

Programação Orientada a Objeto

OO4 - Encapsulamento, Construtores e Herança

Dr. Márcia L. Aguená Castro
marciaaguená@gmail.com

OO4 - Encapsulamento, Construtores e Herança

Encapsulamento

- A definição de **encapsulamento** é “ocultar ou proteger detalhes de implementação”.
 - Para que serve o vidro de um relógio de ponteiros? E o seus pinos de regulagem?
 - O que aconteceria se pudéssemos mexer livremente nos ponteiros para ajustar a hora?
 - O vidro e a caixa do relógio encapsulam seu mecanismo.
 - O relógio oferece a interface do visor e os pinos de regulagem.

Encapsulamento

- **Encapsulamento** nas **linguagens procedurais**: procedimentos e funções ocultam detalhes de implementação.
- As principais vantagens são **legibilidade**, **manutenibilidade** e **reutilização** de código
- Em Java, o encapsulamento é implementado por meio do uso da palavra-chave `private` em todos os atributos de uma classe (ou seja, o encapsulamento é garantido **sintaticamente**).

OO4 - Encapsulamento, Construtores e Herança

Encapsulamento

- Altere a classe Pessoa incluindo a palavra-chave private:

```
1  public class Pessoa{  
2      private String nome;  
3      public String getNome() {  
4          return this.nome;  
5      }  
6  }
```

- Agora compile as classes Pessoa e TestaPessoa.
- O que acontece? Como resolver?

OO4 - Encapsulamento, Construtores e Herança

Encapsulamento

- Um atributo ou método declarado como `private` não faz parte da interface pública dessa classe **e não é possível acessá-lo** por meio do operador ponto como era feito anteriormente no exemplo.

`p1.nome = "Reinaldo";`

- Voltando ao exemplo do relógio de ponteiros (classe Pessoa), é como se os **ponteiros (atributos)**, que antes eram acessíveis ao usuário, tivessem sido colocados dentro de um **vidro (métodos)** e agora só pode ser visto através dele.

OO4 - Encapsulamento, Construtores e Herança

Encapsulamento

```
1 public class Pessoa{
2     private String nome;
3     public String getNome() {
4         return this.nome;
5     }
6     public void setNome(String $nome) {
7         this.nome = $n;
8     }
9 }
```

Declaração do método
setNome() na classe
Pessoa

```
1 public class TestaPessoa{
2     public static void main (String[] args) {
3         Pessoa p1 = new Pessoa();
4         p1.setNome("Reinaldo");
5         System.out.println("O nome da pessoa é " + p1.getNome());
6     }
7 }
```

Utilização do método
setNome() na classe

OO4 - Encapsulamento, Construtores e Herança

Encapsulamento

Atenção: Não copie ou cole nenhum exercício. A repetição é intencional para criar fluência na linguagem.

1. Dentro do diretório `sistema`, crie a seguinte classe:

- `ContaPoupanca`.

2. Crie os seguintes atributos para a classe `ContaPoupança`:

- Todos os atributos da classe conta (`"id"`, `"numero"`, `"saldo"`);
- `"diaDeAniversario"` do tipo `short`;
- `"taxaDeCorrecao"` to tipo `double`.

OO4 - Encapsulamento, Construtores e Herança

Encapsulamento

1. Para as classes Banco, Agencia, Conta, ContaCorrente e ContaPoupança deixe todos os atributos privados;
2. Para as classes Banco, Agencia, Conta, ContaCorrente e ContaPoupança crie os métodos get's e set's para todos atributos;
3. Na classe AplicacaoFinanceira:
 1. Substitua todas os acessos diretos a atributos dos objetos banco1, agencia1, conta1 e contaCorrente1 por métodos set's e get's;

3. Na classe `AplicacaoFinanceira` (continuação):
 2. Crie um objeto `contaPoupanca1` da classe `ContaPoupanca`;
 3. Utilizando métodos `set`'s entre com valores para os atributos do objeto `contaPoupanca1`;
 4. Utilizando métodos `get`'s mostre os valores dos atributos de `contaPoupanca1`.
4. Compile a classe `AplicacaoFinanceira` e corrija os possíveis erros.

