Abstração, Encapsulamento e Herança: Pilares da POO em Java

Veja neste artigo três dos pilares da Programação Orientada a Objetos: Abstração, Encapsulamento e Herança. Serão apresentados, além dos conceitos, exemplos práticos implementados na linguagem Java.

Introdução

A Programação Orientada a Objetos conhecida como POO, é onde o desenvolvedor tem de começar a pensar fora da caixa, a imaginar uma forma aonde será preciso recorrer ao mundo real para o desenvolvimento das aplicações, pois hoje toda a programação em Java é orientada a objetos.

Para obter esse entendimento, é necessário conhecer alguns dos pilares da Orientação a Objetos que são: **Abstração, Encapsulamento, Herança e Polimorfismo**. Neste artigo o pilar do Polimorfismo não será visto.

1º Pilar - Abstração

É utilizada para a definição de entidades do mundo real. Sendo onde são criadas as classes. Essas entidades são consideradas tudo que é real, tendo como consideração as suas características e ações, veja na Figura 1 como funciona.

Entidade	Características	Ações
Carro, Moto	tamanho, cor, peso, altura	acelerar, parar, ligar, desligar
Elevador	tamanho, peso máximo	subir, descer, escolher andar
Conta Banco	saldo, limite, número	depositar, sacar, ver extrato

Figura 1: Abstrações do mundo real

Uma classe é reconhecida quando tem a palavra reservada “class”. Na Listagem 1 é mostrada a classe “Conta” com seus atributos (características) e métodos (ações). Para saber mais sobre métodos acesse o link: http://www.devmedia.com.br/trabalhando-com-metodos-em-java/25917.

Listagem 1: Exemplo de abstração da classe Conta.

```
public class Conta {
        int numero;
        double saldo;
        double limite;

        void depositar(double valor){
                this.saldo += valor;
        }

        void sacar(double valor){
                this.saldo -= valor;
        }

        void imprimeExtrato(){
                System.out.println("Saldo: "+this.saldo);
        }
}
```

2º pilar - Encapsulamento

É a técnica utilizada para esconder uma ideia, ou seja, não expôr detalhes internos para o usuário, tornando partes do sistema mais independentes possível. Por exemplo, quando um controle remoto estraga apenas é trocado ou consertado o controle e não a televisão inteira. Nesse exemplo do controle remoto, acontece a forma clássica de encapsulamento, pois quando o usuário muda de canal não se sabe que programação acontece entre a televisão e o controle para efetuar tal ação.

Como um exemplo mais técnico podemos descrever o que acontece em um sistema de vendas, aonde temos cadastros de funcionários, usuários, gerentes, clientes, produtos entre outros. Se por acaso acontecer um problema na parte do usuário é somente nesse setor que será realizada a manutenção não afetando os demais.

Em um processo de encapsulamento os atributos das classes são do tipo **private**. Para

acessar esses tipos de modificadores, é necessário criar métodos **setters** e **getters**.

Por entendimento os métodos setters servem para alterar a informação de uma propriedade de um objeto. E os métodos getters para retornar o valor dessa propriedade.

Veja um exemplo de encapsulamento, na Listagem 2 gera-se os atributos privados (private) e é realizado o processo de geração dos métodos setters e getters.

Métodos getters	Métodos setters

public String getNome() {
return nome;
}
 	
public double getSalario() {
return salario;
}
	public void setNome(String nome) {
	this.nome = nome;
	}
	
	public void setSalario(double salario) {
	this.salario = salario;
	}

Figura 2: Métodos getters e setters

Listagem 2: Encapsulamento da classe Funcionario.

```
public class Funcionario {
	private double salario;
	private String nome;

	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}

	public void setSalario(double salario) {
```

```
                this.salario = salario;
        }

        public double getSalario() {
                return salario;
        }
}
```

Na Listagem 3, é instanciado a classe “Funcionario”, onde a variável de referência é usada para invocar os métodos setters, informando algum dado. Ao final, é usado os métodos getters dentro do “System.out.println” para gerar a saída dos resultados que foram passados nos métodos setters.

