

Abstração, Encapsulamento e Herança: Pilares da POO em 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.

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 |
|---|---|
| <pre>public String getNome() { return nome; }</pre> | <pre>public void setNome(String nome) { this.nome = nome; }</pre> |
| <pre>public double getSalario() { return salario; }</pre> | <pre>public void setSalario(double salario) { this.salario = salario; }</pre> |

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**.

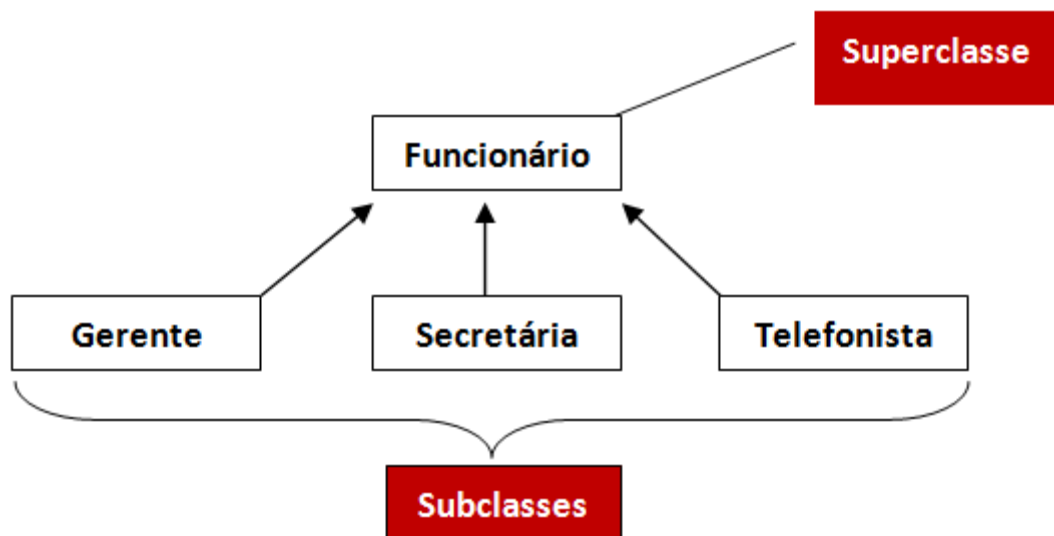


Figura 3: Hierarquia das classes

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: Subclasse 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 “salário”. 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());
        System.out.println("Bonificação.: 
"+funcionario.calculaBonificacao());
        System.out.println();

        System.out.println("##### Telefonista #####");
        System.out.println("Nome.: "+telefonista.getNome());
    }
}

```

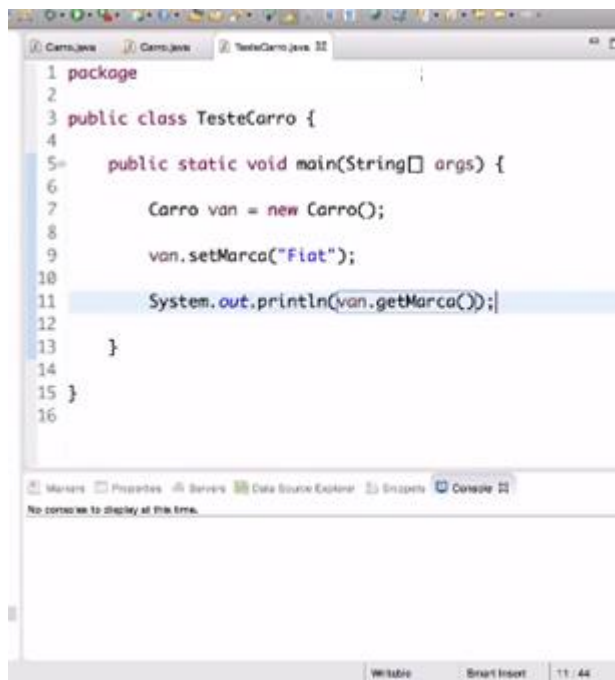
```
        System.out.println("Salário.:  
"+telefonista.getSalario());  
        System.out.println("Estação      de      Trabalho.:  
"+telefonista.getEstacaoDeTrabalho());  
        System.out.println("Bonificação.:  
"+telefonista.calculaBonificacao());  
        System.out.println();  
  
        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();  
    }  
  
}
```

Encapsulamento :

A ocultação de informações é considerada parte do **encapsulamento**, mas se fizermos uma pesquisa na internet, podemos encontrar a seguinte definição para **encapsulamento**: Um mecanismo da linguagem de programação para restringir o acesso a alguns componentes dos objetos, escondendo os dados de uma classe



