



Hailton David Lemos(hailton@terra.com.br).Tecnólogo em Internet e Redes, Bacharel em Administração de Empresas, Especialista em: Tecnologia da Informação, MBA em Planejamento e Gestão Estratégica, Matemática e Estatística. Trabalha com desenvolvimento de Sistema há mais de 20 anos, atualmente desenvolve sistemas especialistas voltados a Planejamento Estratégico, Tomada de Decisão e Gestão utilizando tecnologias Java, Perl, OWC, VBA, Openoffice, Office.

Encapsulamento, Polimorfismo, Herança.

O paradigma da Orientação a Objetos traz um ganho significativo na qualidade da produção de software, porém grandes benefícios são alcançados quando as técnicas de programação OO são colocadas em prática com o uso de uma tecnologia que nos permita usar todas as características da OO; além de agregar à programação o uso de boas práticas de programação e padrões de projeto, design patterns. Um objeto é uma entidade do mundo real que tem uma identidade. Objetos podem representar entidades concretas, um arquivo no meu computador, uma bicicleta ou entidades conceituais, uma estratégia de jogo, uma política de escalonamento em um sistema operacional. Cada objeto ter sua identidade significa que, dois objetos são distintos mesmo que eles apresentem exatamente as mesmas características.

Objetos são instâncias de classes, que determinam qual informação um objeto contém e como ele pode manipulá-la. Um programa desenvolvido com uma linguagem de programação orientada a objetos manipula estruturas de dados através dos objetos da mesma forma que um programa em linguagem tradicional utiliza variáveis.

Em orientação a objeto, uma classe é uma estrutura que abstrai um conjunto de objetos com características similares. Uma classe define o comportamento de seus objetos através de métodos e os estados possíveis destes objetos através de atributos. Em outros termos, uma classe descreve os serviços providos por seus objetos e quais informações eles podem armazenar. Classes não são diretamente suportadas em todas as linguagens, e são necessárias para que uma linguagem seja orientada a objetos. A programação orientada a objeto tem três pilares, Encapsulamento, herança e polimorfismo, mas antes de tratarmos destes assuntos se faz necessário o entendimento de alguns conceitos iniciais para que tudo possa ficar claro à medida que a aula for dando andamento.

Uma interface nada mais é do que um bloco de código definindo um tipo e os métodos e atributos que esse tipo deve possuir. Na prática o que acontece é que qualquer classe que quiser ser do tipo definido pela interface deve implementar os métodos dessa interface. A interface não contém nenhum código de implementação, apenas assinaturas de métodos e/ou atributos que devem ter seu código implementado nas classes que implementarem essa interface. A Interface define um padrão para especificação do comportamento de classes. Porém, os métodos de uma interface são implementados de maneira particular a cada classe; ou seja, permitem expressar comportamento sem se preocupar com a implementação. Uma interface não possui atributos. Uma classe pode implementar várias interfaces, mas pode ter apenas uma superclasse.

```
1. public class TV {
2.     private int tamanho;
3.     private int canal;
4.     private int volume;
5.     private boolean ligada;
6.     public TV(int tamanho) {
7.         this.tamanho = tamanho;
8.         this.canal = 0;
9.         this.volume = 0;
10.        this.ligada = false;
11.    }
12.    // abaixo vem todos os métodos construtores get e set...
13.    // Encapsulamento
14. }
```

```
1. public interface ControleRemoto {
2.     void mudarCanal(int canal);
3.     void aumentarVolume (int taxa);
4.     void diminuirVolume (int taxa);
5.     boolean ligar();
6.     boolean desligar();
7. }
```

Agora temos nossa interface e a definição do que é a TV, vamos desenvolver duas TVs diferentes, imaginando que fossem duas marcas completamente distintas e que uma não tem nenhuma relação com a outra. Como ambas as TVs irão implementar a interface ControleRemoto, então, no corpo das duas classes devem conter todos os métodos da interface. No exemplo usado abaixo, apenas implementaremos os métodos ligar e desligar.