OO4 - Encapsulamento, Construtores e Herança

Construtores

- Construtores são **métodos especiais**, que têm a responsabilidade de passar valores aos atributos de um objeto no momento da instanciação;
- É sempre utilizado em conjunto com um operador new. Esse operador tem a responsabilidade de :
 - **Alocar memória** suficiente para o objeto sendo construído,
 - **Inicializar os atributos** com os valores padrões (ex: objeto com **null** e inteiros com **0**) e ;
 - Chamar o **construtor** a sua direita.

OO4 - Encapsulamento, Construtores e Herança

Construtores

- A declaração de um construtor em Java é, geralmente:
 - **público,**
 - **não tem** tipo de retorno e
 - sempre usa **EXATAMENTE o mesmo nome da classe.**
- Caso você não declare um construtor, **o compilador Java cria um construtor vazio padrão** automaticamente para a classe em questão.
- Porém, se você implementar qualquer construtor **não for o vazio**, e **chamar um construtor vazio**, precisa **implementar o construtor vazio.**

OO4 - Encapsulamento, Construtores e Herança

Construtores

- Classe `Pessoa` com dois construtores adicionados, sendo um vazio:

```
1  public class Pessoa{
2      private String nome;
3      public Pessoa() {}
4      public Pessoa(String $nome) {
5          this.nome=$nome;
6      }
7      public String getNome() {
8          return this.nome;
9      }
10     public void setNome(String $nome) {
11         this.nome = $nome;
12     }
13 }
```

OO4 - Encapsulamento, Construtores e Herança

Construtores

- Utilizando os construtores da classe Pessoa:

```
1 public class TestaConstrutor{
2     public static void main (String[] args) {
3         Pessoa p1 = new Pessoa();
4         Pessoa p2 = new Pessoa("Marcia");
5         p1.setNome("Reinaldo");
6         System.out.println("Nome 1: " + p1.getNome());
7         System.out.println("Nome 2: " + p2.getNome());
8     }
9 }
```

OO4 - Encapsulamento, Construtores e Herança

Construtor

Atenção: Não copie ou cole nenhum exercício. A repetição é intencional para criar fluência na linguagem.

1. Crie a classe `Cliente`, com os seguintes atributos privados:
 1. "cpf" do tipo `String`;
 2. "nome" do tipo `String`;
2. Crie métodos `get's` e `set's` para `Cliente`;
3. Crie construtores vazios para as classes: `Banco`, `Agencia`, `Conta`, `ContaCorrente`, `ContaPoupanca` e `Cliente`;
4. Crie construtores com todos os atributos para as classes: `Banco`, `Agencia`, `Conta`, `ContaCorrente`, `ContaPoupanca` e `Cliente`;

5. Na classe `AplicacaoFinanceira`:

1. Crie um objeto `cliente1` da classe `Cliente` utilizando o construtor completo;
2. Mostre os valores dos atributos de `cliente1` utilizando os métodos `get's`;
3. Altere a criação dos objetos `Banco`, `Agencia`, `Conta`, `ContaCorrente` e `ContaPoupanca` utilizando construtores completos;

6. Compile a classe `AplicacaoFinanceira` e corrija os possíveis erros.

5. Na classe `AplicacaoFinanceira`:

1. Crie um objeto `cliente1` da classe `Cliente` utilizando o construtor completo;
2. Mostre os valores dos atributos de `cliente1` utilizando os métodos `get's`;
3. Altere a criação dos objetos `Banco`, `Agencia`, `Conta`, `ContaCorrente` e `ContaPoupanca` utilizando construtores completos;

6. Compile a classe `AplicacaoFinanceira` e corrija os possíveis erros.

OO4 - Encapsulamento, Construtores e Herança

Herança

- Até agora usamos o exemplo da classe Pessoa em nossos exemplos e ela continha somente um atributo nome, seus respectivos métodos get's e set's e seus construtores.
- Suponhamos agora que uma nova função de nosso sistema faz com que seja necessária a criação de dois tipos de pessoa: uma **pessoa física**, que possua um atributo cpf do tipo String, e uma **pessoa jurídica**, que possua um atributo cnpj, também do tipo String.