```
3 public class Carro {
4
5     private String marca;
6     private String modelo;
7     private int numPassageiros;
8     private double capCombustivel;
9     private double consumoCombustivel;
10
11     public String getMarca(){
12         return this.marca;
13     }
14
15     public void setMarca(String marca){
16         this.marca = marca;
17     }
18
19     p
```



```
1 package
2
3 public class TesteCarro {
4
5     public static void main(String[] args) {
6
7         Carro van = new Carro();
8
9         van.setMarca("Fiat");
10
11         System.out.println(van.getMarca());
12
13     }
14 }
15 }
16 }
```


**"Os atributos da classe
devem ser alterados
dentro da própria
classe".**

Herança:

A **herança** é um mecanismo da Orientação a Objeto que permite criar novas classes a partir de classes já existentes, aproveitando-se das características existentes na classe a ser estendida. ... A linguagem **Java** permite o uso de **herança** simples, mas não permite a implementação de **herança** múltipla.

Primeiro criase a classe Pessoa.java com o código descrito abaixo

```
1. package heranca;
2.
3. Pessoa.java
4. public class Pessoa {
5.     String nome,
6.     idade,
7.     endereco= "R: Java ,501";
8.
9.
10.
11.     public void ImprimeNome(){
12.         System.out.println("o nome é:");
13.         System.out.println("Endereco: " + endereco);
14.
15.
16.     }
17.
18.
19. }
```

logo criamos a classe Fornecedor.java com o código abaixo:

Pode se observar que o método ImprimeNome está sendo sobrescrito diferentemente de sua classe pai Pessoa.java esse conceito se dá o nome de **override!!!**

```
1. Fornecedor.java
2.
3. package heranca;
4.
5. Fornecedor.java
6.
7. public class Fornecedor extends Pessoa {
8.
9.
10.     String cnpj;
11.
12.     public void ImprimeNome (){
13.         System.out.println("O nome do fornecedor é : " + nome + "\n Cnpj: " + cn
14.         pj);
15.     }
```

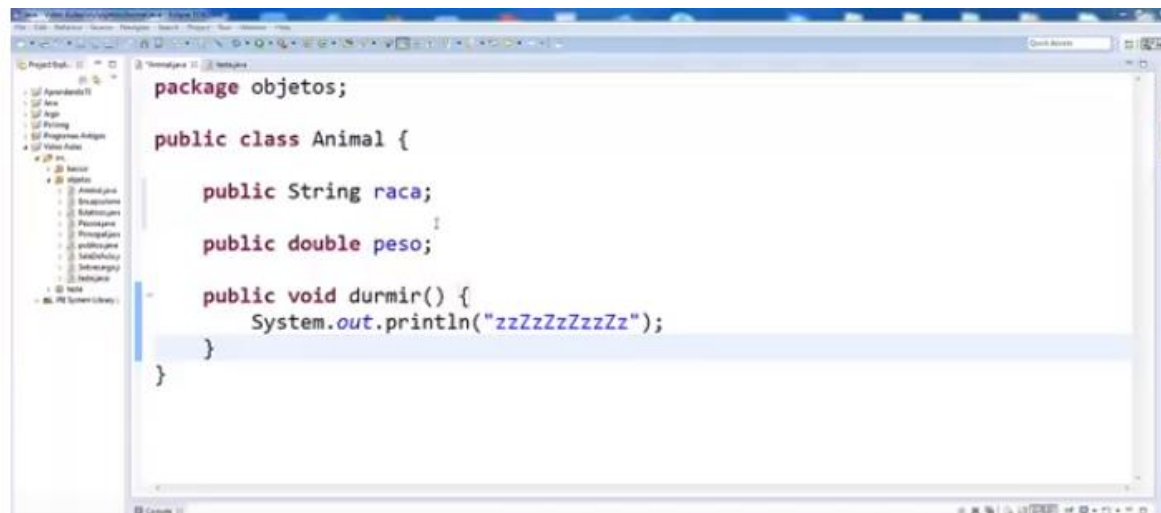
a classe Cliente.java com o código abaixo:
utilizando também o método ImprimeNome().

```
1. cliente.java
2.
3. package heranca;
4.
5. public class Cliente extends Pessoa {
6.
7.     String cpf;
8.
9.     public void ImprimeNome () {
10.
11.         System.out.println("Nome do cliente é : " + nome + "\n Nº CPF: " + cpf +
12.             "\n Seu endereço : " + endereco);
13.
14.     }
15.
16. }
```

e a classe Principal.java, que será a classe executora.
Note que a classe Principal.java faz a chamada dos métodos e possui o método principal o main por isso estou nomeando a mesma como executora.