A TV modelo 001 é uma TV simples, sem muitos recursos que quando acionarmos o comando desligar irá simplesmente desligar.

```
1. public class ModeloTV001 extends TV implements ControleRemoto {
2.     public final String MODELO = "TV001";
3.     public ModeloTV001(int tamanho) {
4.         super(tamanho);
5.     }
6.     public void desligar() {
7.         super.setLigada(false);
8.     }
9.     public void ligar() {
10.        super.setLigada(true);
11.    }
12.    public void aumentarVolume(int taxa) { }
13.    public void diminuirVolume(int taxa) { }
14.    public void mudarCanal(int canal) { }
15. }
```

O modelo X é uma TV mais moderna, que quando acionarmos o comando desligar irá apresentar uma mensagem dizendo "*tchau!*".

```
1. public class ModeloX extends TV implements ControleRemoto {
2.     public final String MODELO = "TV-X";
3.     public ModeloSDX(int tamanho) {
4.         super(tamanho);
5.     }
6.     public void desligar() {
7.         System.out.println("Obrigado por Utilizar a Televisão!");
8.         super.setLigada(false);
9.     }
10.    public void ligar() {
11.        super.setLigada(true);
12.    }
13.    public void aumentarVolume(int taxa) { }
14.    public void diminuirVolume(int taxa) { }
15.    public void mudarCanal(int canal) { }
16. }
```

Como pode ser visto, ambos possuem a mesma ação que é desligar, porém cada um executa de forma diferente.

```
1. public class ExemploInterfaceamento {
2.     public static void main(String[] args) {
3.         ModeloTV001 tv1 = new ModeloTV001(21);
4.         ModeloSDX tv2 = new ModeloX (42);
5.         tv1.ligar();
6.     }
7. }
```

```

6. tv2.ligar();
7. System.out.print("TV1 - modelo " + tv1.MODELO + " está ");
8. System.out.println(tv1.isLigada() ? "ligada" : "desligada");
9. System.out.print("TV2 - modelo " + tv2.MODELO + " está ");
10. System.out.println(tv1.isLigada() ? "ligada" : "desligada");
11. // ambas as TVs estão ligadas e vamos desligá-las
12. System.out.println("Desligando modelo " + tv1.MODELO);
13. tv1.desligar();
14. System.out.println("Desligando modelo " + tv2.MODELO);
15. tv2.desligar();
16. }
17. }

```

Uma classe abstrata nada mais é do que uma especificação conceitual para outras classes. Isso quer dizer que nunca iremos instanciá-la. Ela apenas fornece um modelo para geração de outras classes. Esta nunca está completa, ou seja, servirá apenas para criação de funcionalidades genéricas de classes filhas. Podemos também chamar as classes abstratas de super classe. Por exemplo, é sabido que Pessoa Física e Pessoa Jurídica possuem o atributo nome como uma informação em comum. Dentre dezenas de informações, a mais comentada que gera uma grande diferença entre as duas são CPF para Física e CNPJ para Jurídica. Este é um bom motivo para que se defina uma classe abstrata. Ao invés de definir o atributo nome para as duas classes, o que gera redundância, cria-se uma classe abstrata e insere um atributo nome dentro dela. Feito isso, haverá a herança das propriedades para as classes filhas, Física e Jurídica, desta maneira ficam definidas dentro de Física o atributo CPF e para Jurídica o CNPJ. O atributo nome vem automaticamente pela super classe.

```

1. public abstract class Pessoa{
2.     protected String xNome;
3.     protected Pessoa(){
4.         xNome = "Sem nome";
5.     }
6.     protected Pessoa(String nome){
7.         xNome = nome;
8.     }
9.     public String getNome(){
10.         return xNome;
11.     }
12. }
13. public class Fisica extends Pessoa{
14.     private String xCPF;
15.     public Fisica(){
16.         super();

```

```

17. }
18. public Fisica(String nome){
19. super(nome);
20. }
21. public String getCPF(){
22. return xCPF;
23. }
24. }
25. public class Juridica extends Pessoa{
26. private String xCNPJ;
27. public Juridica(){
28. super();
29. }
30. public Juridica(String nome){
31. super(nome);
32. }
33. public String getCNPJ(){
34. return xCNPJ;
35. }
36. public String getNome(){
37. return super.getNome();
38. }
39. }
40. class Principal{
41. public static void main(String[] args){
42. Fisica pessoa1 = new Fisica("Daniel");
43. System.out.println (pessoa1.getNome());
44. System.out.println (pessoa1.getCPF());
45. Juridica pessoa2 = new Juridica();
46. System.out.println (pessoa2.getNome());
47. System.out.println (pessoa2.getCNPJ());
48. }
49. }

```

Em programação orientada a objetos, modificador de acesso, é a palavra-chave que define como um atributo, método, ou classe será visto no contexto que estiver inserido.