- Dentro do contexto do exemplo, essas duas novas classes substituiriam a classe Pessoa (que deixaria de existir), já que **todos os tipos de pessoas** já são representadas pelas duas novas classes
- E com o que conhecemos de Java até agora, essas duas classes seriam implementadas da forma a seguir...

OO4 - Encapsulamento, Construtores e Herança

Herança

```
1 public class PessoaFisica{
2     private String nome;
3     private String cpf;
4
5     public PessoaFisica() {}
6
7     public PessoaFisica(String $nome, String $cpf) {
8         this.nome = $nome;
9         this.cpf = $cpf;
10    }
```

```
1 public class PessoaJuridica{
2     private String nome;
3     private String cnpj;
4
5     public PessoaJuridica() {}
6
7     public PessoaJuridica(String $nome, String $cnpj) {
8         this.nome = $nome;
9         this.cnpj = $cnpj;
10    }
```

OO4 - Encapsulamento, Construtores e Herança

Herança

```
12 public String getName() {  
13     return nome;  
14 }  
15  
16 public void setName(String $nome) {  
17     this.nome = $nome;  
18 }  
19  
20 public String getCpf() {  
21     return this.cpf;  
22 }  
23  
24 public void setCpf(String $cpf) {  
25     this.cpf = $cpf;  
26 }  
27 }
```

Muita repetição de código, não?

```
12 public String getName() {  
13     return this.nome;  
14 }  
15  
16 public void setName(String $nome) {  
17     this.nome = $nome;  
18 }  
19  
20 public String getCnpj() {  
21     return this.cnpj;  
22 }  
23  
24 public void setCnpj(String $cnpj) {  
25     this.cnpj = $cnpj;  
26 }  
27 }
```

Herança

- **Herança** é um relacionamento estabelecido entre duas ou mais classes em que uma classe é conhecida como **classe pai** ou **superclasse** e a(s) outra(s) é(são) conhecida(s) como **classe(s) filha(s)** ou **subclasse(s)**.
- Uma classe filha **herda** todos os **atributos**, **métodos** e **relacionamentos** definidos pela sua classe pai.

OO4 - Encapsulamento, Construtores e Herança

Herança

- Para descobrir se há herança entre duas classes, basta usar a pergunta **é um(a)** da seguinte forma:
 - Uma pessoa física **é uma** pessoa? Sim; então existe herança
 - Uma pessoa jurídica **é uma** pessoa? Sim; então existe herança
 - Um carro **é uma** pessoa? Não; então não existe herança
 - Uma pessoa **é uma** pessoa jurídica? Não, nem todas as pessoas são pessoas jurídicas. A herança tem somente um sentido.

OO4 - Encapsulamento, Construtores e Herança

Herança

- As principais vantagens da herança são:
 - A **reutilização de código** (como o que é definido pela classe pai é herdado pela classe filha, não há necessidade de se escrever o mesmo código duas vezes);
 - A **possibilidade de polimorfismo** (próximo tópico do curso)
- Em Java, a herança se dá pelo uso da palavra-chave `extends` na declaração de uma classe filha.
 - Java permite somente **herança simples** (ou seja, não existe o conceito de herança múltipla como em outras linguagens como c++)

OO4 - Encapsulamento, Construtores e Herança

Herança

```
1 public class Pessoa{
2     private String nome;
3
4     public Pessoa() {}
5
6     public Pessoa(String $nome) {
7         this.nome = $nome;
8     }
9
10    public String getNome() {
11        return this.nome;
12    }
13
14    public void setNome(String $nome) {
15        this.nome = $nome;
16    }
17 }
```

- **A superclasse**
Pessoa continua com o que é de comum às **subclasses**
PessoaFisica e PessoaJuridica.

