - Exemplo

A herança permite vários níveis na hierarquia de classes, podendo criar tantos subtipos quanto necessário, até se chegar ao nível de especialização desejado. Podemos tratar subtipos como se fossem seus supertipos, por exemplo, o sistema de RH pode tratar uma instância de Presidente como se fosse um objeto do tipo Funcionário, em determinada funcionalidade.

Herança - Exemplo

O que um aluno, um professor e um funcionário possuem em comum? Todos eles são pessoas e, portanto, compartilham alguns dados comuns. Todos têm nome, idade, endereço, etc. E, o que diferencia um aluno de outra pessoa qualquer? Um aluno possui uma matrícula; Um funcionário possui um código de funcionário, data de admissão, salário, etc.; Um professor possui um código de professor e informações relacionadas à sua formação. .

Herança - Exemplo

O que um aluno, um professor e um funcionário possuem em comum? Todos eles são pessoas e, portanto, compartilham alguns dados comuns. Todos têm nome, idade, endereço, etc. E, o que diferencia um aluno de outra pessoa qualquer? Um aluno possui uma matrícula; Um funcionário possui um código de funcionário, data de admissão, salário, etc.; Um professor possui um código de professor e informações relacionadas à sua formação. .

Herança - Exemplo

É aqui que a herança se torna uma ferramenta de grande utilidade. 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, ou seja, a herança parte do geral para o mais específico. Comece criando uma classe Pessoa, Pessoa.java, como mostrado no código a seguir:

Herança - Exemplo


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

Esta classe possui os atributos nome e idade. Estes atributos são comuns a todas as pessoas. Veja agora como podemos criar uma classe Aluno que herda estes atributos da classe Pessoa e inclui seu próprio atributo, a saber, seu número de matrícula. Eis o código:

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

Herança - Exemplo

Observe que, em Java, a palavra-chave usada para indicar herança é extends. A classe Aluno agora possui três atributos: nome, idade e matricula. Exemplo:

```java
public class Estudos{
  public static void main(String args[]){
    Aluno aluno = new Aluno();
    aluno.nome = "Aluno Esforçado";
    aluno.idade = 20;
    aluno.matricula = "XXXX99999";

    System.out.println("Nome: " + aluno.nome + "\n" +
    "Idade: " + aluno.idade + "\n" +
    "Matrícula: " + aluno.matricula);
  }
}
```

Herança - Exemplo

Observe que, em Java, a palavra-chave usada para indicar herança é extends. A classe Aluno agora possui três atributos: nome, idade e matricula. 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 - Exemplo

A herança nos fornece um grande benefício. Ao concentrarmos características comuns em uma classe e derivar as classes mais específicas a partir desta, nós estamos preparados para a adição de novas funcionalidades ao sistema. Se mais adiante uma nova propriedade comum tiver que ser adicionada, não precisaremos efetuar alterações em todas as classes. Basta alterar a superclasse e pronto. As classes derivadas serão automaticamente atualizadas.

Polimorfismo

Outro importante pilar da programação orientada a objetos é o polimorfismo. Em geral, trata-se do princípio pelo qual duas ou mais classes derivadas de uma mesma classe base podem invocar métodos que tem a mesma assinatura, porém possuem comportamentos distintos. Portanto, polimorfismo é um mecanismo por meio do qual é possível selecionar funcionalidades utilizadas forma dinâmica por um programa ao decorrer de sua execução.

OBRIGADO