Geralmente, utilizam-se modificadores de acesso para privar os atributos do acesso direto, tornando-os privados, e implementa-se métodos públicos que acessam e alteram os atributos. Métodos privados geralmente são usados apenas por outros métodos que são públicos, e que podem ser chamados a partir de outro objeto da mesma classe a fim de não repetir código em mais de um método.

Para utilizar estes modificadores de acesso, basta que o escreva antes do nome do atributo, método ou classe.

- **Public:** O modificador `public` deixará visível a classe ou membro para todas as outras classes, subclasses e pacotes do projeto Java.
- **Protected:** O modificador `protected` deixará visível o atributo para todas as outras classes e subclasses que pertencem ao mesmo pacote. A principal diferença é que apenas as classes do mesmo pacote têm acesso ao membro. O pacote da subclasse não tem acesso ao membro.
- **Private:** O modificador `private` deixará visível o atributo apenas para a classe em que este atributo se encontra.

Package-Private: é o modificador padrão quando outro não é definido. Isto torna acessível na própria classe, nas classes e subclasses do mesmo pacote. Ele geralmente é utilizado para construtores e métodos que só dever ser invocados pelas classes e subclasses do pacote, constantes estáticas que são úteis apenas dentro do pacote em que estive inserido.

Exemplo:

1. `public class MinhaClasse { //classe public`
2. `private int inteiro; //atributo inteiro private`
3. `protected float decimal; //atributo float protected`
4. `boolean ativado; //atributo booleano package-private`
5. `}`

Por padrão, a linguagem Java permite acesso aos membros apenas ao pacote em que ele se encontra. De forma ilustrativa, abaixo está uma tabela demonstrando todas estas características.

Modificador	Classe	Pacote	Subclasse	Globalmente
Public	<i>Sim</i>	<i>Sim</i>	<i>Sim</i>	<i>Sim</i>
Protected	<i>Sim</i>	<i>Sim</i>	<i>Sim</i>	<i>Não</i>
Sem Modificador (Padrão)	<i>Sim</i>	<i>Sim</i>	<i>Não</i>	<i>Não</i>
Private	<i>Sim</i>	<i>Não</i>	<i>Não</i>	<i>Não</i>

Um dos grandes diferenciais da programação orientada a objetos em relação a outros paradigmas de programação que também permitem a definição de estruturas e operações sobre essas estruturas estão no conceito de herança, mecanismo através do quais definições existentes podem ser facilmente estendidas. Juntamente com a herança deve ser enfatizada a importância do polimorfismo, que permite selecionar funcionalidades que um programa irá utilizar de forma dinâmica, durante sua execução.

Encapsulamento: Encapsulamento vem de encapsular, que em programação orientada a objetos significa separar o programa em partes, o mais isolado possível. A idéia é tornar o software mais flexível, fácil de modificar e de criar novas implementações. O Encapsulamento serve para controlar o acesso aos atributos e métodos de uma classe. É uma forma eficiente de proteger os dados manipulados dentro da classe, além de determinar onde esta classe poderá ser manipulada. Usamos o nível de acesso mais restritivo, private, que faça sentido para um membro particular. Sempre usamos private, a menos que tenhamos um bom motivo para deixá-lo com outro nível de acesso. Não devemos permitir o acesso público aos membros, exceto em caso de ser constantes. Isso porque membros públicos tendem a nos ligar a uma implementação em particular e limita a nossa flexibilidade em mudar o código. O encapsulamento que é dividido em dois níveis:

- Nível de classe: Quando determinamos o acesso de uma classe inteira que pode ser public ou Package-Private (padrão);
- Nível de membro: Quando determinamos o acesso de atributos ou métodos de uma classe que podem ser public, private, protected ou Package-Private (padrão).