OO4 - Encapsulamento, Construtores e Herança

Herança

```
1 public class PessoaFisica extends Pessoa{
2     private String cpf;
3
4     public PessoaFisica() {}
5
6     public PessoaFisica(String $nome, String $cpf) {
7         super.setNome($nome);
8         this.cpf = $cpf;
9     }
10
11     public String getCpf() {
12         return this.cpf;
13     }
14
15     public void setCpf(String $cpf) {
16         this.cpf = $cpf;
17     }
18 }
```

- Note que a classe PessoaFisica define apenas o atributo cpf, mas herda o atributo nome e seus get's e set's.
- this se refere ao **objeto corrente**, super se refere ao objeto da **superclasse**. Por meio de super podemos chamar **qualquer método público da superclasse**.

OO4 - Encapsulamento, Construtores e Herança

Herança

```
1 public class PessoaFisica extends Pessoa{
2     private String cpf;
3
4     public PessoaFisica() {}
5
6     public PessoaFisica(String $nome, String $cpf) {
7         super.setNome($nome);
8         this.cpf = $cpf;
9     }
10
11     public String getCpf() {
12         return this.cpf;
13     }
14
15     public void setCpf(String $cpf) {
16         this.cpf = $cpf;
17     }
18 }
```

- Dentro de construtores é possível chamar **super**, sem nenhum nome de método para se referir a um **construtor da superclasse**.
- Caso chame o construtor da superclasse (**super**), é preciso que essa invocação seja **o primeiro comando do construtor da subclasse**.

OO4 - Encapsulamento, Construtores e Herança

Herança

- Os atributos e métodos da superclasse Pessoa podem ser utilizados sem diferença nenhuma na classe TestaHeranca.

```
1 public class TestaHeranca{
2     public static void main (String[] args) {
3         PessoaFisica p1 = new PessoaFisica();
4         PessoaJuridica p2 = new PessoaJuridica();
5         p1.setNome("Reinaldo");
6         p1.setCpf("123.456.789-00");
7         p2.setNome("39Dev");
8         p2.setCnpj("123.456.789/0001-00");
9         System.out.println("Nome : " + p1.getNome());
10        System.out.println("CPF : " + p1.getCpf());
11        System.out.println("Nome : " + p2.getNome());
12        System.out.println("CNPJ : " + p2.getCnpj());
13    }
14 }
```

OO4 - Encapsulamento, Construtores e Herança

Herança

Atenção: Não copie ou cole nenhum exercício. A repetição é intencional para criar fluência na linguagem.

1. Crie as classes `ClientePessoaFisica`, com os seguintes atributos privados:
 1. "cpf" do tipo `String`;
 2. "rg" do tipo `String`;
2. Crie as classes `ClientePessoaJuridica`, com o seguinte atributo privado:
 1. "cnpj" do tipo `String`;
3. Crie métodos `get's` e `set's` para `ClientePessoaFisica` e `ClientePessoaJuridica`;

4. Faça com que as classes `ClientePessoaFisica` e `ClientePessoaJuridica` herdem a classe `Cliente`
5. Crie construtores vazios para as classes: `ClientePessoaFisica` e `ClientePessoaJuridica`;
6. Crie construtores com todos os atributos para as classes: `ClientePessoaFisica` e `ClientePessoaJuridica`, invocando construtor de `Cliente`;
7. Faça com que as classes `ContaCorrente` e `ContaPoupanca` herdem a classe `Conta`;

8. Na classe `AplicacaoFinanceira`:

1. Remova a criação do objeto `conta1`, copiando seus atributos para `contaCorrente1` e `contaPoupanca1`;
2. Remova a criação do objeto `cliente`, copiando seus tributos para `clientePessoaFisica1` e crie novos para `clientePessoaJuridica1`;
3. Ajuste a saída de tela com os dados do sistema.