Listagem 3: Classe Testadora dos métodos getters e setters.

```
public class TestaFuncionario {

        public static void main(String[] args) {
                Funcionario funcionario = new Funcionario();
                funcionario.setNome("Thiago");
                funcionario.setSalario(2500);

                System.out.println(funcionario.getNome());
                System.out.println(funcionario.getSalario());

        }
}
```

3º pilar - Herança

Na Programação Orientada a Objetos o significado de herança tem o mesmo significado para o mundo real. Assim como um filho pode herdar alguma característica do pai, na Orientação a Objetos é permitido que uma classe herde atributos e métodos da outra, tendo apenas uma restrição para a herança. Os modificadores de acessos das classes, métodos e atributos só podem estar com visibilidade **public** e **protected**

para que sejam herdados.

Uma das grandes vantagens de usar o recurso da herança é na reutilização do código. Esse reaproveitamento pode ser acionado quando se identifica que o atributo ou método de uma classe será igual para as outras. Para efetuar uma herança de uma classe é utilizada a palavra reservada chamada **extends**.

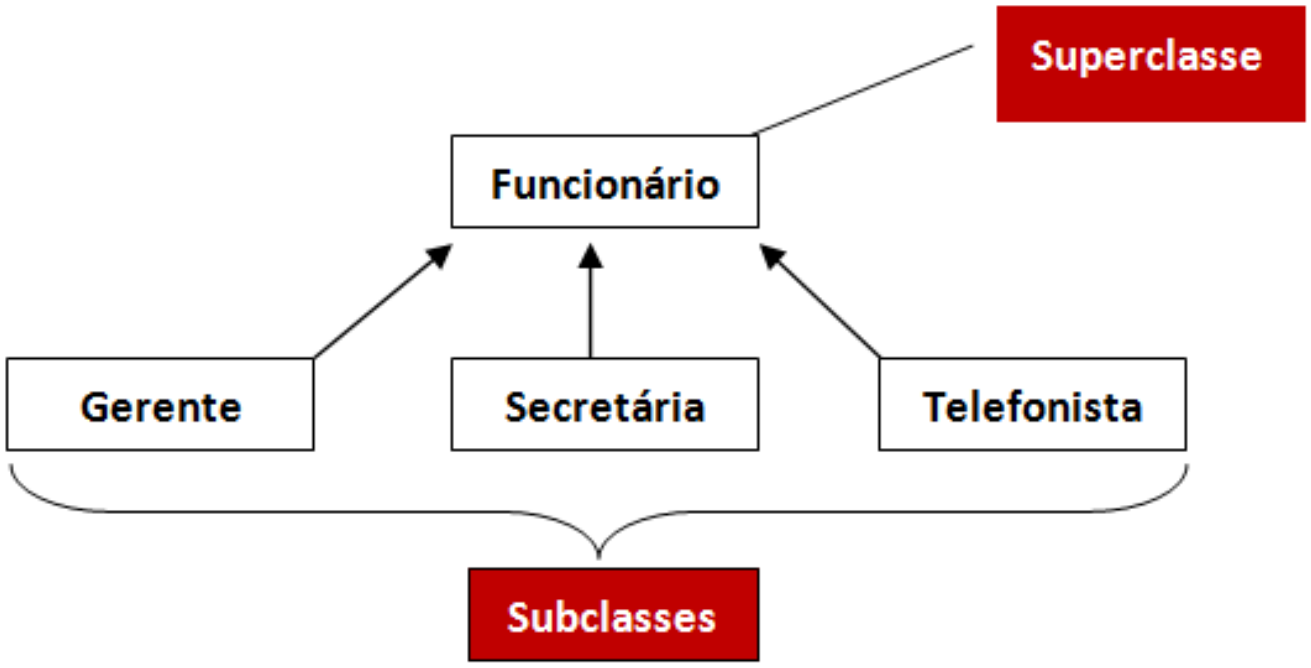


Figura 3: Hierarquia das classes

Para saber se estamos aplicando a herança corretamente, realiza-se o teste "**É UM**". Esse teste simples ajuda a detectar se a subclasse pode herdar a superclasse.