```
1. Principal.java
2. package heranca;
3.
4. public class Principal {
5.
6.
7.     public static void main(String[] args) {
8.
9.         Cliente c = new Cliente();
10.
11.         c.nome="Luiz";
12.         c.cpf="073.777.796-21";
13.         c.ImprimeNome();
14.
15.         Fornecedor f = new Fornecedor ();
16.
17.         f.nome="Deltatronic";
18.         f.cnpj="073.856.9856.52-10";
19.
20.         f.ImprimeNome();
21.
22.
23.     }
24.
25. }
```

Exemplo



```
package objetos;

public class Animal {

    public String raza;

    public double peso;

    public void dormir() {
        System.out.println("zzZzZzzZzz");
    }
}
```



New Java Class

Create a new Java class.

Source folder: Video Aulas/src Browse...

Package: objetos Browse...

☐ Enclosing type: Browse...

Name: Cachorro

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☒ final ☐ static

Superclass: java.lang.Object Browse...

Interfaces: Add... Remove

Which method stubs would you like to create?

☒ public static void main(String[] args)

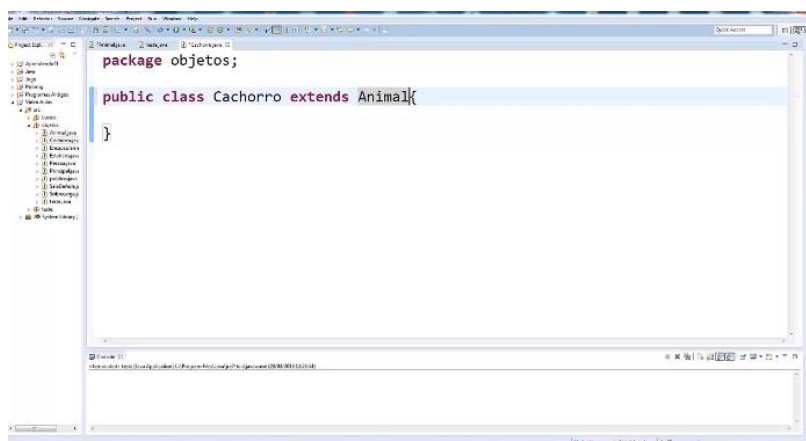
☐ Constructors from superclass

☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

? Finish Cancel



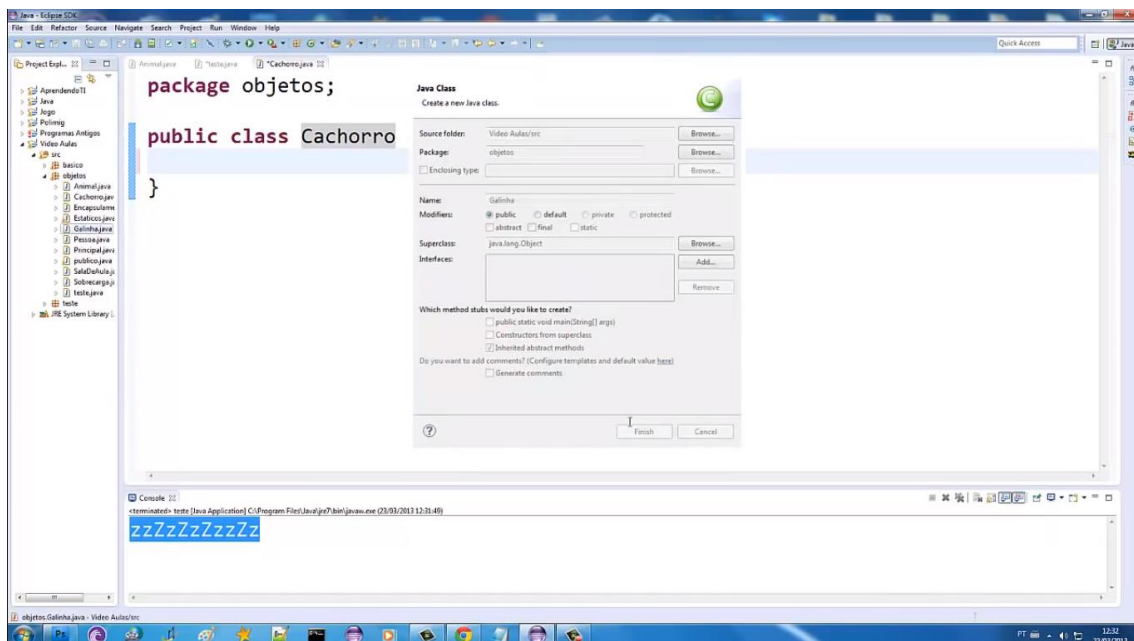
```
package objetos;