Então para ter um método encapsulado utilizamos um modificador de acesso que geralmente é public, além do tipo de retorno dele. Para se ter acesso a algum atributo ou método que esteja encapsulado utiliza-se o conceito de get e set. Por definição, com SET é feita uma atribuição a algum atributo, ou seja, define, diz o valor que algum atributo deve ter. E com GET é possível recuperar esse valor.

Exemplo:

1. private String atributo1 = new String();
2. private String atributo2 = new String();
3. public String getAtributo1(){
4. return this.atributo1;
5. }
6. public String getAtributo2(){
7. return this.atributo2;
8. }

Exemplo:

1. public class Pessoa{
2. private String nome;
3. private String sobrenome;
4. private String dataNasc;

```
5. private String rg;
6. private String[] telefones;

7. public String getNome(){
8. return nome;
9. }
10. public void setNome(String n){
11. nome = n;
12. }
13. public String getSobrenome(){
14. return sobrenome;
15. }
16. public void setSobrenome(String s){
17. sobrenome = s;
18. }
19. public String getDataNasc(){
20. return dataNasc;
21. }
22. public void setDataNasc(String d){
23. dataNasc = d;
24. }
25. public String getRg(){
26. return rg;
27. }
28. public void setRg(String r){
29. r = rg;
30. }
31. public String getTelefones(){
32. return telefones;
33. }
34. public void setTelefones(String[] telefones){
35. telefones[] = telefones;
36. }
37. }
```

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. Este mecanismo é muito interessante, pois promove um grande reuso e reaproveitamento de código existente. Com a herança é possível criar classes derivadas, subclasses, a partir de classes bases, superclasses. As subclasses são mais especializadas do que as suas superclasses, mais genéricas. As subclasses herdam todas as características de suas superclasses, como suas variáveis e métodos. A linguagem Java permite o uso de herança simples, mas não permite a implementação de herança múltipla. Para superar essa

limitação o Java faz uso de interfaces, o qual pode ser visto como uma “promessa” que certos métodos com características previamente estabelecidas serão implementados, usando inclusive a palavra reservada `implements` para garantir esta implementação. As interfaces possuem sintaxe similar as classes, no entanto apresentam apenas a especificação das funcionalidades que uma classe deve conter, sem determinar como essa funcionalidade deve ser implementada. Apresentam apenas protótipos dos métodos.

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. 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, bem como reescrever métodos já existentes na superclasse, polimorfismo. 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. Porém não é possível tratar um supertipo como se fosse um subtipo, a não ser que o objeto em questão seja realmente do subtipo desejado e a linguagem suporte este tipo de tratamento, seja por meio de conversão de tipos ou outro mecanismo. Algumas linguagens de programação permitem herança múltipla, ou seja, uma classe pode estender características de várias classes ao mesmo tempo. É o caso do C++. Outras linguagens não permitem herança múltipla, por se tratar de algo perigoso se não usada corretamente. É o caso do Java. Na Orientação a Objetos as palavras classe base, supertipo, superclasse, classe pai e classe mãe são sinônimos, bem como as palavras classe derivada, subtipo, subclasse e classe filha também são sinônimos.

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.

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:

```
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. }
```

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 = "AC33-65";
7.     System.out.println("Nome: " + aluno.nome + "\n" +
8.     "Idade: " + aluno.idade + "\n" +
9.     "Matrícula: " + aluno.matricula);
10.  }
11. }
```

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.

```
1. public abstract class Animal {
2. public abstract void fazerBarulho();
3. }
4. public class Cachorro extends Animal {
5. public void fazerBarulho() {
6. System.out.println("AuAu!");
7. }
8. }
9. public class Gato extends Animal {
10. public void fazerBarulho() {
11. System.out.println("Miau!");
12. }
13. }
14. class Veiculo {
15. public Veiculo() {
16. System.out.print("Veiculo ");
17. }
18. public void checkList() {
19. System.out.println("Veiculo.checkList");
20. }
21. public void adjust() {
22. System.out.println("Veiculo.adjust");
23. }
24. public void cleanup() {
25. System.out.println("Veiculo.cleanup");
26. }
27. }
28. class Automovel extends Veiculo {
29. public Automovel() {
30. System.out.println("Automovel");
31. }
32. public void checkList() {
33. System.out.println("Automovel.checkList");
34. }
35. public void adjust() {
36. System.out.println("Automovel.adjust");
37. }
38. public void cleanup() {
39. System.out.println("Automovel.cleanup");
40. }
41. }
42. class Bicicleta extends Veiculo {
43. public Bicicleta() {
44. System.out.println("Bicicleta");
```