Por exemplo, na Figura 3, está mostrando que a classe "Gerente" herda da classe "Funcionário", se for aplicado o teste "É UM" nota-se que o teste é aprovado, pois o "Gerente" também "É UM" Funcionário.

Veja nos exemplos abaixo como aplicar o recurso da herança em uma classe.

Na Listagem 4, existe a superclasse "Funcionario" que servirá de base para as subclasses usarem seus atributos ou métodos.

Listagem 4: Superclasse Funcionario.

```
public class Funcionario {
	private String nome;
	private double salario;

	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}

	public double getSalario() {
		return salario;
	}

	public void setSalario(double salario) {
		this.salario = salario;
	}

	public double calculaBonificacao(){
		return this.salario * 0.1;
	}

}
```

No exemplo da Listagem 5, podemos ver que a classe “Gerente” está herdando da classe “Funcionario” através da palavra reservada extends. Acontece também a mudança do comportamento de herança, a partir do método “calculaBonificacao” que é sobrescrito, pois entende-se que o valor da classe “Gerente” é diferente para as demais.

Listagem 5: Sublclasse Gerente.

```
public class Gerente extends Funcionario {
	private String usuario;
	private String senha;
```

```
        public String getUsuario() {
                return usuario;
        }

        public void setUsuario(String usuario) {
                this.usuario = usuario;
        }

        public String getSenha() {
                return senha;
        }

        public void setSenha(String senha) {
                this.senha = senha;
        }

        public  double calculaBonificacao(){
                return this.getSalario() * 0.6 + 100;
        }

}
```

Listagem 6: Subclasse Secretaria.

```
public class Secretaria extends Funcionario {
        private int ramal;

        public void setRamal(int ramal) {
                this.ramal = ramal;
        }

        public int getRamal() {
                return ramal;
        }
}
```

Listagem 7: Subclasse Telefonista.

```
public class Telefonista extends Funcionario {
        private int estacaoDeTrabalho;
```

```
        public void setEstacaoDeTrabalho(int estacaoDeTrabalho) {
                this.estacaoDeTrabalho = estacaoDeTrabalho;
        }

        public int getEstacaoDeTrabalho() {
                return estacaoDeTrabalho;
        }
}
```

Na Listagem 8 são apresentadas todas as classes e mostrada a reutilização de código, um exemplo são os atributos “nome” e “salario”. Portanto, não foi preciso criar em todas as classes, apenas criou-se na superclasse. Apenas lembrando que o acesso dos atributos ou métodos de uma superclasse é permitido somente se estão definidos com o modo de visibilidade como “public” ou “protected”.

Listagem 8: Classe Testadora

```
public class TestaFuncionario {

        public static void main(String[] args) {

                Gerente gerente = new Gerente();
                gerente.setNome("Carlos Vieira");
                gerente.setSalario(3000.58);
                gerente.setUsuario("carlos.vieira");
                gerente.setSenha("5523");

                Funcionario funcionario = new Funcionario();
                funcionario.setNome("Pedro Castelo");
                funcionario.setSalario(1500);

                Telefonista telefonista = new Telefonista();
                telefonista.setNome("Luana Brana");
                telefonista.setSalario(1300.00);
                telefonista.setEstacaoDeTrabalho(20);

                Secretaria secretaria = new Secretaria();
                secretaria.setNome("Maria Ribeiro");
                secretaria.setSalario(1125.25);
```

```
                secretaria.setRamal(5);

                System.out.println("##### Gerente #####");
                System.out.println("Nome.: "+gerente.getNome());
                System.out.println("Salário.: "+gerente.getSalario());
                System.out.println("Usuário.: "+gerente.getUsuario());
                System.out.println("Senha.: "+gerente.getSenha());
                System.out.println("Bonificação.: "+gerente.calculaBonificacao());
                System.out.println();

                System.out.println("##### Funcionário #####");
                System.out.println("Nome.: "+funcionario.getNome());
                System.out.println("Salário.: "+funcionario.getSalario());
                                                                    lculaBonificacao()
```