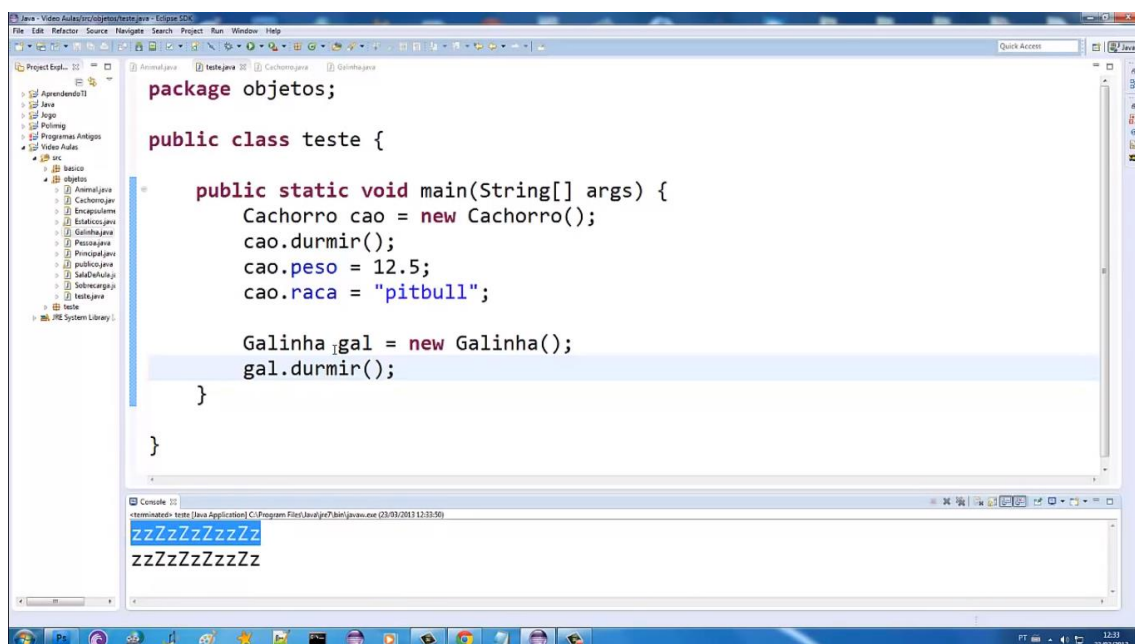
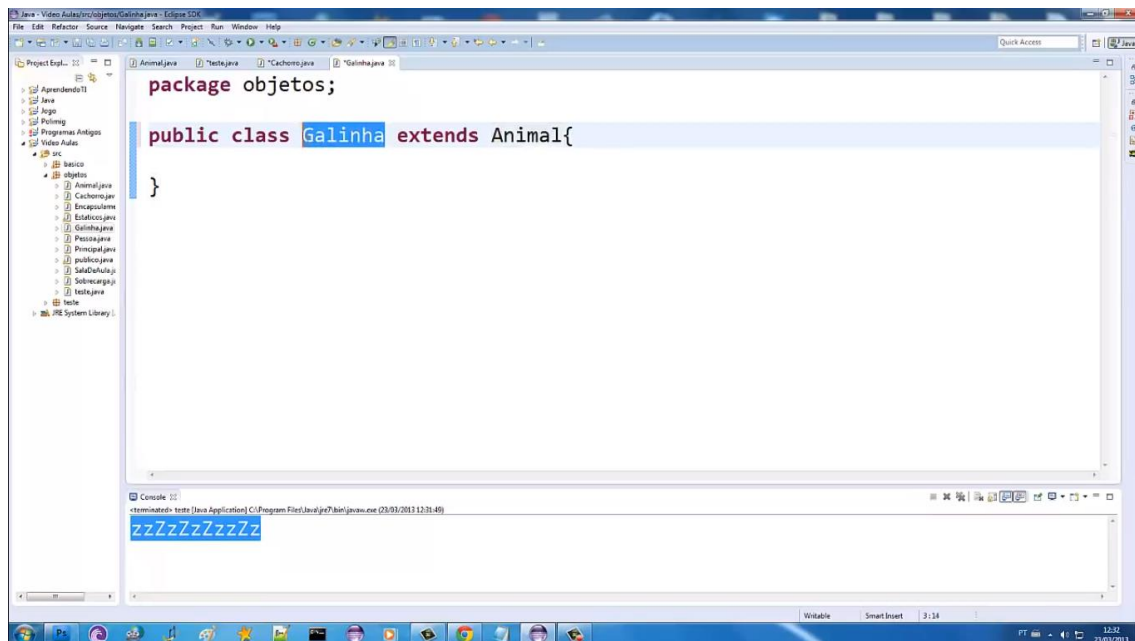
public class Cachorro extends Animal{
}
```