```
45. }
46. public void checkList() {
47. System.out.println("Bicicleta.checkList");
48. }
49. public void adjust() {
50. System.out.println("Bicicleta.adjust");
51. }
52. public void cleanup() {
53. System.out.println("Bicicleta.cleanup");
54. }
55. }
```

Polimorfismo: Polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação, assinatura, mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse. O overload não é um tipo de polimorfismo, pois com overload a assinatura do método obrigatoriamente tem que ter argumentos diferentes, requisito que fere o conceito de Polimorfismo citado acima. De forma genérica, polimorfismo significa várias formas. No caso da Orientação a Objetos, polimorfismo denota uma situação na qual um objeto pode se comportar de maneiras diferentes ao receber uma mensagem, dependendo do seu tipo de criação. Por exemplo, a operação move quando aplicada a uma janela de um sistema de interfaces tem um comportamento distinto do que quando aplicada a uma peça de um jogo de xadrez. Um método é uma implementação específica de uma operação para certa classe. Polimorfismo também implica que uma operação de uma mesma classe pode ser implementada por mais de um método. O usuário não precisa saber quantas implementações existem para uma operação, ou explicitar qual método deve ser utilizado: a linguagem de programação deve ser capaz de selecionar o método correto a partir do nome da operação, classe do objeto e argumentos para a operação. Desta forma, novas classes podem ser adicionadas sem necessidade de modificação de código já existente, pois cada classe apenas define os seus métodos e atributos. Em Java, o polimorfismo se manifesta apenas em chamadas de métodos. A decisão sobre qual o método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução, através do mecanismo de ligação tardia. A ligação tardia ocorre quando o método a ser invocado é definido durante a execução do programa. Através do mecanismo de sobrecarga, dois métodos de uma classe podem ter o mesmo nome, desde que suas assinaturas sejam diferentes, entretanto isso não é polimorfismo. Como dito

anteriormente, tal situação não gera conflito, pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos dos argumentos do método. Nesse caso, diz-se que ocorre a ligação prematura para o método correto. Em Java, todas as determinações de métodos a executar ocorrem através de ligação tardia exceto em dois casos:

1. Métodos declarados como `final` não podem ser redefinidos e, portanto não são passíveis de invocação polimórfica da parte de seus descendentes; e
2. Métodos declarados como `private` são implicitamente finais.

No caso de polimorfismo, é necessário que os métodos tenham exatamente a mesma identificação, sendo utilizado o mecanismo de redefinição de métodos, que é o mesmo que sobrescrita de métodos em classes derivadas. A redefinição ocorre quando um método cuja assinatura já tenha sido especificada recebe uma nova definição, ou seja, um novo corpo, em uma classe derivada. É importante observar que, quando polimorfismo está sendo utilizado, o comportamento que será adotado por um método só será definido durante a execução. Embora em geral esse seja um mecanismo que facilite o desenvolvimento e a compreensão do código orientado a objetos, há algumas situações onde o resultado da execução pode ser não-intuitivo.

Exemplo:

```
abstract class Mamífero {
    public abstract double obterCotaDiariaDeLeite();
}
```

```
class Elefante extends Mamífero {
    public double obterCotaDiariaDeLeite(){
        return 20.0;
    }
}
```

```
class Rato extends Mamifero {
    public double obterCotaDiariaDeLeite() {
        return 0.5;
    }
}
```

```
class Aplicativo {
    public static void main(String args[]){
        System.out.println("Polimorfismo\n");
        Mamifero mamifero1 = new Elefante();
    }
}
```

```
    System.out.println("Cota diaria de leite do elefante: " +  
mamifero1.obterCotaDiariaDeLeite());  
    Mamifero mamifero2 = new Rato();  
    System.out.println("Cota diaria de leite do rato: " +  
mamifero2.obterCotaDiariaDeLeite());  
    }  
}
```