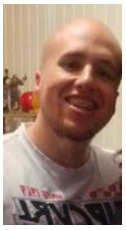
```
                System.out.println("##### Telefonista #####");
                System.out.println("Nome.: "+telefonista.getNome());
                System.out.println("Salário.: "+telefonista.getSalario());
                System.out.println("Estação de Trabalho.: "+telefonista.getEstacaoDe
                System.out.println("Bonificação.: "+telefonista.calculaBonificacao()
                System.out.println();

                System.out.println("##### Secretária #####");
                System.out.println("Nome.: "+secretaria.getNome());
                System.out.println("Salário.: "+secretaria.getSalario());
                System.out.println("Ramal.: "+secretaria.getRamal());
                System.out.println("Bonificação.: "+secretaria.calculaBonificacao())
                System.out.println();
        }

}
```

Conclusão

Portanto foram mostrados alguns exemplos e conceitos dos três pilares da Programação Orientada a Objetos com implementações na linguagem Java. Espero que tenham gostado e até a próxima!



Thiago Vinícius Varallo Palmeira

Um entusiasta das linguagens de programação. Fórum: varallos.com.br/foruns Site: varallos.com.br YouTube: youtube.com/varallos1 'Deveria existir uma pitada de diletantismo na crítica. Pois o diletante é um entusias [...]

O que você achou deste post?

Gostei (6) (0)

+ Mais conteúdo sobre Java

Todos os comentarios (5)

Postar dúvida / Comentário

Meus comentarios

Gustavo Christino

Estava lendo esse artigo muito interessante sobre orientação a objetos e percebi um pequeno erro. Na Listagem 4 os métodos deveriam ser do tipo "protected", mas estão como "private". Dessa forma, apenas os 'getters' e 'setters' seriam herdados. Obrigado.

[há +1 ano] - Responder

Gustavo Christino

Correção...

-> Os atributos deveriam ser "protected"...

[há +1 ano] - Responder

Gustavo Christino

Estava lendo esse artigo muito interessante sobre orientação a objetos e percebi um pequeno erro. Na Listagem 4 os métodos deveriam ser do tipo "protected", mas estão como "private". Dessa forma, apenas os 'getters' e 'setters' seriam herdados. Obrigado.

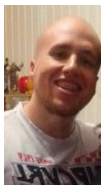
[há +1 ano] - Responder

Gustavo Christino

Correção...

-> Os atributos deveriam ser "protected"...

[há +1 ano] - Responder



[autor] Thiago Vinícius Varallo Palmeira

Olá Gustavo os atributos ficaram private, pois quis acessar os métodos e não as variáveis, essa seria uma das formas para deixar mais encapsulado esses dados. Claro que temos também essa opção pois se falando em herança também seria válido aplicar dessa maneira que comentou, ou também aplicar nos métodos o modificador protected. :)

[há +1 ano] - Responder

Mais posts

Artigo

Gerando QrCode com Java ZXing

Artigo

Usando pacote javax.mail para enviar email em Java

Artigo

Programação Funcional: código limpo e padrões de projeto

Artigo

Código limpo e padrões de projeto na programação funcional

Artigo

Elasticsearch: Aprendizado de Máquina para Web com Apache Spark e Crawler4J

Video aula

Configurando DataSources no Tomcat - Curso de Java Avançado - Aula 15

Video aula

Configurando o war do Probe - Curso de Java Avançado - Aula 14

Artigo

Como criar aplicações multiplataforma com React JavaScript

Artigo

Introdução ao Framework ORM Android JPDroid – Parte 2

Listar mais conteúdo