```
package objetos;

public class teste {

    public static void main(String[] args) {
        Cachorro cao = new Cachorro();
        cao.durmir();
        cao.peso = 12.5;
        cao.raca = "pitbull";
    }
}
```

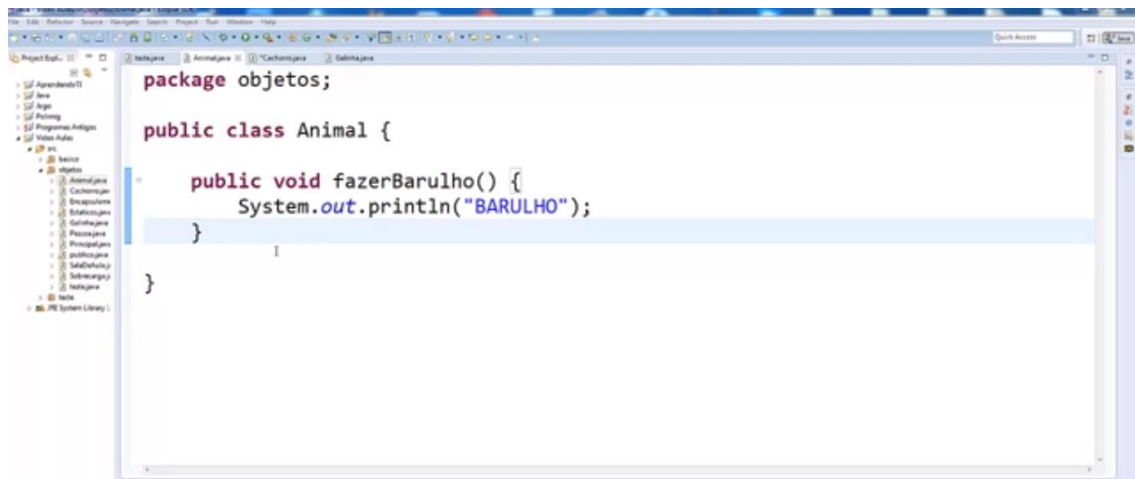




Em resumo simplifica o reaproveitamento de metodos

Polimorfismo :

Polimorfismo significa "muitas formas", é o termo definido em linguagens orientadas a objeto, como porexemplo **Java**, **C#** e **C++**, que permite ao desenvolvedor usar o mesmo elemento de formas diferentes. **Polimorfismo** denota uma situação na qual um objeto pode se comportar de maneiras diferentes ao receber uma mensagem

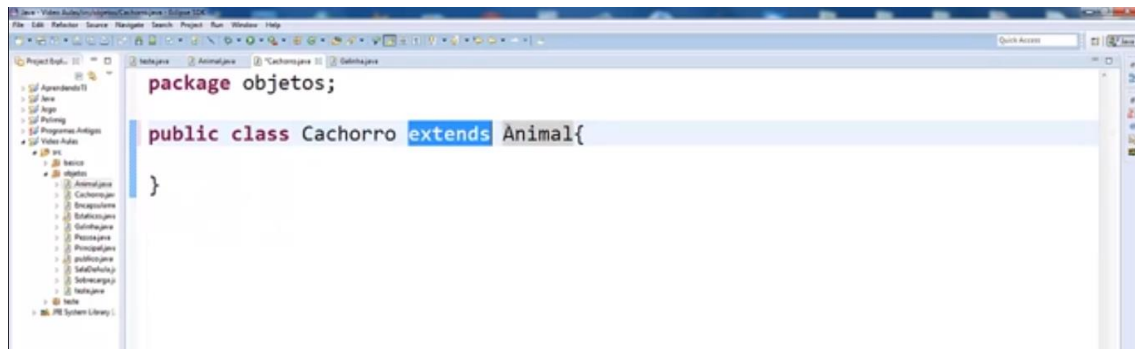


The screenshot shows an IDE window with a project explorer on the left and a code editor on the right. The project explorer shows a package named 'objetos'. The code editor contains the following Java code:

```
package objetos;

public class Animal {

    public void fazerBarulho() {
        System.out.println("BARULHO");
    }
}
```

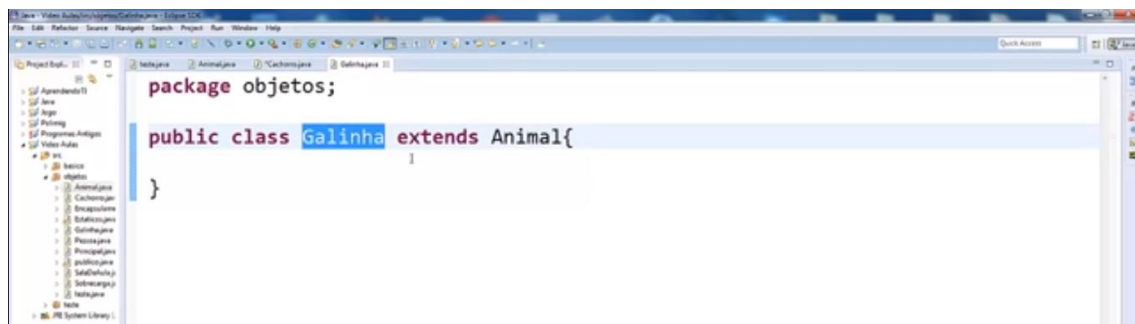


The screenshot shows the same IDE window. The code editor now contains the following Java code:

```
package objetos;

public class Cachorro extends Animal{

}
```



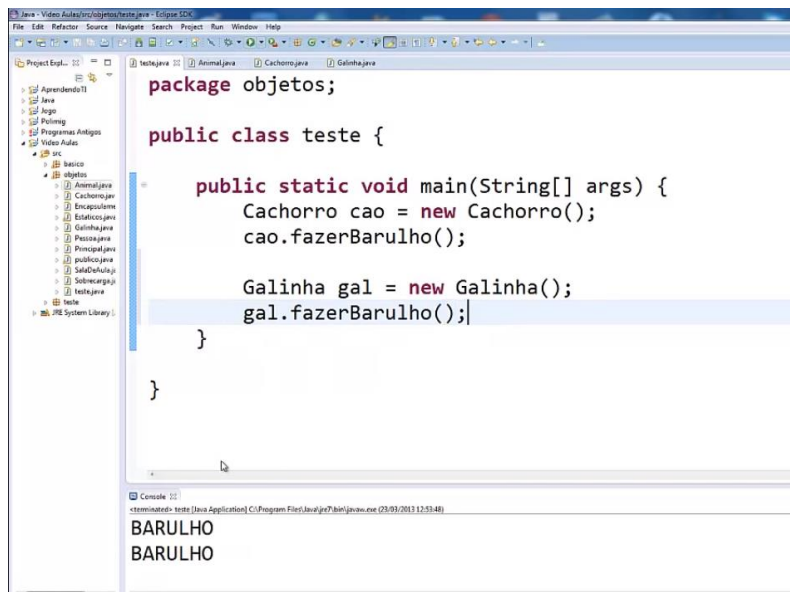
The screenshot shows the same IDE window. The code editor now contains the following Java code:

```
package objetos;

public class Galinha extends Animal{

}
```

Herdou de animal



```
package objetos;

public class teste {

    public static void main(String[] args) {
        Cachorro cao = new Cachorro();
        cao.fazerBarulho();

        Galinha gal = new Galinha();
        gal.fazerBarulho();
    }
}
```

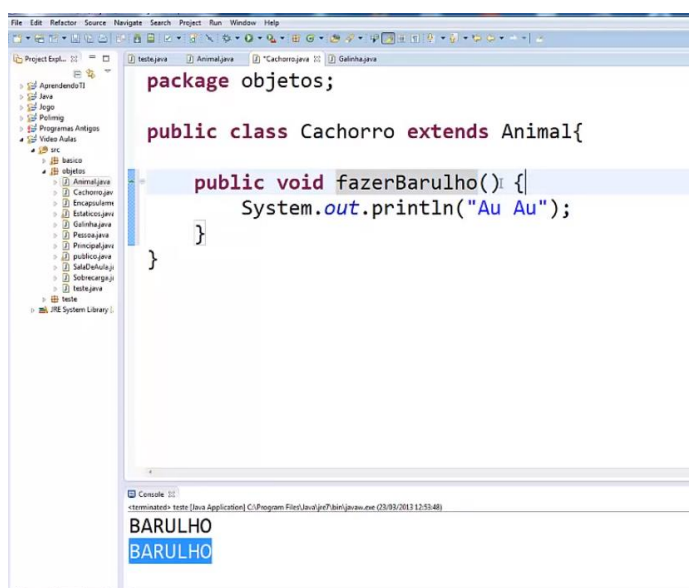
Console

terminated: teste [Java Application] C:\Program Files\Java\j7\bin\java.exe (23/03/2013 12:53:48)

BARULHO
BARULHO

Mas nem todo animal faz o mesmo barulho , certo?

Vamos sobrescrever o método herdado



```
package objetos;

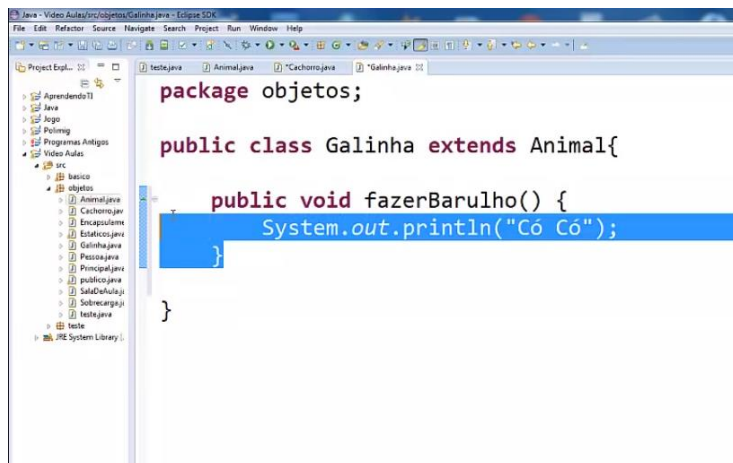
public class Cachorro extends Animal{

    public void fazerBarulho() {
        System.out.println("Au Au");
    }
}
```

Console

terminated: teste [Java Application] C:\Program Files\Java\j7\bin\java.exe (23/03/2013 12:53:48)

BARULHO
BARULHO

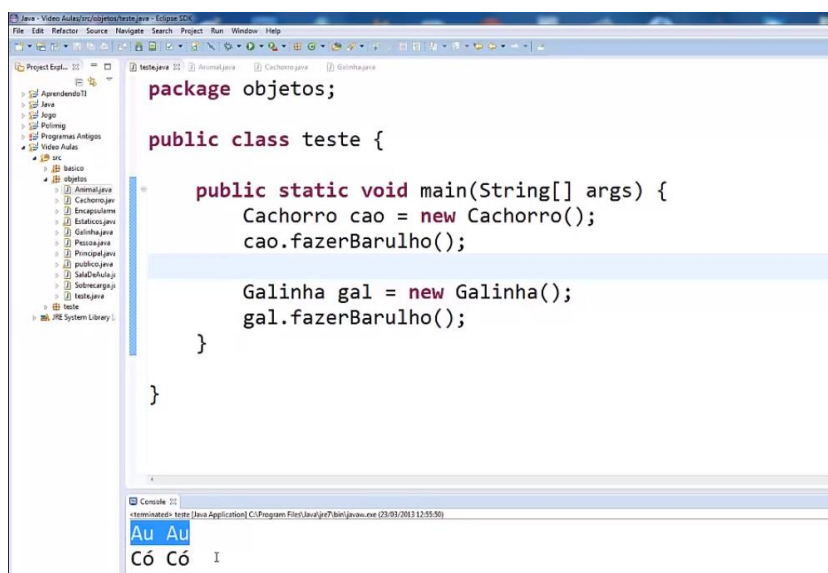


```
package objetos;

public class Galinha extends Animal{

    public void fazerBarulho() {
        System.out.println("Cô Cô");
    }

}
```



```
package objetos;

public class teste {

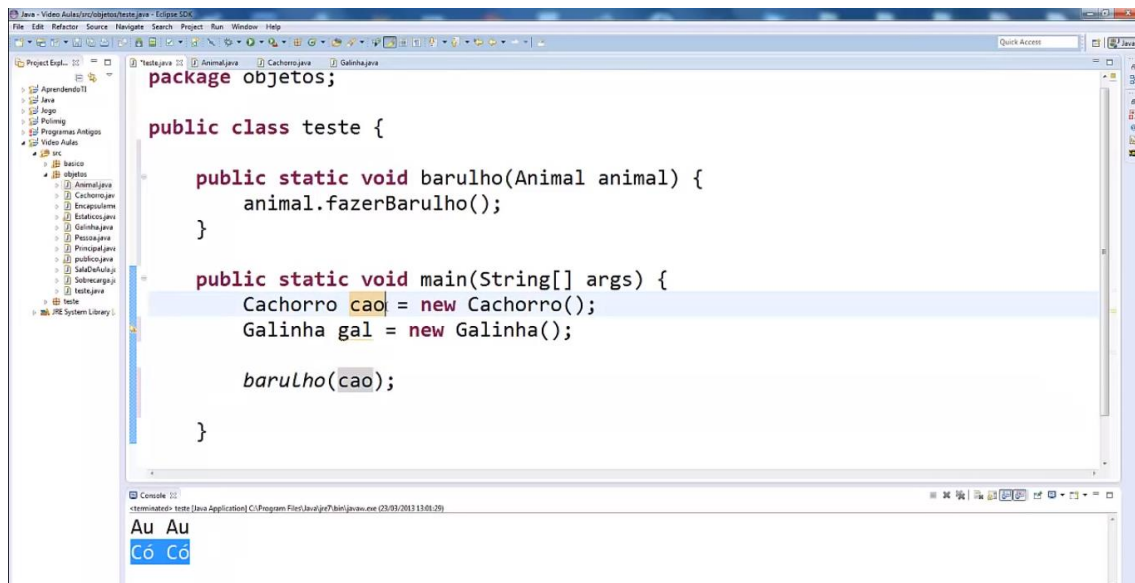
    public static void main(String[] args) {
        Cachorro cao = new Cachorro();
        cao.fazerBarulho();

        Galinha gal = new Galinha();
        gal.fazerBarulho();
    }

}

Au Au
Cô Cô
```

Usando as técnicas de polimorfismo , eu não sei qual o animal ele quer MAS eu quero que faça o barulho certo



The screenshot shows the Eclipse IDE with a Java project named 'Video Aulas'. The package explorer on the left shows a package named 'objetos' containing several classes: 'Animal.java', 'Cachorro.java', 'Galinha.java', 'Encapsulamento.java', 'Estadisticos.java', 'Galinha.java', 'Passo.java', 'Principal.java', 'publico.java', 'SaltoDehula.java', 'Solmeas.java', and 'teste.java'. The main editor displays the code for 'teste.java'.

```
package objetos;

public class teste {

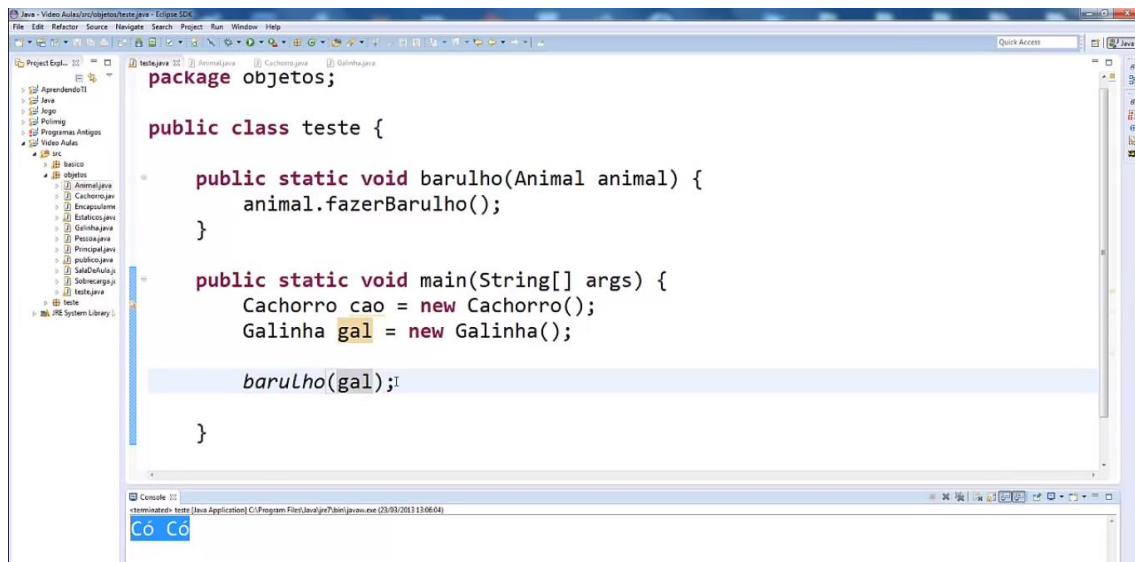
    public static void barulho(Animal animal) {
        animal.fazerBarulho();
    }

    public static void main(String[] args) {
        Cachorro cao = new Cachorro();
        Galinha gal = new Galinha();

        barulho(cao);
    }
}
```

The console at the bottom shows the output of the program:

```
Console:
-terminated- teste [Java Application] C:\Program Files\Java\j7\bin\java.exe (23/03/2013 13:01:39)
Au Au
C6 C6
```



The screenshot shows the Eclipse IDE with the same Java project. The package explorer on the left shows the same package 'objetos' with the same classes. The main editor displays the code for 'teste.java'.

```
package objetos;

public class teste {

    public static void barulho(Animal animal) {
        animal.fazerBarulho();
    }

    public static void main(String[] args) {
        Cachorro cao = new Cachorro();
        Galinha gal = new Galinha();

        barulho(gal);
    }
}
```

The console at the bottom shows the output of the program:

```
Console:
-terminated- teste [Java Application] C:\Program Files\Java\j7\bin\java.exe (23/03/2013 13:06:04)
C6 C6
```